Data structures for current multi-core and future many-core architectures

Eleni Kanellou

▶ To cite this version:

Eleni Kanellou. Data structures for current multi-core and future many-core architectures. Hardware Architecture [cs.AR]. Université de Rennes, 2015. English. NNT : 2015REN1S171 . tel-01256954v2

HAL Id: tel-01256954

https://theses.hal.science/tel-01256954v2

Submitted on 5 Sep 2016

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Eleni KANELLOU

préparée à l'unité de recherche IRISA
Institut de Recherche en Informatique et Système Aléatoires
Université de Rennes 1

Data Structures for Current Multi-core and Future Many-core Architectures

Thèse à soutenir à Rennes
le 14 Décembre 2015

devant le jury composé de :

Prof. Petr KUZNETSOV
INFRES, Telecom ParisTech / *Rapporteur*

Prof. Achour MOSTÉFAOUI
LINA - UFR Sciences et Techniques / *Rapporteur*

Prof. Panagiota FATOUROU
ICS-FORTH & University of Crete / *Examinatrice*

Prof. Hugues FAUCONNIER
LIAFA Paris 7 Denis Diderot / *Examinateur*

Prof. François TAÏANI
IRISA, Université de Rennes 1 / *Examinateur*

Prof. Michel RAYNAL
IRISA, Université de Rennes 1 / *Directeur de thèse*

“No man is an island, entire of itself;
every man is a piece of the continent, a part of the main.”
John Donne, *Devotions upon Emergent Occasions* (1624)

Acknowledgments

I would like to express my deepest gratitude to my thesis director Prof. Michel Raynal, who kindly supervised my PhD process, and to Prof. Panagiota Fatourou, who acted as mentor, guide and co-supervisor. Without their valuable guidance, the elaboration of the present work would not have been possible.

I would like to thank the esteemed members of the jury that agreed to examine my work and provided input and corrections.

I would also like to thank my co-authors in the publications that were elaborated during this thesis, for their fruitful collaboration and inspiring input.

Special thanks goes to my dear colleagues and co-authors Nikos Kallimanis and Christi Symeonidou. Nikos generously shared his ideas and expertise with me and always provided a light-hearted view on the life of the researcher. Christi shared with me both pleasant as well as difficult work moments with enormous empathy. She is a work companion whose attention to detail and perseverance are an inspiring example. Thank you both for our collaborations and the amusing nights spent on paper submissions.

During my work in the TransForm project, I had the chance to meet or work with people that provided interesting conversations and furthered my education in concurrent computing. I would like to thank all members of TransForm and in particular, Prof. Hagit Attiya, Prof. Petr Kuznetsov, and Dr. Sandeep Hans for their kind and helpful interactions with me.

Many thanks go also to the members of the ASAP team at INRIA, with which I spent a very enjoyable part of the PhD time.

I would like to thank my immediate and not-so-immediate family for their support, making a particular mention to my brother, Ilias, an inexhaustible source of unlikely humor and intelligent conversation.

My very tender gratitude is reserved for my parents, Yannis and Eva. They showed unquestioning endurance in the face of each of the decisions that led me to pursuing a PhD degree; they provided loving moral and emotional support and an eager and patient ear during all the times when I thought I would not be able to overcome difficulties and complete a thesis; and, maybe more importantly, they provided vital material support that made practical aspects of everyday life easier for me, allowing me to concentrate on my work. This thesis is dedicated to them and their efforts.

Last, but very far from least, I would like to thank Arnaud. Not only was he a cherished companion and caring colleague during the time this thesis was elaborated, but he also had the kindness of agreeing to assist me in several of the administrative procedures that had to be taken care of for this thesis. I would not have been able to organize the defense without his invaluable practical help. Arnaud, thank you for spiking the hardship of studying for a doctoral degree with so many happy moments!

Contents

Chapter 1

Introduction

1.1 Motivation

Much like the proverbial pebble dropped into the pond, the effects of developments observed in transistor integration during the past decade have rippled through the layers that comprise a computing system, permeating several of its aspects. As such, the increasing number of transistors per area led to the stagnation of the frequency and performance increase of a single processor core, which in turn led to an important paradigm shift in chip design: that of increasing computing power and speed not by diminishing transistor distance on a die, but by including more than one processor core in it.

This trend is so pervasive that it does not sound unreasonable to imagine that soon, one will be hard-pressed to find electronic devices that in fact rely on a single-core processor. Already the range of devices that incorporate multiple processor cores on a chip is broad enough to include devices as mundane as mobile phones [vB09], as critical as big data servers [Hew13], and as innocuous as video gaming platforms [KBLD08]. The *multi-core era* is indubitably here.

More processors imply more processes running in parallel and the continuing advances of technology mean that the potential number of these processes keeps increasing. Following in the spirit of the observation commonly known as 'Moore's law' – i.e. that the number of transistors that can be fit on a chip doubles roughly every two years –, a 'new Moore's law' [Vaj11] predicts that the number of processor cores that are included in a chip will double roughly every two years. While it remains to be seen if this exact formulation will prove accurate, it nevertheless seems that these technology advances will usher in the *many-core era.*

As these advances of technology become more and more integrated into aspects of everyday use, the need arises to program them appropriately. This rippling effect then, initiated in hardware and reaching the software, thrusts concurrent reasoning into the spotlight. Although commonly perceived as a field restricted to a "select few" experts, it is becoming urgent to make it more accessible to the "average programmer". While it is uncertain whether expertise in it will become the additional skill of any software developer, it is nevertheless steadily emerging as an almost necessary tool in exploiting the capabilities that the new hardware has to offer, in order to obtain the desired performance increase that it has been created to provide. Thus, in order to use multi- and many-core architectures, programmers no longer only have to worry about understanding the effects that, for instance, out-of-order execution or the different speed of memory response have on the correctness of their programs. Instead, more and more they have to be aware of competing, concurrent accesses to shared resources and the possibly hazardous effects of asynchrony among processes running in parallel.

Since a typical shared resource of software is the memory, the competition for accessing shared data emerges as the new performance hindrance. On one hand, the speed of access to shared memory does not keep up with the corresponding trends of increase of computing power. On the other hand, as a shared resource between a multitude of processes, it acts as a significant bottleneck. The extent to which these characteristics, if neglected, can exacerbate performance problems in multi- and many-core settings, becomes more apparent, if one considers the cost of maintaining cache coherence. While computing power can be amplified by sharing a workload between several cores, the performance of hardware cache coherence does not keep up with this trend, i.e. does not scale, as the number of cores is increasing. Furthermore, as the only communication medium in such a setting, memory is not only accessed for storing and retrieving raw data and computation results, but it is also accessed in order to set or read meta-data that serve as means of inter-process coordination.

As the aforementioned trends have influenced the perceived upper layer of computing – the software–, the effects of the paradigm shift towards multi- and many-cores seem to have rippled up to the edge of the metaphorical pond, and to rebound again towards their hardware origin: Indeed, there is notable industry momentum supporting the partial or entire abandoning of cache coherence in the near future. A first approach consists in many-core architectures that are composed of so-called coherence islands, i.e. settings in which processor cores within the same island are provided with hardware cache coherence, but where this cache coherence is not ensured across islands. Taking this a step further, prototypes have already been proposed, in which no cache coherence at all is provided [GHKR11, CAB^{+}13, LKL^{+}12]. Furthermore, the network-on-chip (NoC) [DT01] interconnect infrastructure proposes the on-chip routing of messages among cores in such settings, eschewing the reliance on a shared memory or a common bus. In such architectures, communication and coordination among processes must be explicitly carried out. This means that the programmer must bear the additional burden of coding the message sending, handling message reception, and reasoning about load balancing and data distribution among processors.

The picture that is slowly forming, is that of a drastically changing status quo. However, one aspect remains unaffected and this aspect is the inherent difficulty in reasoning about concurrency. This difficulty is not necessarily something subjective that can simply be traced back to the talents – or lack thereof – of the individual programmer. Instead, it stems from the sheer complexity of having to consider so many possible interleavings of accesses to shared resources as may occur in a concurrent environment. Even though the increase of performance is a common goal, the challenges may differ in character, depending on whether one works in a shared memory or message-passing context. As such, in a shared memory environment, one might be more concerned with e.g. avoiding overwhelming cache effects, coping with crash failures, or handling locks correctly. In a message-passing context, it might be more important to e.g. minimize communication overhead, or to tailor it to the underlying architecture. However, concerns in either context originate in the fundamental necessity of providing consistency of data and ensuring an acceptable level of progress.

In view of the fast spreading of multi-core systems, numerous new programming solutions have risen, which not only aim at exploiting the available hardware capabilities but which also aim at providing an easier-to-use abstraction, so that the programming of new machines may be more accessible to the average developer. Software libraries are a notable example. High-productivity languages such as Java provide libraries of concurrent data structures that can be used as black boxes. The programmer can simply invoke the methods of the data structure implementation without needing to worry about explicitly coding the process synchronization that is required for the correct execution of the data structure's operations. Another notable example is *transactional memory* (`TM`) [HM93, ST95]. This paradigm is more general-purpose than that of data structure libraries. It consists of modeling the shared memory as a collection of *transactional data items* and in providing the programmer with the *transaction abstraction.* Accesses to data items that are enclosed in a transaction are guaranteed to happen atomically if the transaction commits; and to not be reflected on the shared memory at all if the transaction aborts. A transactional memory implementation is used by a programmer for this purpose. This implementation includes routines that provide algorithms for initiating and terminating a transaction as well as for accessing the data items. The programmer uses the transactional memory simply by enhancing the sequential code with the transactional routines. The correct handling of concurrency is a task that is taken care of by an expert `TM` designer, who is in charge of coming up with correct `TM` routine implementations that, when executed, synchronize the access to data items in a way that does not violate data consistency.

Practices such as these allow a programmer to exploit current and up-coming architecture without having to reason in depth about concurrency. Using such implementations or libraries means that she can develop software without intimate knowledge of the intricacies of process concurrency and communication or, in some cases, the technical details of the particular architecture on which an application will be run. The correctness of the resulting applications depends on whether the `TM` or library on which it is based, is correctly implemented. While an expert may chose to create from scratch software that is specifically tailored to extensively exploit the characteristics of an architecture, tools like the aforementioned ones are an important asset when it comes to fully using the parallelism that is available, because they make it accessible to the average programmer, because they can be used to create portable applications that do not depend vitally on the underlying architecture, and because they can in many cases facilitate the porting of legacy code from the sequential to the concurrent environment.

1.2 Contributions of this thesis

Section 1.1 describes the context in which the present thesis was elaborated. Our aim was to contribute to a layer of software, which abstracts the hardware to the programmer and facilitates the use of what we consider to be up-and-coming architectures. When designing such algorithms, two fundamental aspects that should be considered are consistency and progress. A consistency condition delimits what are the correct responses for the simulated operations on the shared data. In the spirit of facilitating the use of current architectures to programmers that are used to sequential reasoning, we are interested in consistency conditions that emulate it. Such conditions are usually considered as *strong* or *strict*, since they impose important restrictions on what responses are acceptable, given an interleaving of accesses to shared data. Such a strong condition that concerns concurrent data structure implementations is *linearizability* [HW90], while the popular consistency condition in the context of transactional memory is *opacity* [GK08]. Those conditions require that the responses to concurrent accesses to shared data are equivalent to some sequential execution.

Progress, on the other hand, is concerned with the termination of routines or data structure operations that a process invokes and executes in a concurrent environment. A progress property, then, defines under which circumstances this termination can be provided. By circumstances, we understand factors such as whether processes are prone to failures, i.e. to sudden and unexpected cessation of their execution, or what hardware means they use in order to communicate. A programmer working in a sequential setting may expect from her programs to terminate in any circumstance, provided that the process does not suffer any failure. A strong progress property mimicking this sequential behavior in a concurrent setting, is *wait-freedom* [Her91]. This property ensures that when a process initiates an operation, it can finish it, independently of the speed or possible failure of other processes in the same system.

While aiming to move along those lines of correctness and progress, we considered approaches for making concurrent programming more accessible. We first focused on those architectures that are currently in wide-spread use, namely multi-cores. Assuming that they rely on cache-coherent shared memory, we have elaborated both a transactional memory and a concurrent data structure approach, which provide strong correctness and progress properties.

A `TM` algorithm with wait-free read-only transactions. A versatile tool, transactional memory can both be used to transform sequential data structure implementations into concurrent ones – by wrapping their operations inside transactions – and to write more generalized concurrent applications, liberating programmers from having to reason about means such as locks in order to implement process synchronization.

A common STM research concern regards the avoidance of transaction aborts. Typically, a transaction may abort in scenarios in which it conflicts with another transaction while accessing a data item. A *conflict* between two transactions occurs when they both access the same data item and at least one of those accesses attempts to update it. The abort mechanism aims at preserving data consistency. However, it may result in performance degradation since it translates to “wasted” computation effort. While, ideally, we would like to have TM implementations that guarantee that all transactions commit, recent research [BGK12a] provides an impossibility result which implies that no TM algorithm can achieve this property. This is especially unfortunate when it affects read-only transactions, i.e. transactions which only contain read accesses to data items. Transactions of this type do not modify the shared memory and, as related research shows [GKV07], they represent an important part of transactions in many applications. This includes applications where transactions are used in order to implement concurrent data structures from sequential ones. Ideally, read-only transactions should be as light-weight as possible in terms of synchronization overhead and meta-data that is used for managing the concurrent access of transactions to data items. Attempting to provide this and to favor the committing of read-only transactions, *pessimistic* TM algorithms [AMS12, MS12] use locks. In a pessimistic TM, no transaction ever aborts; However, pessimistic TM decrease parallelism by having *update* transactions, i.e. those that perform updates to transactional variables, execute one after the other. On the other hand, in *optimistic* TM transactions are executed concurrently and they commit only if they have not encountered any conflict during their execution.

In order to address the drawbacks of those approaches, we introduce WFR-TM, a TM algorithm that attempts to combine desirable characteristics of both optimistic and pessimistic TM implementations. In WFR-TM, *read-only* transactions are *wait-free*, i.e. they always commit within a finite number of their own steps. In the interest of being light-weight, they never execute expensive synchronization operations (such as CAS, etc.). These desirable characteristics of read-only transactions are achieved without sacrificing the parallelism between update transactions. Update transactions “pessimistically” synchronize with concurrently executed read-only transactions, in that they wait for such transactions to complete. However, they behave “optimistically” when they coordinate with each other: they are executed concurrently in a speculative way, and they commit if they have not encountered any conflict with other update transactions during their execution. In the spirit of providing the programmer with a correct STM implementation, we formally prove that WFR-TM satisfies opacity and provides wait-freedom for read-only transactions, while ensuring that update transactions deadlock-free.

Wait-free concurrent data structure with complex operations. In a related avenue, we further delved into providing ease of programmability for multi-cores through the use of concurrent data structure implementations. Specifically, we studied concurrent data structures that support enhanced functionality by providing complex read-only operations. Contrary to a data structure’s traditional operations, complex read-only operations are useful for obtaining a partial or total consistent view of it, i.e. a reading of the state of the data structure – either in its entirety or only a subset of it – at a particular moment. Obtaining such a view is trivial in the sequential setting, since a single process accesses the data structure. In the concurrent setting, however, an operation trying to obtain that view, also sometimes referred to as *iterator* or *snapshot*, may run in parallel with other operations by other processes that update it. Currently, several concurrent implementations of well-studied data structures, such as lists, queues, stacks, trees, or skip-lists are provided in the literature and research in this direction continues to emerge. Such a concurrent implementation provides the data structure's basic operations through algorithms which take into account that multiple processes may be accessing it, and therefore, take care of their synchronization.

Consider such a concurrent implementation that provides the structure's basic operations but that does not provide one for obtaining a consistent view. In case a programmer requires to implement this operation herself, the desired effect of facilitating concurrent programming is negated: The programmer suddenly has to delve into the synchronization details of the structure and explicitly code them in her implementation. So, more and more effort is devoted to enhancing concurrent data structures with such an operation. Implementations of such data structures can be included in programming language libraries and be used for synchronization, once more without requiring high concurrency expertise from the programmer. Although the functionality is more restricted, their design thus faces similar challenges as those met when designing `TM` algorithms. However, it also provides similar benefits.

We take on this challenge by addressing the problem of designing a complicated concurrent data structure: We present `Dense`, a concurrent, edge-oriented, weighted graph implementation. `Dense` incorporates the capability of taking dynamic, partial snapshots of the edges of the graph. We provide this capability by introducing a novel model for the graph data structure, which defines the following two functionalities: An update to the graph adds or removes an edge, or modifies an edge's weight. A *dynamic traversal* takes a snapshot of a subset of the graph's edges, exhibiting the particular characteristic that this subset can be determined dynamically while the traversal is taking place. Updates and dynamic traversals can run concurrent to each other. By exhibiting transaction-like behavior, our dynamic traversal is a versatile function that can implement a variety of different graph traversal patterns. At the same time, despite the similarity to STM, the model is restricted enough to avoid the abort semantics associated with transactions. This is important because it helps ensure that graph operations are both linearizable and wait-free.

Distributed Data Structures for non-cache coherent architectures. Having provided solutions for architectures that are currently in use, we tackle similar issues for many-cores, i.e. architectures that we expect to be dominant in the future. Since cache-coherence does not scale well as the number of cores increases [NDB$^+$14], future many-core architectures, which will offer hundreds or even thousands of cores, are expected to substitute full cache-coherence with multiple coherence islands, each comprised of several cores sharing a part of the memory, without, however, providing hardware cache coherence among cores of different islands. Instead, the coherence islands will be interconnected using fast communication channels. This trend is expected to go as far as to forgo cache coherence all together – this is the case in fully non cache-coherent architecture prototypes, such as the SCC [HDH$^+$10] and the Runnemede [CAB$^+$13]. Thus, while the previous two thesis contributions that we presented concern architectures relying on shared memory, the final approach pursued by this thesis assumes a message-passing communication infrastructure.

Just as the shared memory context, however, message-passing is considered difficult to program, as it often requires highly skilled and experienced programmers to reason about load balancing, distributing data among processors, explicit communication and synchronization, not only to achieve the best of performance, but even to ensure simple correctness of a program. In the interest of helping software developers to overcome these difficulties, we consider that in this setting also, the design of effective distributed data structures is crucial for many applications.

In the context of this thesis, we study general techniques for implementing distributed versions of data structures intended for many-core architectures with non- or partially cache-coherent memory and we highlight how they can be applied by providing implementations of essential data structures such as stacks, queues, and lists.

Once more, we aim at leaving the habitual paradigm on the programmer's side intact. A programmer using these algorithms should be able to just invoke operations without having to be concerned about the communication infrastructure. On the contrary, the implementations of the data structure operations that we provide are meant to be tailored to suit the available hardware – e.g. the communication or connectivity characteristics of the cores that comprise it, etc. – or the expected workload – e.g. the estimated locality of the data, whether the operations on the data structure are mostly reads or updates, etc. Thus, the techniques that these algorithms are based on exhibit different properties because they are meant to serve different purposes, such as addressing different workloads, or exploiting the communication characteristics of a given architecture.

We focus on two techniques. We first present a *directory-based* approach, where the elements that comprise the data structure are stored in a distributed directory. In this technique, a synchronizing server acts as the coordinator that indicates where those elements should be stored or retrieved from. This approach achieves load-balancing in situations where the data structure is large. However, the manner in which the elements of the data structure are handled, remains "hidden" from the programmer. The same is true for the *token-based* approach, the second design technique that we present. In our token-based algorithms, elements that comprise the data structure are stored in a designated set of cores, which furthermore form a ring. One of them acts as the *token-server* so that storing and retrieving data structure elements takes place on its memory module. If the memory module of this core fills up, but the core receives requests to store more elements, then it forwards the token to the next core in the ring. Similarly, if the requested operations on the data structure have emptied the token server's memory module of data structure elements, but the core receives requests to remove more, then it forwards the token to the previous core in the ring. This approach exploits data locality and is better suited for cases where the data structure size is moderate.

Apart from aiding in making many-core architectures more accessible to programmers that are accustomed to sequential programming, an added benefit of our implementations is that they can facilitate the re-use of applications designed for shared memory. The algorithms we present are intended as a step towards providing libraries of data structures adapted to message-passing infrastructures. Shared-memory applications that rely on equivalent data structure libraries could then be ported to a message-passing setting by substituting one library for another. Notably, research effort has been devoted [ABH$^+$01, MS10, NGF08, YC97, ZWL02] into implementing distributed run-time environments for high-productivity languages such as Java. While these implementations assume non-cache-coherence, they nevertheless maintain the shared-memory abstraction towards the programmer. The data structures that we provide correspond to several of the data structures that are included in the Java concurrency utilities package [Lea06, Ora] and could be used to substitute it.

Our implementations and their claimed performance properties have been experimentally tested on a non cache-coherent 512-core architecture, built using the FORMIC hardware prototype boards [LKL$^+$12]. Furthermore, in the interest of providing a basis on which to create correct software applications, we provide proofs that our data structure implementations are linearizable.

1.2.1 List of Publications

In the process of elaborating the present thesis, the following publications were produced.

1. Tyler Crain, Eleni Kanellou, Michel Raynal. STM Systems: Enforcing Strong Isolation between Transactions and Non-transactional Code. 12th International Conference on Algorithms and Architectures for Parallel Processing - ICA3PP 2012.

2. Panagiota Fatourou, Eleni Kanellou, Eleftherios Kosmas, Md Forhad Rabbi. WFR-TM: Wait-free Readers without Sacrificing Speculation of Writers. 18th International Conference on Principles of Distributed Systems - OPODIS 2014.

3. Dmytro Dziuma, Panagiota Fatourou, and Eleni Kanellou. "Consistency for Transactional Memory Computing". In "Transactional Memory: Foundations, Algorithms, Tools and Applications" (COST Action Euro-TM Book, page 3).

4. Panagiota Fatourou, Mykhailo Iaremko, Eleni Kanellou, and Eleftherios Kosmas. "Algorithmic Techniques in STM Design". In "Transactional Memory: Foundations, Algorithms, Tools and Applications" (COST Action Euro-TM Book, page 101).

5. Dmytro Dziuma, Panagiota Fatourou, and Eleni Kanellou. "Consistency for Transactional Memory Computing". Bulletin of the EATCS, No. 113, June 2014 (http://bulletin.eatcs.org/index.php/beatcs/article/view/288)

6. Panagiota Fatourou, Nikolaos D. Kallimanis, Eleni Kanellou, Odysseas Makridakis, Christi Symeonidou. Distributed Data Structures for Future Many-core Architectures. FORTH-ICS Technical Report TR-447.

7. Nikolaos D. Kallimanis, Eleni Kanellou. Wait-free Concurrent Graph Objects with Dynamic Traversals. 19th International Conference on Principles of Distributed Systems - OPODIS 2015.

Contents that concern publications 2 to 7 comprise the body of this thesis.

1.3 Roadmap

The chapters that follow provide details on the work that was elaborated for this thesis. Chapter 2 introduces the formal model under which we view the two architecture paradigms and details the hardware assumptions that we make. Furthermore, it provides the definitions of the theoretical concepts that we employ in designing the presented algorithms and in proving their correctness. Chapter 3 then presents our solutions for current multi-core architectures. Specifically, Section 3.1 introduces transactional memory and WFR-TM, the TM implementation that we propose. Section 3.2 then presents the data structure model and the implementation of Dense, a wait-free concurrent graph with complex read-only operations. Section 3.3 concludes this chapter by reviewing state-of-the-art literature relevant to transactional memory and data structure implementations for the shared memory context, i.e. what we perceive as the communication paradigm of current multi-core machines. The data structure implementations that we propose for the many-core setting are detailed in Chapter 4. Specifically, the directory-based approach is explored in Section 4.1, while the token-based approach is presented in Section 4.2. Although our focus through-out this thesis is mostly on correctness, we dedicate Section 4.4 to providing a summary of the experimental evaluation of the data structure implementations, exposed in of some design techniques that are meant to exploit many-core hardware characteristics even deeper, in the interest of achieving higher performance. Section 4.5 reviews the related literature in distributed data structure design. Chapter 5 concludes this work by summarizing the main contributions, discussing their implications, and sketching out possible directions for future work.

Chapter 2

System Model and Definitions

In this chapter, we provide a formal model for the shared memory and the message-passing systems that we target and elaborate on our view of the hardware where necessary. The assumptions presented here will serve as basis for the algorithms presented in subsequent chapters. These assumptions aim at reflecting the communication reality of current, cache-coherent multicore architectures and the communication expectation for future, non-cache-coherent many-core architectures. In either of the communication paradigms that we consider, we assume a system of n asynchronous processes, i.e. processes that execute at arbitrary speeds. We denote processes as p_i, $i \in \{1, 2, \ldots, n\}$. We consider that each process acts as a state machine, i.e. it executes a single sequential program. However, multiple different processes can execute concurrently.

The following sections further elaborate on how we model the two communication paradigms and highlight their differences. Section 2.1 details our shared memory model, in which we also provide definitions for the *transactional memory* abstraction. Section 2.2 and Section 2.3 provide definitions of consistency and progress properties, respectively. Our message-passing model is described in Section 2.4. Finally, Section 2.5 provides an overview of pseudocode conventions that are employed throughout this work.

2.1 Shared Memory Systems

Hardware assumptions. In this paradigm, all processors are integrated onto a single chip. We do not make an explicit assumption of homogeneity of those processors, although we have to at least assume that even in the case of heterogeneity, the instruction sets of the different types of cores overlap into a subset that contains all those primitives that are used in our algorithms. We assume input and output devices through which the processes communicate with the environment but are not further concerned with modeling them. Processes communicate via a shared memory. We make no further assumptions about memory, i.e. whether it is on-chip or not, whether processors have their individual cache or not, etc., other than that processes operate in a system that provides full cache coherence.

Base objects. We model the shared memory as a finite collection of base objects, which we consider that are provided by the system hardware. A *base object* has a state and supports a set of operations, called *primitives*, to read or update its state. In this work, we make use of the objects detailed below and consider that the execution of a primitive by a process occurs atomically.

- A *read/write object* O stores a value v from some set S and supports two atomic primitives `read` and `write`; `read`(O) returns the current value of O without changing it, and `write`(O, v) writes the value v into O and returns an acknowledgement.

- A `CAS` object O stores a value v from some set S and supports, in addition to `read`, the atomic primitive `CAS`(O, v', v) which checks whether the value of O is v' and, if so, it sets the value of O to v and returns `true`, otherwise, it returns `false` and the value of O remains unchanged.

- An `Add` object O has a state that takes values out of some set of integers S and supports, in addition to `read`, the atomic primitive `Add`(O, v), $v \in S$, which (arithmetically) adds the value v to the state of O.

- An `LL/SC` object O has a state that takes values out of some set S. It provides the primitives `LL`(O) and `SC`(O, v), $v \in S$. `LL`(O) returns the current state of O. `SC`(O, v), executed by a process p_i, $i \in \{1, 2, \ldots, n\}$, must follow an execution of `LL`(O) also by p_i. It changes the state of O to v if O has not changed since the last execution of `LL`(O) by p_i.

Concurrent data structures. A concurrent data structure also has a state, which is stored in shared base objects. Furthermore, for each process, it provides algorithms for each operation the data structure supports. A process executes an operation by issuing an *invocation* for it and an operation terminates by returning a *response* to the process.

Transactions, t-operations, histories. A *transaction* executes a piece of sequential code which accesses (reads or writes) pieces of data, called *data items.* A data item may be accessed by several processes simultaneously when a transaction is executed in a concurrent environment. A transaction may *commit*, in which case all its updates to data items take effect, or *abort*, in which case all its updates are discarded.

A *software transactional memory (STM) algorithm* uses a collection of base objects to store the state of data items and data (referred to as *meta-data*) used to manipulate transactions. In order to facilitate their concurrent access, data items have a shared representation, which is referred to as a *transactional variable*, or *t-variable.* In the remainder of this work, we may abuse terminology and use t-variable in order to refer to both the shared representation and the data item itself. For the purposes of this work, we consider that accesses to data items consist in *reads* and *writes.* The data items that a transaction reads are referred to as its *read-set.* The data items a transaction writes to are referred to as its *write-set.* The union of read-set and write-set is referred to as the *data-set.*

Processes can access the transactional memory through a collection of operation that it offers, which we refer to as *transactional operations* or *t-operations*. An STM algorithm provides implementations for t-operations. In this work, we are concerned with implementations for the following t-operations:

- BeginTx. It initiates a transaction T and returns either a reference to T or a special value A_T indicating that T has to abort;

- CreateTvar. It creates the shared representation of a newly allocated data item and either returns a reference to that shared representation or A_T;

- ReadTvar. It receives as argument the t-variable x to be accessed (and possibly the process p invoking ReadTvar and the transaction T for which p invokes ReadTvar) and returns either a value v for x or A_T.

- WriteTvar. It receives as arguments the t-variable x to be modified, a value v (and possibly the process p invoking WriteTvar and the transaction T for which p invokes WriteTvar), and returns either an acknowledgment or A_T.

- CommitTx. It is invoked after all t-variable accesses of a transaction have been performed, in order to attempt to effectuate the transaction's changes: if it finds that the execution of the transaction is correct, then the transaction commits, and a special value C_T is returned; otherwise the transaction aborts and A_T is returned.

- AbortTx. It is invoked in order to abort a transaction and it always returns A_T.

A t-operation starts its execution when the process executing it issues an *invocation* for it; the t-operation completes its execution when the process executing it receives a *response*. If the t-operations are invoked by the process while it is executing transaction T, we may say for simplicity that T invokes the operations. We refer to C_T as the *commit response* and to A_T as the *abort response*. Either of C_T and A_T may be referred to as the *response* of T.

Notice that although the transaction abstraction gives the illusion of atomicity, nevertheless, the execution of a t-operation *op* is not atomic, i.e. the process executing it may perform a sequence of primitives on base objects in order to complete it. We remark that these invocations and responses are considered atomic. We refer to t-operation invocations and responses as *events*.

A *history* is a finite sequence of t-operation invocations and responses. Given some history H, we say that a transaction T (executed by a process p) *is in* H or H *contains* T, if there are invocations and responses of t-operations in H that are issued (or received) by p for T. The *transaction subhistory* of H for T, denoted by $H|T$, is the subsequence of all events in H issued by p for T. The *process subhistory* of H for a process p, denoted by $H|p$, is the subsequence of all events in H issued by p. We say that a response *res* in some history H *matches* an invocation *inv* of a t-operation *op* in H, if they are both by the same process p, *res* follows *inv* in H, *res* is a response for *op*, and there is no other event by p between *inv* and *res* in H.

A history H is *well-formed* if, for each transaction T in H, $H|T$ is an alternating sequence of invocations and matching responses, starting with an invocation of BeginTx, such that the following hold: (1) no event in $H|T$ follows C_T or A_T; (2) if T' is some transaction in H that is also executed by the same process that executes T, then either the last event of $H|T$ precedes in H the first event of $H|T'$ or the last event of $H|T'$ precedes in H the first event of $H|T$. For the remainder of this work, we focus on well-formed histories. Consider such a history H. A t-operation is *complete* in H, if there is a response for it in H; otherwise, the t-operation is *pending*. A transaction T is *committed* in H, if $H|T$ includes C_T; a transaction T is *aborted* in H, if $H|T$ includes A_T. A transaction is *complete* in H, if it is either committed or aborted in H, otherwise it is *live*. A transaction T is *commit-pending* in H if T is live in H and $H|T$ includes an invocation to CommitTx for T. If $H|T$ contains at least one invocation of WriteTvar, T is called an *update* transaction; otherwise, T is *read-only*. We denote by $comm(H)$ the subsequence of all events in H issued and received for committed transactions.

For each process p, we denote by $H|p$ the subsequence of H containing all invocations and responses of t-operations issued or received by p. Two histories H and H' are *equivalent*, if for each process p, $H|p = H'|p$. This means that H and H' are equivalent if they contain the same set of transactions, and each t-operation invoked in H is also invoked in H' and receives the same response in both H and H'. Then, even though the order of invocation and response events may be different in H' compared to H, nevertheless the orders of invocation and response events are the same in $H|p$ and $H'|p$ for each process p.

We denote by $Complete(H)$ a set of histories that extend H. Specifically, a history H' is in $Complete(H)$ if and only if, all of the following hold:

1. H' is well-formed, H is a prefix of H', and H and H' contain the same set of transactions;
2. for every live transaction[1] T in H:
   (a) if $H|T$ ends with an invocation of CommitTx, H' contains either C_T or A_T;
   (b) if $H|T$ ends with an invocation other than CommitTx, H' contains A_T;
   (c) if $H|T$ ends with a response, H' contains Abort$_T$ and A_T.
3. H' does not contain any other additional events.

Roughly speaking, each history in $Complete(H)$ is an extension of H where some of the commit-pending transactions in H appear as committed and all other live transactions appear as aborted. We say that H is *complete* if all transactions in H are complete. Each history in $Complete(H)$ is complete.

Executions. Each process has an internal state. A *configuration* C is a vector that describes the system at some point in time, i.e. it provides information about the state of each process

[1] We remark that the order in which the live transactions of H are inspected to form H' is immaterial, i.e. all histories that result by processing the live transactions in any possible such order are added in $Complete(H)$.

and the state of each base object. In an *initial configuration*, the states in which processes and base objects are in are referred to as *initial states.* We denote an initial configuration by C_0.

A *step* of a process consists of applying a single primitive on some base object, the response to that primitive, and zero or more local computation performed by the process; local computation accesses only local variables of the process, so it may cause the internal state of the process to change but it does not change the state of any base object. As a step, we also consider the invocation of a routine or of a data structure operation, as well as the response to such an invocation. We consider that each step is executed atomically. A step is also considered as an event.

A (possibly infinite) sequence $C_0, \phi_1, C_1, \ldots, C_{i-1}, \phi_i, C_i, \ldots$, of alternating configurations (C_k) and events (ϕ_k), starting from C_0, where for each $k \geq 0$, C_{k+1} results from applying event ϕ_{k+1} to configuration C_k, is referred to as an *execution.* A subsequence of an execution α in the form $C_i, \phi_{i+1}, C_{i+1}, \ldots, C_j, \phi_{j+1}, C_{j+1}$, of alternating configurations and events, starting from some configuration C_k, $k > 0$, is referred to as an *execution interval* of α.

If some configuration C occurs before some configuration C', $C \neq C'$, in an execution α, then we say that C *precedes* C' in α and denote it as $C < C'$. Conversely, we say that C' *follows* C in α. Using the same terminology and operator, we also denote the precedence relation that α imposes between an event ϕ_i and an event ϕ_j, or precedence among an event ϕ_i and a configuration C_j. Notice that for the remainder of this thesis, we only consider executions where the invocation of an operation precedes its response.

Let α_1 and α_2 be two execution intervals of some execution α. If the last configuration of α_1 precedes or is the same with the first configuration in α_2, then we say that α_1 precedes α_2 and denote it $\alpha_1 < \alpha_2$. In that case we also say that α_2 follows α_1. If neither $\alpha_1 < \alpha_2$ nor $\alpha_2 < \alpha_1$ are the case, then we say that α_1 and α_2 *overlap.*

Given the instance of some operation *op* for which the invocation and response events are included in α, we define α_{op}, the execution interval of *op*, as that subsequence of α which starts with the configuration in which *op* is invoked and ends with the configuration that results from the response of *op.* We refer to such an operation as *completed.* If only the invocation of an operation *op* is included in α, then the execution interval of *op* is the suffix of α that starts with the configuration in which *op* is invoked. In that case, we say that *op* is *incomplete.* If there are no two operation instances op_1, op_2 in α for which the execution intervals overlap, then we say that α is a *sequential* execution, or that operations in α are executed *sequentially.*

Given an execution α, the history of α, denoted by H_α, is the subsequence of α that only consists of the invocations and the responses of t-operations. Given a complete transaction T in α, we define the *execution interval* of T as the subsequence of consecutive steps of α starting with the configuration in which T is invoked and ending with the configuration that results from the response of T. The execution interval of a transaction T that does not complete in α is the suffix of α starting with with the configuration in which T is invoked. A t-operation is complete in α if it is complete in H_α; otherwise it is pending. A transaction T is committed (res. live, commit-pending) in α if it is committed (res. live, commit-pending) in H_α.

A well-formed history imposes a partial order, referred to as *real-time order*, on the set of transactions it contains. We denote the real-time order as $<_H$ and defined as that partial order, for which it holds that for any two transactions T_1 and T_2 in H, if T_1 is complete in H and the last event of $H|T_1$ precedes the first event of $H|T_2$ in H, then $T_1 <_H T_2$. Transactions T_1 and T_2 are *concurrent* in H, if neither $T_1 <_H T_2$ nor $T_2 <_H T_1$. Similarly, T_1 and T_2 are *concurrent* in an execution α, if neither $T_1 <_{H_\alpha} T_2$ nor $T_2 <_{H_\alpha} T_1$. A history H is *sequential* if no two transactions in H are concurrent. Given two well-formed histories H_1 and H_2 which contain the same set of transactions, we say that H_2 *respects* the real-time order of H_1, if for any two transactions T_1 and T_2 that are both in H_1 and in H_2, it holds that if $T_1 <_{H_1} T_2$, then also $T_1 <_{H_2} T_2$.

2.2 Correctness

Legality. A transaction T in a sequential history S is *legal* if for every invocation inv of ReadTvar on each data item x that T performs, whose response is $res \neq A_T$, the following hold: (i) T contains an invocation of WriteTvar for x by T that precedes inv, in which case res is the value used as argument of the last such invocation; or in case (i) does not hold, if (ii) S contains a committed transaction T', which contains an invocation of WriteTvar for x, in which case res is the value used as argument of the last such t-operation invocation by a committed transaction that precedes T in$S$; or in case neither (i) nor (ii) hold, if (iii) res is the initial value for x. A complete sequential history S is *legal* if every transaction in S is legal.

Consistency conditions. Strict serializability is traditionally considered a basic consistency condition for concurrent transaction execution. Although it originates in database systems, we reformulate it for transactional memory.

Definition 2.1 (Strict Serializabiltiy [Pap79]). *A history H is strictly serializable, if there exist a history $H' \in Complete(H)$ and a legal sequential history S such that S is equivalent to $comm(H')$ and S respects $<_{comm(H')}$. An execution α is strictly serializable, if H_α is strictly serializable. An STM algorithm is strictly serializable, if each execution α it produces is strictly serializable.*

In order to define correctness for concurrent data structures, we need to reason about operations. A condition that is reminiscent to strict serializability, but applies to data structure operations, is linearizability. Roughly speaking, if transactions were restricted to containing only one access to a data item, then strict serializability and linearizability would be equivalent.

Definition 2.2 (Linearizability [HW90]). *An execution α is linearizable if it is possible to assign a linearization point inside the execution interval of each completed operation in α and possibly some of the incomplete operations in α, so that the result of each of those operations is the same as it would be, if they had been performed sequentially in the order dictated by their linearization points.*

A concurrent data structure is linearizable if all its executions are linearizable.

Given the particularities of STM algorithms when compared to databases, currently a condition derivative of but stricter than strict serializability is commonly used in the TM context.

Definition 2.3 (**Opacity** [GK08]). *A history H is opaque if there exists some history H' in $Complete(H)$, and a legal sequential history S such that S is equivalent to H' and S respects the real-time order of H'. An execution α is opaque if H_α is opaque and a TM algorithm is opaque if all executions that it produces are opaque.*

In contrast to strict serializability, opacity does not only impose restrictions on transactions that are committed or commit-pending, but what is more, it implies restrictions for transactions that are live or aborted.

2.3 Progress

Liveness assumptions. In this work, we consider that processes that participate in an execution α may suffer from *crash failures*, i.e. we consider that a process may unexpectedly stop taking steps in α after some configuration C.

Progress properties. We say that a process p executes solo in some execution interval α' of some execution α it participates in, if during that interval, the only process that takes steps is p. We say that a process suffers from *starvation* in an infinite execution α if after some configuration C it does not receive a response to an operation it has invoked before C, even though it keeps taking steps after C. We say that a system suffers from *deadlock* if in an infinite execution α, there is a configuration C after which no process receives a response to an operation that it has invoked, even though the process continues taking steps after C.

In this context, we consider that a liveness condition concerns completion of operations or of transactions. With this criterion, the following definitions list some progress properties from weakest to strongest.

Definition 2.4 (**Obstruction-freedom** [HLM03]). *A data structure implementation or STM is obstruction-free if in any execution α that it produces, each process can finish the execution of its operation, provided that it can run solo after some configuration C for a sufficient number of steps.*

The following definition refers to what is also known as the *non-blocking property.*

Definition 2.5 (**Lock-freedom**). *A data structure implementation or STM is lock-free if in any execution α that it produces, then starting from any configuration C in α, some process that does not suffer a crash failure is able to terminate within a finite number of steps, the operation it was executing at C or an operation it invokes after C, if at C it wasn't executing any.*

The above definition implies that in case α is an infinite execution, then infinitely many invoked operations finish their execution, each within a finite number of steps independently of the speed or the failure of other processes.

Definition 2.6 (**Wait-freedom** [Her91]). *A data structure implementation or STM is wait-free if in any execution α that it produces, each participating process that does not suffer a crash failure finishes the execution of every operation or t-operation that it initiates within a finite number of steps, independently of the speed or the failure of other processes.*

2.4 Message-Passing

We consider that this model corresponds to many-core architectures and use the following paragraphs in order to outline the important differences to the shared memory model. Nevertheless, several of those definitions as well as consistency and progress conditions apply to both contexts.

Hardware assumptions. Inspired by the characteristics of non cache-coherent architectures [CAB$^+$13] and prototypes [LKL$^+$12], we consider an architecture which features *m islands* (or *clusters*), each comprised of c cores (located in one or more processors). The main memory is split into modules, with each module associated to a distinct island (or core). A fast cache memory is located close to each core. No hardware cache-coherence is provided among cores of different islands: different copies of the same variable residing on caches of different islands may be inconsistent. The islands are interconnected with fast communication channels. The architecture may provide cache-coherence for the memory modules of an island to processes executing on the cores of the island, i.e. the cores of the same island may see the memory modules of the island as cache-coherent shared memory. If this is so, we say that the architecture is *partially non cache-coherent*; otherwise, it is *fully non cache-coherent*.

Messages. We consider that each process has a *mailbox*, implemented as a hardware FIFO queue. A process can send messages to other processes by invoking `send` and it can receive messages from other processes by invoking `receive`. We further assume that messages are not lost and that they are delivered in order. An invocation of `receive` blocks until the requested message arrives. The first parameter of an invocation to `send` determines the core identifier to which the message is sent. We assume that the maximum message size supported by an architecture is generally either equal to a few memory words or a cache line.

Additional communication mechanisms. In order to facilitate communication that involves data that exceeds the maximum message size, we assume that *Direct Memory Access (DMA)* is available. A DMA engine allows certain hardware subsystems to access the system's memory without any interference with the CPU. We assume that each core can perform Dma(A, B, d) to copy a memory chunk of size d from memory address A to memory address B using a DMA (where A and B may be addresses in local or a remote memory module). We remark that DMA is not executed atomically. To model a DMA, we can assume that it consists of a sequence of atomic reads of smaller parts (e.g. one or more words) of the memory chunk to be transferred, and atomic writes of each of these parts to the other memory module.

Remote DMA transfers can be used as a performance optimization mechanism: once the size of the memory chunk to be transferred becomes larger (by a small multiplicative factor) than the maximum message size supported by the architecture, it is more efficient to realize the transfer using DMA (in comparison to sending messages).

Distributed data structures. An implementation of a data structure (DS) stores its state in the memory modules and provides an algorithm, for each process, to implement each operation supported by the DS. For correctness, we consider linearizability. We aim at designing algorithms that always terminate, i.e. reach a state where all messages sent have been delivered and no step is enabled. In this work, we do not cope with message or process failures in the message-passing context.

Executions, steps, events. We model the submission and delivery of messages sent by processes by including incoming and outgoing message buffers in the state of each process (as described in distributed computing books [AW04, Lyn96]). As in the shared memory case, a *configuration* is a vector describing the state of each process. In addition to the shared memory definition, however, the state includes the message buffers,the state of the caches (or the shared variables in case shared memory is supported among the cores of each island) and the states of the memory modules. In an *initial configuration*, each process is in an initial state, the shared variables and the memory modules are in initial states and all message buffers are empty.

An *event* can be either a step by some process, or the delivery of a message; in one step, a process may either transmit exactly one message to some process and at least one message to every other process, or access (read or write) exactly one shared variable. An execution is an alternating sequence of configurations and steps, starting with an initial configuration. A step is *enabled* at a configuration C, if the process will execute this step next time it will be scheduled. Execution intervals are defined as in the shared memory context.

Communication Complexity. Communication between the cores of the same island is usually faster than that across islands. Thus, the communication complexity of an algorithm for a non cache-coherent architecture must be measured in two different levels, namely the intra-island communication and the inter-island communication. The *intra-island communication complexity* of an operation *op* in an implementation I is the maximum, over all executions of I and over all instances of *op* in each execution, of the total number of messages sent by every core to cores residing *on different islands* for executing this instance of *op*. If the architecture is fully non cache-coherent, then the *inter-island communication complexity* of an operation *op* in I is the maximum, over all executions of I, over all instances of *op* in each execution and over all islands, of the total number of messages sent by every core of the island to cores residing *on the same island* for executing this instance of *op*; in case of a partially non cache-coherent architecture, it is the maximum, over all executions, over all instances of *op* in each execution and over all islands, of the total number of cache-misses that the cores of the island experience to execute this instance of *op* (this is known as the *cache-coherence (CC) model* [HS08, MCS91]).

Time complexity. We define the time complexity of an operation in an implementation I based on timed versions [Lyn96] of executions of I, where times are assigned to events as follows: (1) the times must start at 0, (2) must be strictly increasing for each individual process, (3) must increase without bound if the execution is infinite, (4) the timestamps of two subsequent events by the same process must differ by at most 1, and (4) the delay of each message sent must be no more than one time unit.

2.5 Conventions for Algorithm Presentation

The algorithms contained in this work are expressed by means of C-like pseudocode. Statements terminate by a new line, thus rescinding of semi-colons. Scope is indicated through indentation and rescinds of the use of brackets. We use the symbol $=$ in order to indicate value assignment, while we use the symbol $==$ to indicate equality check (as in conditional statements). Conversely, the symbol $\neq$ indicates check for inequality. We further adopt the C-like operators $--$ and $++$ in order to indicate that a variable is decremented or incremented by 1, respectively. Operations and procedures that might be explicitly written out as a small algorithm in an actual programming language are abstracted in our case, for ease of presentation. Instead, we use common mathematical symbols (such as $\cup$, $\in$, $\notin$, etc) to indicate them. The symbol $\emptyset$ denotes an empty set. Pseudocode may be annotated with comments. In this case, those are indicated by preceding them with // if they span no more than one text line, or by including them between $/*\ldots*/$, if they span several lines.

Chapter 3

Data Structures for Multi-core Architectures Supporting Cache-coherence

In this chapter, we focus on current, cache-coherent architectures.

In Section 3.1, we present WFR-TM, an opaque transactional memory algorithm that ensures that read-only transactions execute exactly once and finish by committing. WFR-TM combines desirable characteristics of the optimistic and the pessimistic concept.

In Section 3.2, we present `Dense`, a concurrent graph implementation with linearizable and wait-free operations. An interesting feature of this graph is that it provides the capability of performing traversals that are dynamically defined. In those *dynamic traversals*, the subset of the graph that is to be visited can be defined at runtime by a process. Nevertheless, a consistent snapshot of the subgraph that was visited is returned. In this aspect, this graph implementation exhibits transaction-like characteristics, where the dynamic traversals resemble memory transactions that can however not be aborted.

Finally, a review of related transactional memory and concurrent data structure literature is presented in Section 3.3.

3.1 Case Study I: WFR-TM, A TM Algorithm

In WFR-TM, a read-only transaction T_r announces itself so that update transactions are aware of its existence. If T_w is an update transaction that updates t-variable x after T_r announced itself, then T_w can only commit after T_r does. This prevents T_w from updating x after T_r has read it. Update transactions may execute in parallel to each other, but may have to abort if they encounter conflicts. In order to detect those, update transactions employ fine-grained locking on the t-variables that they access. A read-only transaction that accesses a locked t-variable can read its value by *snooping* into the write-set of the transaction that has locked it. We remark that it is not necessary to know in advance whether a transaction is read-only; any transaction is read-only when it is initiated and becomes an update transaction the first time it accesses a t-variable for write. WFR-TM satisfies opacity and provides wait-free read-only and deadlock-free update transactions.

In the following, we provide a detailed description of WFR-TM and a formal proof of the properties we claim for it. In order to do so, throughout this section we rely on the theoretical transactional memory model that is presented in Section 2.1.

Author's contribution. The contents of this section have been published in [FKKR14] and are a joint work. The author contributed to the algorithm design and proof of correctness of the algorithm presented in this section.

3.1.1 Overview and Main Ideas

Each transaction starts by announcing itself into an appropriate element of an announce array. This array has size n, with one entry for each process, used by the corresponding process to announce its transactions.

Update transactions execute speculatively and employ fine-grained locking to ensure consistency when updating t-variables. Specifically, each transaction T keeps track of the t-variables that it accesses by maintaining a read-set and a write-set. The read-set contains an entry for each t-variable that T reads, where the value read from the t-variable is stored. Similarly, for each t-variable that T writes, the write-set contains an associated entry which stores the value that T wants to write to the t-variable. At commit time, T attempts to obtain the locks that are associated with each t-variable in its read-set and its write-set.

In order to avoid deadlocks, the locks are acquired in ascending order based on the address of the t-variables. Once T acquires the lock for some t-variable x in its write-set, it maintains in the corresponding entry of its write-set, the value that x had at the time that T acquired the lock for it. Once T acquires all required locks, it enters its updating phase, where it actually updates the t-variables recorded in its write-set, and then enters its waiting phase, where it waits for active announced read-only transactions to commit. T finally releases all the acquired locks and commits. We remark that WFR-TM guarantees that if T enters its updating phase, then T will commit within a finite number of steps.

For each transaction T, WFR-TM maintains a record for it. The record for T contains T's status, a variable that represents the current state of T and can take the values simulating, updating, waiting, committed or aborted. Each transaction starts by speculatively executing its code during its simulating phase. An update transaction (that does not abort early) additionally executes an updating phase and a waiting phase. This last phase is needed to ensure wait-freedom for read-only transactions. The record for T also contains the read-set and write-set of T, as well as a set called *beforeMe* of active transactions that will be linearized before T. This set is needed in order to ensure consistency of reads.

For each t-variable x, WFR-TM maintains a record containing the current value of x, its version which is a strictly increasing sequential number, and a pointer *owner* to some transaction's record which indicates whether x is locked. An update transaction T_w acquires the lock for x each time it successfully executes a `CAS` to identify itself as the *owner* of x; x is considered to be unlocked if either the owner field of its record is null or the status of the transaction that it points to is aborted or committed. T_w releases all the locks it has acquired by successfully changing its status to either committed or aborted (i.e. in one atomic step).

WFR-TM provides wait-freedom for any read-only transaction T by ensuring that T_r reads consistent values independently of whether the transactional variables that it accesses are locked, as follows. When a t-variable x is unlocked, T_r reads its value from x's record. Suppose that x is locked by some update transaction T_w at some point. We define an old value and a new value for x at that point. The old value for x is the value stored in x's record at the moment that it was locked by T_w, whereas the new value for x is the value that T_w wants to write to x. Notice that the old value of x is contained it its record until T_w writes the new value for it during its updating phase. Afterwards, the old value is recorded in the write-set of T_w.

During its initialization, each transaction T takes a snapshot of the announce array, i.e. a consistent view of the announced transactions together with their statuses. We remark that taking this snapshot is easier than in the standard wait-free implementations of snapshot objects presented in the literature [AAD+93, And94, And93, AR93], since, in WFR-TM, update transactions are waiting for read-only transactions to commit. Using this snapshot, T decides whether it must read or ignore the values written by update transactions that are active during T's execution. Specifically, while T is taking the snapshot of the announce array, it adds into the $beforeMe$ set all those announced transactions whose status is either *waiting* or *committed.* If T reads from x and finds that it is locked by an update transaction T_w, then it checks if T_w is in T's $beforeMe$ set. If this is so, T reads directly from the record of x. Since T_w's status was *waiting* or *committed* when it was recorded by T, during T's initialization this value was the new value of T_w. If T_w is not in T's $beforeMe$ set, T ignores the value that T_w want to write on x and decides which value to read for x based on the status of T_w. If T_w is in its simulating phase, T returns the value found in x's record (and thus ignores the value that T_w wants to write since T_w has not yet started updating its t-variables). If T_w is in its updating phase, T reads the old value for x from T_w's write-set. This is necessary because in this case, T_w is in the process of updating the t-variables contained in its write-set, so some of them may contain the new values and some of them may still contain the old values. For instance, if the read-set of T contains two t-variables x and y updated by T_w, and T reads both of them from their records, it may read the old value for x and the new value for y, which would be inconsistent. The same action is taken by T if T_w is either in its waiting phase or it is committed, since similar consistency problems could appear if T has read other t-variables written by T_w while T_w was in earlier phases of its execution. In all these cases, if T is a read-only transaction, then during its commit time, T_w will wait for T to commit before committing itself. This procedure ensures consistency of the values read for the t-variables by read-only transactions.

Before committing, each update transaction reads all entries of the announce array and waits for the completion of each announced read-only transaction that it encounters. By incorporating

Algorithm 1 Data structures of WFR-TM.

```
typedef statval {SIMULATING, UPDATING, WAITING, COMMITTED, ABORTED}

type txrec
    uint pid
    statval status
    set of wnode elements wset
    set of pointers to
        txrec elements beforeMe

type tvarrec
    value val
    uint ver
    txrec *owner

type rnode
    tvarrec *tvar
    value val
    uint ver

type wnode
    tvarrec *tvar
    value oldval
    uint oldver
    value newval

// Shared variable
shared txrec *A[1..n]

// Persistent local variable for process p
set of rnode elements $rs_p$
```

this waiting mechanism, WFR-TM ensures that if a read-only transaction T_r ignores the value written to a t-variable by an update transaction T_w, then T_w does not commit before T_r has committed. This is necessary to argue that at the time that T_r commits, it will not have read an inconsistent set of values. It is also necessary for guaranteeing the progress properties of the algorithm.

For each t-variable x, there is a version associated to it whose value is unique for each value stored in x. An update transaction T_w performs its reads by executing the same actions described above for read-only transactions. Additionally, since the waiting mechanism is not employed between update transactions, in order to ensure opacity, T_w must validate its read-set whenever it reads a t-variable for the first time, as well as a final time before it starts its updating phase. Specifically, T_w validates the read-set by comparing the current version of each t-variable contained there in, against the version that T_w last read for this t-variable (which is contained in its read-set). T_w aborts if a mismatch is found for some t-variable. We remark that T_w performs the final validation in an indirect way by acquiring the lock for each t-variable contained in its read-set. If a version mismatch is found, the `CAS` used to acquire the lock for the corresponding t-variable, fails, and T_w aborts.

3.1.2 Algorithm Description

Data Structures. Algorithm 1 presents the data structures of WFR-TM. For each transaction T, WFR-TM stores a record of type `txrec` that contains: 1) the identifier *pid* of the process that initiated T, 2) a three-bit variable *status*, storing the status of T, 3) a set *wset* of elements of type `wnode`, implementing the write-set of T, and 4) a set $beforeMe$ of pointers to elements of type `txrec`. Also, each process p maintains a local set rs_p of elements of type `rnode`, implementing the read-set of each transaction it initiates.

For each t-variable x, WFR-TM stores a `CAS` object of type `tvarrec`, containing: i) the value *val* of x which we assume to be of type `value`, ii) the version number *ver* of x which is an unsigned integer, and iii) a pointer *owner* to a `txrec` record. To implement WFR-TM with single-word `CAS` objects, indirection can be used as in [HLMS03, TMG$^+$09].

We remark that an element of type `rnode`, maintained for a t-variable x, contains: i) a pointer *tvar* to the `tvarrec` record of x, ii) the value *val* of x read by T, and iii) an unsigned integer value *ver* representing the version number of x read by T. Moreover, an element of type `wnode`, maintained for a t-variable x, contains: i) a pointer *tvar* to the `tvarrec` record of x, ii) the (old) value *oldval* of x, iii) an unsigned integer *oldver* representing the (old) version number of x, and iv) the value *newval* that T will store into x.

Finally, A is the announce array maintained by WFR-TM. Initially, each entry of A points to a dummy `txrec` record whose *status* is equal to `COMMITTED` and *wset* is the empty set. Also, for each t-variable x, the fields of the `tvarrec` record of x have the following values: i) *val* contains an initial value, ii) *ver* is equal to 0, and iii) *owner* points to a dummy `txrec` record whose *status* field is equal to `COMMITTED`.

Pseudocode Description. The pseudocode of WFR-TM is provided in Algorithms 2 and 3. We remark that in the pseudocode, the commit and abort responses are modeled with the boolean values `true` and `false`, respectively. We continue to present detailed descriptions for the implementations of the transactional routines (as well for the routines that each of them calls).

BeginTx When called by process p for transaction T, it creates (line 23) and initializes (lines 24 - 28) the `txrec` record of T, and then announces T in $A[p]$ (line 29). Finally, it calls CheckIfPerformed to appropriately initialize the *beforeMe* set of T (line 30).

Each iteration of the while loop of CheckIfPerformed, reads all elements of A (lines 34 - 35) and adds to the *beforeMe* set of T (line 37) new update transactions (i.e. those that are not already in *beforeMe*) whose status is either waiting or committed (line 36). A new iteration will start if some transaction is added to *beforeMe* in the current iteration. This procedure guarantees that at the beginning of the last iteration of that execution of the `for` of line 34 that is executed during the last iteration of the `do while` of lines 33 to 38, *beforeMe* contains a consistent snapshot of the announced transactions that have entered their waiting phase (or are committed).

We now explain why CheckIfPerformed terminates within a finite number of steps. Any update transaction T_w that is announced after the announcement of T cannot commit before CheckIfPerformed completes. This is so because even if T_w reaches its commit phase, T_w will consider T as a read-only transaction (since T has an empty write-set as long as it executes CheckIfPerformed), so T_w will wait for T to either terminate or become an update transaction. This ensures that only a limited number of new update transactions can appear while CheckIfPerformed is executed, which in turn ensures that CheckIfPerformed returns in a finite number of steps.

CreateTvar When called by process p for transaction T, it creates, initializes (line 40), and returns (line 41) a new `tvarrec` record for the newly allocated t-variable.

Algorithm 2 Pseudocode for BeginTx, CheckIfPerformed, CreateTvar, ReadTvar, and Validate of WFR-TM.

```
txrec *BeginTx() by process $p$:
  txrec $*newTx$ = new txrec
  $newTx \rightarrow pid = p$
  $newTx \rightarrow status$ = SIMULATING
  $newTx \rightarrow wset$ = empty set of wnode elements
  $newTx \rightarrow beforeMe$ = empty set of pointers to txrec elements
  $rs_p$ = empty set of rnode elements
  $A[p] = newTx$                                // $T$ announces itself
  CheckIfPerformed($newTx$)                     // $T$ initializes its $beforeMe$ set
  return ($newTx$)

CheckIfPerformed(txrec *$newTx$) by process $p$:
  do
    for $i = 1$ up to $n$, excluding $p$, do
      $tran = A[i]$
      // check if $tran$ is an update transaction not in $newTx$'s $beforeMe$ set that has entered its waiting phase,
      if ($tran \notin newTx \rightarrow beforeMe$ AND $tran \rightarrow wset \neq \emptyset$ AND
          $tran \rightarrow status \in \{$WAITING, COMMITTED$\}$) then
        add $tran$ in $newTx \rightarrow beforeMe$
  while a new element is added in $newTx \rightarrow beforeMe$

tvarrec *CreateTvar(txrec *$tx$) by process $p$:
  tvarrec $newTvar$ = new tvarrec $\langle \perp, 0, tx \rangle$
  return ($newTvar$)

$\langle$boolean, value$\rangle$ *ReadTvar(txrec *$tx$, tvarrec *$tvar$) by process $p$:
  if an element $el$ with $el.tvar = tvar$ exists in $tx \rightarrow wset$ then
    return $\langle$true, $el.newval\rangle$
  if an element $el$ with $el.tvar == tvar$ exists in $rs_p$ then
    return $\langle$true, $el.val\rangle$
  $\langle val, ver, owner \rangle = *tvar$
  $status = owner \rightarrow status$
                    // if $tvar$ is locked by a transaction $T_w$ that is not to be linearized before $tx$ and $T_w$
                    // is in its updating or waiting phase, then read the old value of $tvar$ from $T_w$'s write-set
  if (an element $el$ with $el.tvar == tvar \in owner \rightarrow wset$ AND
      $owner \notin tx \rightarrow beforeMe$ AND $status \neq$ SIMULATING) then
    $\langle val, ver \rangle = \langle el.oldval, el.oldver \rangle$
  add $\langle tvar, val, ver \rangle$ in $rs_p$
  if ($tx \rightarrow wset \neq \emptyset$ AND Validate($tx$) = false) then  // call Validate to ensure opacity
    $tx \rightarrow status$ = ABORTED
    return $\langle$false, $\perp\rangle$
  return $\langle$true, $val\rangle$

boolean Validate(txrec *$tx$) by process $p$:
  for each element $el$ in $rs_p$
    $\langle val, ver, owner \rangle = *el.tvar$
    if ($ver \neq el.ver$) then return false
  return true
```

Algorithm 3 Pseudocode for WriteTvar, CommitTx, LockDataSet, and WaitReaders of WFR-TM.

```
boolean WriteTvar(txrec *$tx$, tvarrec *$tvar$, value $val$) by process $p$:
  if an element $el$ with $el.tvar == tvar$ exists in $tx \to wset$ then
    update $el.newval$ with $val$
  else add $\langle tvar, \bot, \bot, val \rangle$ in $tx \to wset$
  return true

boolean CommitTx(txrec *$tx$)by process $p$:
  if ($tx \to wset == null$) then                  // if $tx$ is read-only, commit
    $tx \to status$ = COMMITTED
    return true

  if (LockDataSet($tx$) == false) then   // if locking of some t-variable fails, abort
    $tx \to status$ = ABORTED
    return false

  $tx \to status$ = UPDATING                   // $tx$ enters updating phase
  for each element $el$ in $tx \to wset$ do
                  // u-cas: write here would also do; we use CAS to be coherent with our model
    CAS(*$el.tvar$,*$el.tvar$, $\langle el.newval, el.tvar \to ver + 1, tx \rangle$)

  $tx \to status$ = WAITING                    // $tx$ enters waiting phase
  WaitReaders($tx$)                            // $tx$ waits announced read-only transactions

  $tx \to status$ = COMMITTED                  // $tx$ commits
  return true

boolean LockDataSet(txrec *$tx$) by process $p$:
  for each element $el$ in $tx \to wset \cup rs_p$, in ascending order (based on $tvar$ field)
    if $\exists$ an element $el' \in rs_p$ with $el'.tvar == el.tvar$ then
                  // if $tx$ has read the $tvar$ before, use this old value for consistency
      $\langle val, ver, owner \rangle = \langle el'.val, el'.ver, el'.tvar \to owner \rangle$
                  // otherwise, if the $tvar$ was not read before, use the current value as old value
    else $\langle val, ver, owner \rangle = *(el.tvar)$

    if ($owner \to status \notin$ {COMMITTED, ABORTED})            // $el.tvar$ is locked
      if $\exists$ an element $el'' \in owner \to wset$ with $el''.tvar == el.tvar$ then
                        // if it is in the write-set of $owner$, locking fails
        return false
                          // otherwise, wait until it is unlocked
      else wait until $owner \to status \in$ {COMMITTED, ABORTED}

    if (CAS(*$el.tvar$, $\langle val, ver, owner \rangle$, $\langle val, ver, tx \rangle$) == false) then   // l-cas: try to lock $el.tvar$
      return false
                             // if $el$ is written by $tx$, then maintain the old value of $el.tvar$
    if ($el \in tx \to wset$) then update $\langle el.oldval, el.oldver \rangle$ with $\langle val, ver \rangle$

  return true

void WaitReaders(txrec *$tx$) by process $p$:
  for $i = 1$ up to $n$, excluding $p$, do
    $tran = A[i]$
    if ($tran \neq null$ AND $tran \to wset == null$) then
      wait until ($tran \to status$ == COMMITTED OR $tran \to wset \neq null$)
```

ReadTvar When called by T to read the value of some t-variable x, ReadTvar first checks if there is an entry for x in the write-set (lines 43 - 44) or in the read-set of T (lines 45 - 46). If this is the case, it returns the value from there (to ensure opacity). Otherwise, the value of x is determined on lines 47 - 50.

Initially, the value $\langle val, ver, owner\rangle$ of x's `tvarrec` record (line 47) and the status of x's *owner* (line 48) are read. If the status of x's *owner* is `SIMULATING`, then the value for x that T returns is *val*, as read on line 47. Otherwise, the first and third condition of line 49 evaluate to `true`. Recall that x has an old value and a new value which are stored in T_w's write-set entry for x (specifically, in fields *oldval* and *newval* of this entry, respectively). If T_w is contained in T's *beforeMe* set, i.e. the second condition of line 49 evaluates to `false`, then T_w's update on x has already been performed before the beginning of T. Therefore, again the value for x that T should read is *val*. However, if T_w is not contained in T's *beforeMe* set, then T should not read T_w's update on x, i.e. the new value of x, and should instead read the old value of x; this value is read on line 50.

After T determines the value to read for x, it adds it together with its corresponding version number in its read-set (line 51). In case T is an update transaction, then its read-set is validated by calling Validate (line 52); Validate (lines 57 - 60) returns `true` when no version number of the elements in T's read-set has changed; it returns `false` otherwise.

WriteTvar When called by T_w to update some t-variable x with value *val*, T_w first checks whether it has previously invoked WriteTvar to modify x. If this is so, then there is already an element for x in T_w's write-set (line 62) and WriteTvar updates the *newval* field of this element to *val* (line 63). Otherwise, a new `wnode` element for x is added in T_w's write-set (line 64).

Recall that when T_w enters its updating phase, the *oldval* and *oldver* fields of x's `wnode` must contain the value and version number, respectively, written by the transaction for which it holds that it had x in its write-set and was the last to commit before T_w's acquisition of the lock of x (or the initial values if such a transaction does not exist). WFR-TM allows another transaction T' to snoop into T_w's write-set (line 50) in order to read the old value of some t-variable contained there. Therefore, T_w's write-set must offer a way to T' to read values that are mutually consistent. To achieve this, WriteTvar sets the *oldval* and *oldver* fields of new `wnode` elements that are added in a write-set to be equal to $\perp$ (line 64). This is necessary for avoiding bad scenarios such as the following: In addition to x, assume that T_w wants to write another t-variable y and let C be a configuration at which T_w has called WriteTvar for x but not yet for y. Thus, T_w has created a write set entry for x, but there is no such entry in T_w's write-set for y. To see what might go wrong, assume that T_w has also read (before C) the contents of x's `tvarrec` and stored them in the *oldval* and *oldver* fields of x's `wnode`. Now, let another transaction T'' lock and update both x and y, and commit. Then, T_w continues by invoking WriteTvar for y. So, it places an entry in its write-set for y and reads the contents of y's `tvarrec` to store in the *oldval* and *oldver* fields of this entry. Then, T_w acquires the locks of both x and y. If T' snoops both x and y from T_w's write-set, it will read inconsistent values.

CommitTx If T is a read-only transaction (i.e. its write-set is empty), CommitTx changes T's status to committed and returns `true` (lines 67 - 69). If T is an update transaction, it attempts to acquire the required locks by calling LockDataSet (line 70), which is described in the next paragraphs. If it fails to acquire some lock, LockDataSet returns `false` and T is aborted (lines 70 - 72). Otherwise, all the required locks have been acquired and LockDataSet returns `true`. Then, T enters its updating phase (lines 73 - 75) and updates the t-variables in its write-set (line 75). Notice that it also increments the version number of each t-variable by one. Afterwards, T enters its waiting phase (line 76) and waits until all announced read-only transactions commit. This is done by calling WaitReaders (line 77). WaitReaders goes through the announce array A, and waits until each active read-only transaction (line 96) either commits or turns out to be an update transaction (line 97).

LockDataSet is called by T to lock each t-variable in its read-set and write-set. Deadlocks are avoided by acquiring the locks in (ascending) order (based on the *tvar* pointer contained in each `rnode` or `wnode` element). Initially, LockDataSet determines the value and version number of each t-variable x that it wants to lock, as follows: If x exists in T's read-set, these values are taken from the corresponding read-set entry (line 83). Otherwise, they are read from x's `tvarrec` record (line 84).

LockDataSet tries to lock x using a `CAS` primitive which stores a pointer to T's `txrec` record into the *owner* field of x's `tvarrec` record (line 90). Notice that this `CAS` also serves as a final validation of the value of x read by T (in case x is in T's read-set). LockDataSet returns `true` only if it successfully locks all the t-variables in T's read-set and write-set (line 92). If x is already locked by some transaction T' (lines 85 to 86), LockDataSet by T returns `false`. If x is locked by some transaction that does not intend to update it, LockDataSet waits until this transaction completes (line 88). Finally, recall that when LockDataSet is invoked, the contents of the *oldval* and *oldver* fields of x's element in T's write-set are $\perp$. In case x is locked, these fields are updated with the determined current values for x (line 91), so that if T enters its updating phase these fields are appropriately set in each element of T's write-set.

3.1.3 Proof of Correctness

In this section, we prove that WFR-TM is opaque. We also study the progress properties of WFR-TM. In Section 3.1.3, we provide some preliminaries including useful notation. In Section 3.1.3, we argue about the correctness of read-only transactions, and in Section 3.1.3 we prove correctness for update transactions. The progress properties of WFR-TM are studied in Section 3.1.4.

Preliminaries Consider any execution α of WFR-TM and let T be any transaction in α. The execution interval of T is denoted by α_T. The process p that initiates T is its initiator. We denote by CE_T the last configuration of α_T (if it exists). We say that T announces itself when it executes the write to $A[p]$ on line 29.

By inspection of the code of WriteTvar (lines 62 - 64), T adds a unique record for each t-variable that it writes in its write-set. Moreover, by inspection of the code of ReadTvar (lines 43 - 55), for each t-variable x read by T, T executes lines 47 - 55 during the first instance of ReadTvar for x executed by T; we denote by $RT_{x,T}$ this instance. We remark that each subsequent instance of ReadTvar executed by T for x returns either on line 44 or on line 46. So, by inspection of the code, T maintains a unique record for each t-variable it reads in its read-set.

Observation 1. *Consider any transaction T in an execution α and let C be any configuration. Then,*

1. *if T has executed at least one instance of* WriteTvar *for some t-variable x by C, there is a unique record for x in T's write set at C;*

2. *if T has executed $RT_{x,T}$ for some t-variable x by C, there is a unique record for x in T's read set at C; any instance of* ReadTvar *for x by T following $RT_{x,T}$ does not execute lines 47 - 55.*

Each time T successfully executes the `CAS` primitive of line 89 for some t-variable x, we say that T becomes the owner of x or acquires the lock for x. We call the `CAS` primitive of line 89, l-cas. Since LockDataSet is executed at most once (line 70) by T, by inspection of the code (lines 81 - 90) it follows that at most one l-cas is executed for each t-variable in the data-set of T. Assume that T acquires the lock for x. We denote by $CL_{x,T}$ the configuration after the successful execution of the l-cas for x by T. Each time T executes the `CAS` primitive of line 75 for some t-variable x with values $\langle v, d\rangle$, we say that T updates the value and the version number of x with v and d, respectively, or writes the value v and version number d for x. We call the `CAS` of line 75, u-cas. Notice that this `CAS` is always successful and thus it could be replaced by a simple `write`. However, that would result in a version of WFR-TM which uses objects that support all three primitives `read`, `write`, and `CAS`.

By inspection of the code (line 23), each transaction is associated with a unique `txrec` record. Recall that the status of T is the value of the field *status* in this record. For simplicity, throughout this proof we abuse notation and we use the same notation to refer to the name of some transaction and to its `txrec` record.

By inspection of the code (line 25), $T.status$ is initially `SIMULATING`. Notice that no transaction other than T can update T's status. If T is read-only, by inspection of the code (lines 52, 55, and 67 - 69), it follows that its status can only change from `SIMULATING` to `COMMITTED` (line 68). If T is an update transaction, then by inspection of the code (lines 52 - 54 and 71 - 72), its status may change from `SIMULATING` to `ABORTED`. Also, by inspection of the code (lines 73, 76, and 78), its status may change from `SIMULATING` to `UPDATING`, from `UPDATING` to `WAITING`, and from `WAITING` to `COMMITTED`. As long as its status is `SIMULATING`, `UPDATING`, or `WAITING`, we say that T is in its simulating, updating, or waiting phase, respectively.

Observation 2. *The following hold for each transaction T and each configuration C in α:*

1. *if T is a read-only transaction and T's status is* `SIMULATING` *at C, T's status can only change to* `COMMITTED` *after C;*

2. *if T is an update transaction and T's status is* `SIMULATING` *at C, T's status can change from* `SIMULATING` *either to* `ABORTED` *or to* `UPDATING` *after C;*

3. *if T's status is* `ABORTED` *at C, T does not execute lines 73 - 79 of* CommitTx *after C;*

4. *if T's status is* `UPDATING` *at C, T's status can change from* `UPDATING` *to* `WAITING` *after C;*

5. *if T's status is* `WAITING` *at C, T's status can only change to* `COMMITTED` *after C.*

If the status of T becomes `COMMITTED` or `ABORTED`, then it never changes again. Recall that in this case we say that T completes (commits or aborts, respectively). Notice that a committed transaction returns `true`, whereas an aborted transaction returns `false`. If T commits in α, we denote by CM_T the configuration after the execution of line 68 or line 78 which changes T's status to `COMMITTED`. If T aborts in α, we denote by CA_T the configuration after the execution of line 53 or line 71 that changes the status of T to `ABORTED`. Notice that if T completes, then either $CE_T = CM_T$ or $CE_T = CA_T$, depending on whether T commits or aborts, respectively.

Consider any update transaction T_w. If T_w enters its waiting phase in α, we denote by CU_{T_w} and CW_{T_w} the configurations after the execution of lines 73 and 76, respectively, which change T_w's status to `UPDATING` and `WAITING`, respectively. By inspection of the code (lines 70, 73, and 76), T_w calls LockDataSet before CU_{T_w} and this call returns `true` (i.e. it is successful). Thus, by inspection of the code (lines 51, 64, 81, 89, and 92), T_w has acquired the locks for all t-variables accessed by T_w before CU_{T_w}.

If T_w acquires the lock for some t-variable x, by inspection of the code (lines 70-79), it follows that at CL_{x,T_w} the status of T_w is equal to `SIMULATING`. We say that T_w maintains the lock for x, or x is locked by T_w, in each configuration following CL_{x,T_w} (including it) in which the status of T_w is neither `COMMITTED` nor `ABORTED`. The change of the status of T_w to `COMMITTED` or `ABORTED`, indicates that T_w releases all locks it has acquired. We denote by α_{x,T_w} the execution interval of α_{T_w} during which T_w maintains the lock for x. We remark that α_{x,T_w} starts with CL_{x,T_w} and, in case T_w completes in α, it ends with the configuration preceding CM_{T_w} or CA_{T_w} (depending on whether T_w commits or aborts, respectively). If T_w does not complete in α, α_{x,T_w} is the suffix of α, starting at CL_{x,T_w}. Table 3.1 briefly summarizes the notation introduced thus far, as well as some notation that will be introduced later. Note that notation that refers to some configuration starts with the letter C.

By inspection of the code (lines 70 - 76) and by the definition of α_{x,T_w}, we derive the following observation.

Observation 3. *Consider any update transaction T_w. Then,*

1. *T_w has acquired the locks for all t-variables accessed by T_w before CU_{T_w};*

2. *for each t-variable x accessed by T_w,*

α_T	the execution interval of T
$RT_{x,T}$	the (first and) unique instance of ReadTvar for x by T during which T executes lines 47 - 55 for x
CE_T	the last configuration of α_T
$CL_{x,T}$	the configuration after the successful execution of the l-cas for x by T (line 89)
CU_T	the configuration after the execution of line 73 that changes the status of T to `UPDATING`
CW_T	the configuration after the execution of line 76 that changes the status of T to `WAITING`
CM_T	the configuration after the execution of line 78, that changes the status of T to `COMMITTED`
CA_T	the configuration after the execution of line 53 or line 71 that changes the status of T to `ABORTED`.
$\alpha_{x,T}$	the execution interval of α_T during which T maintains the lock for x
CR_T	the configuration at the beginning of the last execution of the `for` of line 34 in CheckIfPerformed by T
$RS_T(C)$	the set containing each triple $\langle x, v, d\rangle$ added to the set rs_p (of the process p executing T) from the beginning of the execution of T until configuration C
RS_T	$RS_T(CE_T)$
ℓ_C	the sequence of transactions of α that have been serialized before or at C
$\mathcal{T}'_x$	the sequence of update transactions (in order) that acquire the lock for a fixed t-variable x in α
$\mathcal{T}_x$	the subsequence of $\mathcal{T}'_x$ containing those transactions that update t-variable x

Table 3.1: Notation used during the proof of WFR-TM.

- *at CL_{x,T_w}, the status of T_w is equal to* `SIMULATING`*;*
- *CU_{T_w} and CW_{T_w} occur in α_{x,T_w};*

3. *$CU_{T_w} < CW_{T_W}$;*

4. *for each t-variable x updated by T_w, T_w updates x during α_{x,T_w}, after CU_{T_w} and before CW_{T_w}.*

We continue to prove that, during α_{x,T_w}, T_w is the owner of x.

Lemma 4. *Consider any update transaction T_w that acquires the lock for some t-variable x. During α_{x,T_w}, the owner field of the* `tvarrec` *record of x contains a pointer to the* `txrec` *record of T_w.*

Proof. By inspection of the code (line 89) and by the definition of CL_{x,T_w}, the claim holds at CL_{x,T_w}. Assume, by the way of contradiction, that there is some configuration in α_{x,T_w} in which the *owner* field of the `tvarrec` record of x contains a pointer to the `txrec` record of a transaction $T'_w \neq T_w$. Let C be the first such configuration. By inspection of the code, it follows that T'_w acquires the lock for x at the step executed before C. Let l_{CAS} be the successful l-cas that T'_w executed in order to acquire the lock for x. Before executing l_{CAS}, T'_w reads the value

$\langle -, -, owner \rangle$ either on line 83 or on line 84; let r_x be this read. Notice that r_x is executed before the end of α_{x,T_w}.

To derive a contradiction, we consider the following cases. Assume first that r_x reads a pointer to the `txrec` record of T_w. By inspection of the code (line 89), the *owner* field of the `tvarrec` of x changes only when a transaction T executes a successful l-cas for x and it is only T that may write a pointer to its `txrec` in this field. Thus, it follows that r_x is performed after CL_{x,T_w}. By definition of α_{x,T_w}, $T_w.status \notin \{\texttt{COMMITTED}, \texttt{ABORTED}\}$ during α_{x,T_w}. So, by inspection of the code (lines 85 - 86), the instance of LockDataSet executed by T'_w returns `false`. Then, by inspection of the code (lines 70 - 72), T'_w aborts, so it does not attempt to lock x. This contradicts the assumption that T'_w has acquired the lock for x at C.

Assume now that r_x returns $owner = T''_w$ with $T''_w \neq T_w$. By inspection of the code (lines 83, 84, and 89), l_{CAS} can only succeed if the *owner* field of the `tvarrec` record of x contains a pointer to the `txrec` record of T''_w. However, since l_{CAS} is the first successful `CAS` for x executed after CL_{x,T_w}, the *owner* field of the `tvarrec` of x contains a pointer to the `txrec` of T_w when l_{CAS} is executed (and not to T''_w). It follows that l_{CAS} does not succeed. This contradicts the definition of l_{CAS}. □

Fix any t-variable x. Let $\mathcal{T}'_x = T'_0, T'_1, T'_2, \ldots$ be the sequence of update transactions (in order) that acquire the lock for x in α; let $T'_0 = T_0$ be the dummy `txrec` to which the *owner* field of the `tvarrec` of x initially points. We remark that some transactions in $\mathcal{T}'_x$ may not invoke WriteTvar for x although they access x by invoking ReadTvar for it. Notice also that some transactions in $\mathcal{T}'_x$ may abort.

Lemma 5. *For each integer $d > 1$, the following hold:*

1. *for each configuration C between $CL_{x,T'_{d-1}}$ (inclusive) and CL_{x,T'_d} (exclusive), the owner field of the* `tvarrec` *of x is equal to T'_{d-1} at C;*

2. $\alpha_{x,T'_{d-1}} < \alpha_{x,T'_d}$.

Proof. Fix any integer $d > 1$ and let C be any configuration between $CL_{x,T'_{d-1}}$ (inclusive) and CL_{x,T'_d} (exclusive). By inspection of the code (line 75, line 89), the *owner* field of the `tvarrec` of x changes only when a transaction T executes a successful l-cas for x and this l-cas writes a pointer to T in the *owner* field of the `tvarrec` of x. Thus, the l-cas by T'_{d-1} and T'_d write a pointer to T'_d and a pointer to T'_{d-1}, respectively, in the *owner* field of the `tvarrec` of x. By the definition of $\mathcal{T}'_x$, no other successful l-cas for x is executed between the l-cas by T'_{d-1} and the l-cas by T'_d. Since C is a configuration between $CL_{x,T'_{d-1}}$ and CL_{x,T'_d}, by the definitions of $CL_{x,T'_{d-1}}$ and CL_{x,T'_d}, it follows that the *owner* field of the `tvarrec` of x is equal to T'_{d-1} at C. So, claim 1 follows.

Claim 2 immediately follows by Lemma 4 and the definition of $\mathcal{T}'_x$. □

Let $\mathcal{T}_x = T_1, T_2, \ldots$ be the subsequence of $\mathcal{T}'_x$ containing those transactions that update t-variable x in α.

Lemma 6. *For each integer $d > 0$, the following hold:*

1. *the u-cas for x executed by T_d changes the ver field of the* `tvarrec` *of x from the value $d-1$ to the value d; at each configuration between the u-cas of T_{d-1} (or from the beginning of the execution, if $d = 1$) and the u-cas of T_d, the ver field of the* `tvarrec` *of x has the value $d - 1$;*

2. *T_d has a* `wnode` *element for x in its write set with value $d - 1$ stored in its oldver field;*

3. *each transaction T between T_{d-1} and T_d in $\mathcal{T}'_x$ that invokes* WriteTvar *for x, has a* `wnode` *element for x in its write set with value $d - 1$ stored in its oldver field.*

Proof. The proof is by induction on d. Fix any $d > 0$ and assume that the claim holds for $d-1$. We prove that the claim holds for d.

Since T_d updates x, Observation 3 (claims 1 and 4) implies that T_d acquires the lock for x by successfully executing an l-cas for x (line 89) before CU_{T_d}; moreover, T_d updates x during α_{x,T_d}. By Lemma 5 (claim 2), α_{x,T_d} and $\alpha_{x,T_{d'}}$, $d' \neq d$, do not overlap.

Assume that T is either T_d or any transaction between T_{d-1} and T_d in $\mathcal{T}'_x$ that invokes WriteTvar for x. By definition of $\mathcal{T}'_x$, T successfully executes an l-cas for x. Moreover, T has invoked WriteTvar for x and, therefore, Observation 1 implies that T has added x in its write set.

Assume first that $d = 1$. By inspection of the code, it follows that the value and the version number of x change only when a successful u-cas for x (line 75) is executed, i.e. when a transaction updates x. Thus, up until the time that T_1 successfully executes its u-cas for x, the *ver* field of the `tvarrec` of x has its initial value (i.e. it has the value 0). Since T precedes T_1 in $\mathcal{T}'_x$, Lemma 5 (claim 2) implies that when T successfully executes its l-cas for x, the *ver* field of the `tvarrec` of x has the value 0.

Assume now that $d > 1$. By the induction hypothesis (claim 1), T_{d-1} executes the `CAS` of line 75 for x and this `CAS` changes the *ver* field of the `tvarrec` of x to the value $d - 1$. By Observation 3 (claim 4), the update of x by T_{d-1} occurs during $\alpha_{x,T_{d-1}}$. By definition of $\mathcal{T}_x$, it follows that T_d is the first transaction to successfully execute a u-cas for x after the successful u-cas for x executed by T_{d-1}. Since T is between T_{d-1} and T_d in $\mathcal{T}'_x$, Lemma 5 (claim 2) implies that, when T successfully executes its l-cas for x, the *ver* field of the `tvarrec` of x has the value written there by T_{d-1}. By induction hypothesis (claim 1), this value is $d-1$; moreover, up until the time that T_d executes its u-cas for x, the *ver* field of the `tvarrec` of x has the value $d - 1$.

In either case, when the successful l-cas of T is executed, the value of the *ver* field of the `tvarrec` of x is equal to $d - 1$. Moreover, when T_d executes the u-cas for x, the *ver* field of the `tvarrec` of x has the value $d-1$. By inspection of the code (line 75), it follows that T_d changes the version number of x from $d - 1$ to d. Since the version number of x changes only when a successful u-cas for x is executed, by the definition of $\mathcal{T}_x$, it follows that claim 1 holds.

Since the value of the *ver* field of the `tvarrec` of x is equal to $d - 1$ when T successfully executes the l-cas for x, by inspection of the code (line 89), it follows that T uses $\langle -, d-1, -\rangle$ as the old value for its l-cas. Since T executes the l-cas for x successfully, T also executes line 91. Recall that T has added x in its write-set. Thus, the condition of the `if` statement of line 91 evaluates to `true`. By inspection of the code (line 91), it follows that T stores the value $d-1$ in the *oldver* field in the `wnode` for x in its write set. So, claims 2 and 3 hold. □

We continue to assign a point, called serialization point, to every read-only transaction that commits in α and to every update transaction that enters its waiting phase in α.

Consider any transaction T in α. Let CR_T be the configuration at the beginning of the last execution of the `for` of line 34 in CheckIfPerformed by T. Notice that CR_T is the configuration where the first iteration of the for of line 34 starts executing during the execution of last iteration of the `do while` of lines 33 - 38. If T is a read-only transaction that commits in α, we place its serialization point at CR_T. If T is an update transaction that enters its waiting phase in α, we place its serialization point at CW_T. By the way serialization points are assigned, the serialization point of each transaction is placed in its execution interval.

Lemma 7. *For each transaction T that is assigned a serialization point in α, the serialization point of T is placed in its execution interval.*

By the way serialization points are assigned, at each configuration C, there is a sequence of transactions of α that have been serialized before or at C. Let ℓ_C denote this sequence.

Consider any transaction T in α, let p be the process executing T, and let C be any configuration. Let $RS_T(C)$ be the set containing each triple $\langle x, v, d\rangle$ that has been added into rs_p from the beginning of the execution of T until C. If T completes, let $RS_T = RS_T(CE_T)$. Consider any triple $\langle x, -, d\rangle \in RS_T(C)$. We say that d is consistent at C, if it is the version number written by the last transaction in ℓ_C that updates x. $RS_T(C)$ is consistent at C, if for each triple $\langle x, -, d\rangle \in RS_T(C)$ the version number d of x is consistent at C. RS_T is consistent at C, if for each triple $\langle x, -, d\rangle \in RS_T$ the version number d of x is consistent at C.

Consider a transaction T that adds a triple with version number d for some t-variable x in its read-set during $RT_{x,T}$. Lemma 6 implies that T_d and T_{d+1} are the update transactions that write version numbers d and $d+1$, respectively, for x. The next lemma proves that during $RT_{x,T}$, T reads on line 47, as the owner for x, either T_d, or T_{d+1}, or any transaction between T_d and T_{d+1} in $\mathcal{T}'_x$ that invokes WriteTvar for x; moreover, if it reads T_d, then T has included T_d in its $beforeMe$ set.

Lemma 8. *Let T be any transaction and let C be a configuration such that $\langle x, -, d\rangle \in RS_T(C)$. Let r and r' be the reads of line 47 and line 48, respectively, executed by T in $RT_{x,T}$ and let T_w be the value returned by r for $x \rightarrow owner$. Then, either $T_w = T_d$ and $T_d \in T \rightarrow beforeMe$, or $T_w = T_{d+1}$, or T_w is any transaction between T_d and T_{d+1} in $\mathcal{T}'_x$ that invokes* WriteTvar *for x.*

Proof. Since $\langle x, -, d\rangle \in RS_T(C)$, Observation 1 implies that T adds $\langle x, -, d\rangle$ in its read-set during $RT_{x,T}$. By inspection of the code (lines 47, 50, and 51), T reads d during $RT_{x,T}$ either on line 47 or on line 50.

Let B be the set of transactions that are between T_d and T_{d+1} in $\mathcal{T}'_x$ and invoke WRITETVAR for x, and let $A = B \cup \{T_{d+1}\}$. We first argue that if T reads d on line 50, then $T_w \in A$. This is so since then, by inspection of the code (lines 47 and 49), T reads x's version number in the *oldver* field of some element e for x in the write-set of T_w. Thus, Lemma 6 (claims 2 and 3) implies that $T_w = T_{d+1}$, or T_w is any transaction between T_d and T_{d+1} in $\mathcal{T}'_x$ that invokes WRITETVAR for x. Thus, $T_w \in A$.

Notice that if T reads d on line 47, then Lemma 6 (claim 1) implies that when r is performed, T_d has successfully executed the u-cas for x, whereas T_{d+1} has not.

To obtain a contradiction, assume that either $T_w \notin A$, or $T_w = T_d$ and $T_d \notin T \rightarrow beforeMe$. Assume first that $T_w \notin A$. Then, it follows that T does not read d on line 50. Thus, T reads d on line 47. Recall that r occurs between the execution of the u-cas for x by T_d and the u-cas for x by T_{d+1}. By Observation 3 (claim 4), the u-cas primitives for x by T_d and by T_{d+1} are performed within α_{x,T_d} and $\alpha_{x,T_{d+1}}$, respectively. Since the *owner* field of x can only change when a transaction executes a successful l-cas for x, the definition of $\mathcal{T}'_x$ implies that r reads, as the owner for x, some transaction in A. This contradicts the assumption that $T_w \notin A$.

Assume now that $T_w = T_d$ and $T_d \notin T \rightarrow beforeMe$. Since $T_w \notin A$, it follows that T does not read d on line 50. Thus, T reads d on line 47.

Notice that the value for the status of T_d returned by r' cannot be `ABORTED` since T_d enters its updating phase. Since T_d updates x and acquires the lock for x, Observation 1 implies that T_d adds an element for x in its write-set. Recall that r occurs between the execution of the u-cas for x by T_d and the u-cas for x by T_{d+1}. Since r' follows r, it follows that r' is performed after the execution of the u-cas for x by T_d. Thus, Observation 3 (claim 4) implies that r' occurs after CU_{T_d} and therefore it must return a value other than `SIMULATING` for the status of T_d.

Since r' occurs after CU_{T_d}, by definition of α_{x,T_d}, it follows that r' occurs after CL_{x,T_d}. Observation 1 (claim 1) implies that an element e with $e.tvar = x$ exists in the write set of T_d when r' occurs. So, during the execution of $RT_{x,T}$, the first condition of the `if` statement of line 49 evaluates to `true`. Since, by assumption, $T_d \notin T \rightarrow beforeMe$, and r' returns a value other than `SIMULATING`, it follows that all the conditions of the `if` statement of line 49 evaluate to `true`. Thus, T executes line 50 to read d. This is a contradiction. □

Lemma 9. *Consider any transaction T and let C be a configuration such that $\langle x, -, d\rangle \in RS_T(C)$. Then, it holds that T_d enters its waiting phase in α and $CW_{T_d} < C$.*

Proof. Let p be the process that executes T_d. During the execution of CHECKIFPERFORMED by T, T (possibly repeatedly) reads, on line 35, the transaction that is announced in $A[p]$ and, on line 36, the status of this transaction. Let r_1 and r_2 be these two reads, as performed by T during the execution of the last iteration of the `do while` loop of lines 33 - 38. Moreover, during the execution of $RT_{x,T}$, T reads the `tvarrec` for x (line 47) and the status (line 48) of the transaction that it read as the owner of x on line 47. Let r_3 and r_4 be these reads.

To obtain a contradiction, suppose that either T_d does not enter its waiting phase or $CW_{T_d} > C$; let C' be either the configuration following the last step taken by T_d in α, or

CW_{T_d}, respectively. We first argue that $T_d \notin T \rightarrow beforeMe$. T reads d for x by executing either line 47 or line 50 during $RT_{x,T}$. If T executes line 50, let r_5 be this read. Notice that by inspection of the code, $r_1 < r_2 < r_3 < r_4 < r_5 < C$, and by assumption, $C < C'$. Thus, the definitions of r_1 and r_2 imply that in the instance of its CheckIfPerformed, either T does not read T_d in $A[p]$ whenever it executes line 35, or if it reads T_d in $A[p]$, it does not read a value equal to `WAITING` or `COMMITTED` for the status of T_d on line 36. Therefore, by inspection of the code (lines 35-37), $T_d \notin T \rightarrow beforeMe$.

Since $\langle x, -, d\rangle \in RS_T(C)$ and $T_d \notin T \rightarrow beforeMe$, Lemma 8 implies that r_3 returns a transaction T' that is either T_{d+1}, or a transaction between T_d and T_{d+1} in $\mathcal{T}'_x$ which invokes WriteTvar for x. Lemma 5 (claim 2) implies that $\alpha_{x,T_d} < \alpha_{x,T'}$. Observation 3 implies that if CW_{T_d} occurs, then it occurs in α_{x,T_d}. Since $r_3 < C'$ and the *owner* field of the `tvarrec` of x changes only when a successful l-cas for x is executed, by the definition of $\mathcal{T}'_x$, it follows that r_3 cannot return T'. This is a contradiction. □

Correctness of read-only transactions. Consider any execution α of WFR-TM. Throughout this section, we consider a read-only transaction T_r that commits in α.

Consider any update transaction T_w that enters its waiting phase in α. Then, by inspection of the code (lines 76 and 77), it follows that if T_w calls WaitReaders, it does so after CW_{T_w}. By inspection of the code (lines 29, 76 - 77, and 94 - 97), if T_r performs its announcement before CW_{T_w}, T_w will wait (line 97) for T_r to commit. Therefore, in this case, T_r commits before the completion of T_w.

Lemma 10. *Consider any update transaction T_w that enters its waiting phase in α. If T_r performs its announcement before CW_{T_w}, then T_r commits before the completion of the waiting phase of T_w in α.*

Assume that T_r reads version number d for t-variable x. Lemma 6 (claim 1) implies that the update transaction that writes the version number d for x is T_d. Lemma 9 implies that T_d enters its waiting phase in α, so T_d is assigned a serialization point in α which is placed at CW_{T_d}. The next lemma shows that the serialization point of T_d is placed before the serialization point of T_r.

Lemma 11. *Consider any triple $\langle x, -, d\rangle \in RS_{T_r}$. Then, $CW_{T_d} < CR_{T_r}$.*

Proof. To obtain a contradiction, suppose that $CW_{T_d} > CR_{T_r}$. Let r and r' be the reads on lines 47 and 48, respectively, executed during $RT_{x,T}$. Let T_w be the transaction returned by r as the owner of x. Lemma 8 implies that either $T_w = T_d$ and $T_d \in T_r \rightarrow beforeMe$, or $T_w = T_{d+1}$, or T_w is a transaction between T_d and T_{d+1} in $\mathcal{T}'_x$ which invokes WriteTvar for x.

Assume first that $T_w = T_d$ and $T_d \in T_r \rightarrow beforeMe$. By inspection of the code (lines 36 and 37), T_d can be added in the *beforeMe* set of T_r only after CW_{T_d}. Since $CR_{T_r} < CW_{T_d}$, this addition occurs after CR_{T_r}. By inspection of the code (line 38), it follows that an iteration of the `do-while` loop of lines 35 to 37 is initiated after CR_{T_r}. This is a contradiction to the definition of CR_{T_r}.

We next assume that $T_w = T_d + 1$ or T_w is any transaction between T_d and T_{d+1} in $\mathcal{T}'_x$ which invokes WriteTvar for x. Since T_r reads version number d for x, Observation 3 (claim 4) and Lemma 6 (claim 1) implies that $r > CL_{x,T_d}$. Since we have assumed that $CW_{T_d} > CR_{T_r}$, Lemma 10 implies that T_r commits before T_d completes its waiting phase. So, r occurs in α_{x,T_d}. Lemma 5 (claim 2) implies that $\alpha_{x,T_d} < \alpha_{x,T_w}$. Since r occurs in α_{x,T_d}, Lemma 6 (claim 1) implies that r cannot return T_w as the owner for x. This is a contradiction. □

We are now ready to prove that the read-set of every read-only transaction that commits is consistent.

Lemma 12. *RS_{T_r} is consistent at CR_{T_r}.*

Proof. Consider any triple $\langle x, -, d\rangle \in RS_{T_r}$. We prove that d is written by the last committed transaction that updates x and is serialized before CR_{T_r}. By Lemma 6, there is a unique update transaction T_d that writes d into x. This is done when T_d successfully executes the u-cas for x. Let C_d be the configuration following this u-cas. By inspection of the pseudocode, $C_d < CW_{T_d}$. By Lemma 11, it follows that $CW_{T_d} < CR_{T_r}$.

Assume, by the way of contradiction, that the last committed transaction that updates x and is serialized before CR_{T_r} is a transaction T_w which writes the value $d' \neq d$ for x. Let p be the process that executes T_w, and let C_w be the configuration following the successful u-cas that T_w executes to write d' as the version number of x. Since T_w is serialized at CW_{T_w}, $CW_{T_d} < CR_{T_r}$, and T_w is the last committed transaction that updates x and is serialized before CR_{T_r}, it follows that $CW_{T_d} < CW_{T_w}$. By Observation 3, CW_{T_d} is in α_{x,T_d} and CW_{T_w} is in α_{x,T_w}. Thus, Lemma 5 (claim 2) implies that $\alpha_{x,T_d} < \alpha_{x,T_w}$. By Lemma 6 (claim 1), it follows that $d' > d$. By Observation 3 (claim 4), C_d occurs in α_{x,T_d} and C_w occurs in α_{x,T_w}. Thus, $C_d < CW_{T_d} < C_w < CW_{T_w} < CR_{T_r}$.

Notice that after CR_{T_r}, T_r reads, on line 35, the transaction that is announced in $A[p]$ and then, on line 36, the status of this transaction. Let r_1 and r_2 be these two reads. Moreover, during RT_{x,T_r}, T_r reads, on line 47, the `tvarrec` for x, and, on line 48, the status of the transaction that it read as the owner of x on line 47. Let r_3 and r_4 be these reads.

In the rest of the proof, we first argue that r_3 does not return d for the version number of x. Thus, T_r must read d in the *oldver* field of some transaction by executing line 50. We denote by r_5 this read. We next argue that r_5 reads from the write-set of T_w and that the read of line 50 occurs only if $T_w \notin T_r \rightarrow beforeMe$. We also argue that r_1 reads T_w in $A[p]$ and r_2 reads `WAITING` for the status of T_w. We then derive a contradiction by proving that T_r adds T_w in $T_r \rightarrow beforeMe$.

We start by proving that r_3 does not return d for the version number of x. By inspection of the code (lines 75-77), T_w has updated the version number of x to d' before CW_{T_w}. Since $r_3 > r_1 > CW_{T_w}$, Lemma 5 (claim 2) and Lemma 6 (claim 1) imply that r_3 returns either d', or a value larger than d' for the version number of x. Thus, d is not read by T_r on line 47. So, by inspection of the pseudocode, d must be read by T_r on line 50, through the *oldver* field of the element maintained for x in the write-set of the owner of x at that point in time.

Since $CR_{T_r} > CW_{T_w}$, Lemma 5 (claim 1) implies that r_3 returns as the owner of x a transaction T'_w, which is either T_w or some other transaction that acquired the lock for x after T_w. We argue that $T'_w = T_w$ and $T_w \notin T_r \rightarrow beforeMe$. Since T_w writes $d' > d$, Lemma 5 (claim 2) and Lemma 6 (claims 2 and 3) imply that among the transactions that acquire the lock after T_w, those that invoke WriteTvar for x have a value larger than d stored in the *oldver* field of the `wnode` for x in their write-sets. It follows that it must be T_w that has the value d in the *oldver* field of the `wnode` for x in its write-set, and that T_w writes $d + 1$. Thus, r_5 returns T_w as the owner for x. Since T_r executes line 50, by inspection of the code, it follows that, in RT_{x,T_r}, the condition of the `if` statement of line 49 is evaluated to `true`. Therefore, $T_w \notin T_r \rightarrow beforeMe$; moreover, r_4 returns a value other than `SIMULATING` for the status of T_w. Since r_4 occurs after CR_{T_r} and therefore, after CW_{T_w}, it follows that r_4 returns either `WAITING` or `COMMITTED` for the status of T_w.

Since T_w is announced before CW_{T_w} (lines 29 and 76), $CW_{T_w} < CR_{T_r} < r_1 < r_3 < r_4$, and r_4 returns either `WAITING` or `COMMITTED` for the status of T_w, it follows that r_1 returns T_w as the owner of x and r_2 returns either `WAITING` or `COMMITTED` for the status of T_w. So, by inspection of the code (lines 36 - 37), it follows that T_r evaluates the condition of the `if` statement of line 36 to `true`, and adds T_w in $T_r \rightarrow beforeMe$. This is a contradiction. □

Correctness of update transactions. Consider any execution α of WFR-TM. Throughout this section, we consider an update transaction T_w. By inspection of the code (lines 26 and 64), T_w is initiated as a read-only transaction and it becomes an update transaction after it first executes line 64.

Lemma 13. *Consider any instance V of* Validate *executed by T_w that returns* `true` *and let C_V be the configuration before the invocation of V. Then, for each triple $\langle x, -, d\rangle \in RS_{T_w}(C_V)$, d is consistent at C_V.*

Proof. Consider any triple $\langle x, -, d\rangle \in RS_{T_w}(C_V)$. We will prove that d is written by the last committed transaction that is serialized before C_V and updates x. By Lemma 6, there is a unique update transaction T_d that writes d in x (line 75). Since $\langle x, -, d\rangle \in RS_{T_w}(C_V)$ (i.e. T_w reads the version number d for x), Lemma 9 implies that T_d enters its waiting phase in α and $CW_{T_d} < C_V$.

Assume, by the way of contradiction, that the last committed transaction that is serialized before C_V is $T_{d'} \neq T_d$ which writes the value $d' \neq d$ for x (line 75). Since $CW_{T_d} < C_V$ and $T_{d'}$ is the last transaction that is serialized before C_V, by the way serialization points are assigned, it must be that $CW_{T_d} < CW_{T_{d'}} < C_V$. By Observation 3 (claim 2), CW_{T_d} occurs in α_{x,T_d} and $CW_{T_{d'}}$ occurs in $\alpha_{x,T_{d'}}$. Therefore, Lemma 5 (claim 2) implies that $\alpha_{x,T_d} < \alpha_{x,T_{d'}}$. Since both T_d and $T_{d'}$ update x, Lemma 6 (claim 1) implies that $d < d'$.

During the execution of V (and therefore, after C_V), T_w reads the version number of x (line 58); let r be this `read`. Since $CW_{T_{d'}} < C_V < r$, and, by Observation 3, $T_{d'}$ writes $d' > d$ for x before $CW_{T_{d'}}$, Lemma 6 (claim 1) implies that r returns either d' or a value larger that

d', as the version number of x. However, since V returns `true`, r must return d for x. This is a contradiction. □

Lemma 14. *If T_w enters its waiting phase in α, RS_{T_w} is consistent at CW_{T_w}.*

Proof. Let $\langle x, -, d\rangle$ be any triple added to the read-set of T_w. We prove that d is written by the last committed transaction that is serialized before CW_{T_w} and updates x. By Lemma 6 (claim 1), there is a unique update transaction T_d that writes d in x; let C_d be the configuration following this `write` (line 75).

Let V be the last instance of Validate (line 52) executed by T_w before CW_{T_w}; let C_V be the configuration preceding the invocation of V. Lemma 13 implies that d is consistent at C_V.

Since T_w enters its waiting phase, by inspection of the code (lines 70 - 71), it follows that the instance D of LockDataSet executed by T_w returns `true`. Since D returns `true`, by inspection of the code (lines 81, 89, and 90), it follows that, in D, the l-cas for x that T_w executes is successful. By inspection of the code (lines 82 - 83, and 89), this `CAS` uses d as the version number of its second parameter. Since it is successful, no transaction updates x between CW_{T_d} and CL_{x,T_w}.

Assume, by the way of contradiction, that the last committed transaction $T_{d'}$ that updates x and is serialized after C_V and before CW_{T_w}, writes the value $d' \neq d$ to x. Since $CW_{T_d} < C_V$ and $T_{d'}$ is the last transaction that is serialized between C_V and CW_{T_w}, by the way serialization points are assigned, it must be that $CW_{T_d} < CW_{T_{d'}} < CW_{T_w}$. By Observation 3 (claim 2), CW_{T_d}, $CW_{T_{d'}}$ and CW_{T_w} occur in α_{x,T_d}, $\alpha_{x,T_{d'}}$ and α_{x,T_w}, respectively. Therefore, Lemma 5 (claim 2) implies that $\alpha_{x,T_d} < \alpha_{x,T_{d'}} < \alpha_{x,T_w}$. By Observation 3 (claim 4), it follows that $T_{d'}$ updates x between CW_{T_d} and $CW_{T_{d'}}$. Since $\alpha_{x,T_{d'}} < \alpha_{x,T_w}$, it follows that $T_{d'}$ updates x between CW_{T_d} and CL_{x,T_w}. This contradicts our claim above that no transaction updates x between CW_{T_d} and CL_{x,T_w}. □

We are now ready to argue that WFR-TM is opaque.

Theorem 15. WFR-TM *is an opaque* `TM` *algorithm.*

Proof. Consider any execution α produced by WFR-TM and let H_α be the history of α. Choose any history H' from $Complete(H_\alpha)$ such that all transactions that enter their waiting phase in α commit in H'.

Recall that we have assigned a serialization point to all read-only transactions that commit and to those update transactions that enter their waiting phase in α. We assign a serialization point to each transaction T that aborts in H'. If T has performed at least one successful instance of ReadTvar, we place this point at the configuration just before T's last invocation of any instance of Validate that returns `true`. Otherwise, we place the serialization point of T at an arbitrary point within its execution interval. Notice that, once we do so, all transactions in H' have been assigned a serialization point.

Let $\ell_\alpha = T''_1, T''_2, \ldots$ be the sequence of transactions in H' in the order defined by their serialization points. Let $S = H'|T''_1, H'|T''_2, \ldots$ be a sequential history. By definition, S is equivalent to H'. Moreover, by the way serialization points are assigned to aborted transactions and by Lemma 7, the serialization point of every transaction T is within T's execution interval. Thus, S respects the real-time order induced by H'_α.

It remains to show that S is legal. Consider any transaction T in S. If T is a read-only transaction that commits in H', Lemma 12 implies that T is legal in S. If T is an update transaction that commits in H', Lemma 14 implies that T is legal in S. Thus, assume that T is a transaction that aborts in H'. If T has not performed any successful instance of ReadTvar, then T is trivially legal in S. Assume finally that T has performed at least one successful instance of ReadTvar. By inspection of the code (lines 52-53 and lines 70-71), T aborts either during the execution of its last instance of ReadTvar (because the invocation of Validate by that instance returns `false`), or during the execution of CommitTx (because LockDataSet returns `false`). In the first case, the invocation of Validate by all previous ReadTvar invoked by T has returned `true`. In the second case, the invocation of Validate in the last invocation of ReadTvar performed by T has returned `true`. Thus, Lemma 13 implies that T is legal in S. □

3.1.4 Proof of Progress.

In this section, we show that WFR-TM is wait-free for read-only transactions, and that update transactions are not prone to deadlock.

Let α be an execution of WFR-TM. Let m_w be the maximum number of t-variables written by any update transaction in α and m_r be the maximum number of t-variables read by any read-only transaction in α.

Lemma 16. *Consider any transaction T executed by some process p_i in α. Then, $T \rightarrow beforeMe$ contains at most two transactions initiated by each process p_j, $1 \leq j \leq n$, $j \neq i$.*

Proof. Notice that new elements are added to $T \rightarrow beforeMe$ only during the execution of CheckIfPerformed by T; specifically, this occurs with the execution of line 37. We will prove that line 37 may be executed by T at most twice for each entry $A[j]$, $1 \leq j \leq n$, $j \neq i$. We remark that since $T \rightarrow wset = \emptyset$ during the execution of CheckIfPerformed by T, T is considered as a read-only transaction as long as it executes its CheckIfPerformed.

Fix any j, $1 \leq j \leq n$, $j \neq i$. To obtain a contradiction, suppose that line 37 is executed by T three times for $A[j]$. Notice that before executing line 37, T reads (on line 35) the `txrec` record of some transaction from $A[j]$; let r_1, r_2, and r_3 be the reads of line 35 in those `for` iterations in which the first, the second, and the third execution, respectively, of line 37 for $A[j]$ occurs by T.

Let T_1, T_2, and T_3 be the transactions returned by r_1, r_2, and r_3, respectively. Notice that T_1, T_2, and T_3 have the same initiator p_j. Since the first execution of line 37 occurs after r_1, the second after r_2, and the third after r_3, by inspection of the code (1st condition of line 36), it follows that $T_1 \neq T_2 \neq T_3$. Moreover, by inspection of the code (3rd condition of line 36), the statuses of T_1, T_2, and T_3 are either `WAITING` or `COMMITTED` when the condition of the `if` statement of line 37 is evaluated after r_1, r_2, and r_3, respectively. So, by inspection of the code (lines 71 - 72, 73, 76, and 78), T_1, T_2, and T_3 do not abort.

By inspection of the code (lines 29, 77 - 78, and 94 - 97), T_1, T_2, and T_3 call WaitReaders after CW_{T_1}, CW_{T_2}, and CW_{T_3}, respectively. Recall that T is considered as a read-only transaction while executing its instance of CheckIfPerformed.

Assume first that the announcement of T precedes the announcement of T_1, thus it precedes CW_{T_1}. So, T_1 waits (line 97) until the instance of CheckIfPerformed initiated by T returns. Therefore, p_j cannot initiate T_2 as long as T executes its instance of CheckIfPerformed. This contradicts the fact that r_2 returns T_2.

Assume now that the announcement of T follows the announcement of T_1. Notice that T_2 is initiated by p_j after the completion of T_1. Since T reads T_1 on line 35 from $A[j]$ (though r_1) and r_1 follows the announcement of T (line 29), it follows that T is announced before the announcement of T_2, and, therefore, also before CW_{T_2}. So, T_2 waits (line 97) until the instance of CheckIfPerformed initiated by T returns. Thus, p_j cannot initiate T_3 as long as T is active. This contradicts the fact that r_3 returns T_3. □

We implement the $beforeMe$ set of each transaction T as a two-dimensional array of $2n$ elements. Then, a search in $T \rightarrow beforeMe$ is executed in $O(1)$ steps. Specifically, the array must have as many rows as the number of processes and two columns, so that two array elements are assigned to each process. Since each process may have at most one transaction active at each point in time, Lemma 16 implies that $T \rightarrow beforeMe$ contains at most two transactions from those initiated by any process p_j, $1 \leq j \leq n$, other than the process p_i that executes T; pointers to these two transactions are stored in the elements of row j of the $beforeMe$ set of p_i. To search if a transaction T' exists in its $beforeMe$ set, p_i reads the initiator p_j of T' from T''s `txrec`, and then it checks if a pointer to T' exists in any of the two elements of row j in its $beforeMe$ array. Thus, searching in the $beforeMe$ set of a process can be performed in $O(1)$ steps. Notice that each transaction must initiate each element of the $beforeMe$ array of its initiator to NULL when it executes BeginTx.

Lemma 17. *Consider any transaction T in α. Then, T completes* BeginTx *within $O(n^2)$ steps.*

Proof. T executes lines 23 - 29 in $O(1)$ steps. Thus, it remains to show that CheckIfPerformed completes in $O(n^2)$ steps. Lemma 16 implies that no more than $2(n-1)$ elements are added in $T \rightarrow BeforeMe$. Thus, no more than $O(n)$ iterations of the `do while` loop are executed. Each iteration reads n elements of the announce array. This results in $O(n^2)$ steps. We remark that each iteration of the `do while` loop additionally performs a search in $T \rightarrow beforeMe$. Recall that if we implement the $beforeMe$ set of T as a two-dimensional array of $2n$ elements, then this search is executed in $O(1)$ steps. □

Theorem 18. *Each read-only transaction commits after $O(n^2 + m_r m_w)$ steps, i.e.* WFR-TM *is wait-free for read-only transactions.*

Proof. Lemma 17 implies that T_r completes BEGINTX within $O(n^2)$ steps. It remains to prove that each instance of READTVAR executed by T_r completes in $O(m_w)$ steps.

Since T_r is a read-only transaction, $T_r \rightarrow wset = \emptyset$. Thus, lines 43 - 44, and 52 are executed in $O(1)$ steps. Lines 45, 46, and 51 execute only local computations. All remaining lines other than 49 are also executed in $O(1)$ steps. Notice that the second condition of line 49 performs a search on the $beforeMe$ set of T_r for transaction *owner*. Recall that if we implement the $beforeMe$ set of T_r as a two-dimensional array of $2n$ elements, then this search can be executed in $O(1)$ steps. Thus, the only condition whose evaluation may cause the execution of more than $O(1)$ steps, when executing line 49, is the condition "$tvar \in owner \rightarrow wset$". The evaluation of this condition requires $O(m_w)$ steps. Thus, each instance of READTVAR executed by T_r, completes within $O(m_w)$ steps.

By inspection of the code (lines 67 to 69), it follows that COMMITTX, when called by a read-only transaction, completes within $O(1)$ steps. Thus, T_r completes its execution within $O(n^2 + m_r m_w)$ steps. □

Consider now an update transaction T_w. By Theorem 18 and by inspection of the code, it follows that for each read-only transaction T_r, T_w may wait (on line 88 or 97) only for a finite number of steps in order for T_r to complete.

Theorem 19. *In any infinite execution of* WFR-TM*, each update transaction T_w completes within a finite number of steps.*

Proof. Lemma 16 implies that $T_w \rightarrow beforeMe$ is finite. Since $T_w \rightarrow wset$, and T_w's read-set are also finite, by inspection of the code, it follows that CREATETVAR, WRITETVAR, VALIDATE, and LOCKDATASET, when called by T_w, complete within a finite number of steps. By inspection of the code (lines 94 - 97), T_w may have to wait for the completion of at most $n - 1$ read-only transactions while executing WAITREADERS. Theorem 18 implies that, for each read-only transaction T_r, T_w waits for a finite number of steps in order for T_r to complete. Thus, WAITREADERS completes within a finite number of steps and therefore the same is true for COMMITTX. □

Theorem 20 proves that WFR-TM provides deadlock-freedom for update transactions.

Theorem 20. *In any infinite execution α of* WFR-TM *in which infinitely many update transactions are initiated, infinitely many update transactions commit.*

Proof. To obtain a contradiction, assume that no update transaction ever commits after some configuration C of α. Then, Theorem 19 implies that infinitely many transactions abort after C. By inspection of the code (lines 52 - 54, 70, and 71), an update transaction T_w aborts either when one of the instances of VALIDATE (line 52), executed by T_w, returns `false`, or when the single instance of LOCKDATASET, executed by T_w during COMMITTX, returns `false`. In the first case, by inspection of the code of VALIDATE, it follows that the version of at least one t-variable has changed since it has been initially read by T; let this update be performed by some transaction T'_w. By inspection of the code (lines 73 - 79) and Theorem 19, it follows that T'_w commits within a finite number of steps. Since no transaction commits after C, it follows that only a finite number of instances of Validate can return `false`, after C.

Let C' be the configuration following the return of the last instance of Validate that returns `false`, after C. So, any update transaction T'_w initiated after C' aborts because the instance D' of LockDataSet it executes returns `false`. By inspection of the code (lines 85 - 86 and 90), D' returns `false` when a t-variable in $RS_{T'_w}$ is locked by some other transaction. By inspection of the code (line 81), each transaction acquires the locks of the t-variables it accesses in (ascending) order. So, there is at least one transaction initiated after C' for which the instance of LockDataSet executed by it will be able to acquire all the required locks and respond with `true`. This is a contradiction. □

3.2 Case Study II: Dense, A Concurrent Graph Algorithm

In `Dense`, operations are wait-free, i.e. an operation by a process that does not fail terminates in a finite number of steps in any execution. Wait-freedom is achieved by employing light-weight helping. Operations are aware of concurrent active dynamic traversals and ensure that those dynamic traversals can return a consistent view by storing old edge versions for them (in the worst case, `Dense` keeps n different versions, one for each process, on a given edge of the graph). The edges are stored in an *adjacency matrix*, which is used for the graph's representation. Thus, `Dense` is so named because it is mostly suitable for dense graphs, i.e. graphs with high connectivity, in which case the allocated adjacency matrix is sufficiently exploited.

In the following, we provide a detailed description of `Dense` and a formal proof of the properties we claim for it.

Author's contribution. The contents of this section are joint work that has been accepted for publication in OPODIS 2015 [KK15]. The author contributed to the algorithm design and proof of correctness of the algorithm presented in this section.

3.2.1 Overview and Main Ideas

A graph $G = \langle V, E\rangle$ is composed of V, a (finite) set of elements referred to as *vertices*, and E, a set of pairs of vertices, referred to as the *edges* between them. Each edge $e_{ij} \in E$ has a weight w_{ij}, that takes values out of some set W. A graph supports several abstract operations, well-known in literature, such as operations for adding vertices or edges, deleting vertices or edges, modifying attributes of vertices or edges, returning specific subsets of the graph vertices or edges, etc. A *concurrent graph* is a graph that can be accessed concurrently, through those types of operations, by n processes.

We propose the *dynamic traversal* (which henceforth may be referred to as d-traversal for brevity) as a concurrent graph operation that exhibits the following characteristics: (i) it starts from a vertex v of the graph; (ii) it visits a sequence of vertices that is not necessarily known at the point that the dynamic traversal initiates; (iii) the sequence of visits may be decided while the visiting is taking place; (iv) the dynamic traversal returns a *consistent view* of the weights of all the edges that it has traversed, i.e., all the returned values have co-existed on the graph at some point in time.

We rely on the following concurrent graph representation. The graph is represented as an $m \times m$ adjacency matrix, for some positive integer m, and it allows the addition of edges, the removal of edges, and the modification of edge weights by providing an updating operation. This operation is `UpdateEdge`(i, j, w), where i, j are indices of vertices in V and where w is in $W \cup \{\perp\}$. It modifies the graph as follows: Assume that $e_{i,j} \in E$. If $w = \perp$, then the edge is removed. Otherwise, its weight is changed to w. If $e_{i,j} \notin E$, it is inserted in E with weight w.

The implementation supports the d-traversal as a composite operation, consisting of the following ones:

- `DynamicTraverse`, which is used to mark the beginning of a d-traversal of the graph.
- `EndTraverse`, which is used to mark the end of a d-traversal of the graph.
- `ReadEdge`(i, j), where i, j are indices of vertices in V. It returns a weight for edge $e_{i,j}$, if $e_{i,j} \in E$, and $\perp$ if $e_{i,j} \notin E$.

An instance of `ReadEdge` is only used in d-traversals, potentially as part of a sequence of `ReadEdge` operations. A d-traversal by process p_u consists in an instance *bt* of a `DynamicTraverse` operation, followed by a finite sequence of instances of `ReadEdge`, followed in turn by an instance *et* of an `EndTraverse` operation. No other operation is invoked between *bt* and *et*. The execution interval of the d-traversal starts in the configuration in which p_u invokes *bt* and ends in the configuration resulting from the response of *et*.

Although we consider linearizability as the correctness criterion for `UpdateEdge` operations, for the d-traversals we consider a criterion analogous to strict serializability, since they constitute complex operations that are reminiscent of restricted transactions. We reformulate the definition of linearizability of Section 2.2 in order to adapt it to the necessities of our graph model.

Definition 3.1 (Linearizability for `Dense` executions with dynamic traversals)**.** *An execution α of* `Dense` *that contains dynamic traversals is linearizable if it is possible to assign a linearization point inside the execution interval of each completed operation in α and possibly some of the incomplete operations in α, and a linearization point in each completed dynamic traversal in α and possibly some incomplete ones, so that the result of each of those operations and dynamic traversals is the same as it would be, if they had been performed sequentially in the order dictated by their linearization points.*

Roughly speaking, we consider that the entire sequence of `ReadEdge` operations enclosed in a dynamic traversal have a linearization point inside the execution interval of the d-traversal, such that the `ReadEdge` return the weights that the traversed edges had in the configuration in which the linearization point is placed. The provided graph operations as well as the d-traversal, are wait-free.

3.2.2 Algorithm Description

Data Structures. Algorithm 4 shows the data structures used by `Dense` (initial values are indicated on lines 17 - 21). Operation information is stored in a structure of type *AnnStruct*. This structure consists of four fields, namely: (i) *op*, of type *OpType*, which represents operations provided by `Dense` (i.e. `DynamicTraverse`, `UpdateEdge`, and the void operation `Noop`); (ii) i and j which identify the edge on which an `UpdateEdge` operation is to be performed (if op = `UpdateEdge`); and (iii) *value*, an integer representing the value that an `UpdateEdge` operation has to write to the weight of the edge specified by fields i and j (if op = `UpdateEdge`).

Algorithm 4 `Dense`: Data structures for a multi-traverse implementation of a concurrent graph object suitable for dense graphs.

```
1  typedef OpType {DynamicTraverse, UpdateEdge, Noop}; // codewords for announced operations

2  type AnnStruct // the data type of the announce array elements
3    OpType $op$ // the announced operation
4    int $i, j$ // if OpType=Update, i and j denote the vertices connected by the edge to be updated
5    int $value$ // the weight to be assigned to the edge if OpType=Update


6  type StateStruct // the data type of the structure storing the graph's state
7    boolean $phase$ // a field indicating the current phase of execution, either Announce or Apply
8    int $seq$ // the sequence number, used as a version counter
9    int $ann[1..n]$
10   int $done[1..n]$
11   int $rvals[1..n]$ // an array storing a value of seq for each process in order to facilitate dynamic traversals
12

13 type EdgeStruct // the data type of a graph edge
     // each array element corresponds to a process and stores a weight and its version
14   ⟨ weightval, int ⟩ $prev[1..n]$
15   int $seq$ // current version of the edge
16   weightval $w$ // current weight of the edge


17 shared int $BitVector[1..n] = 0$; // used as a vector of $n$ bits, one for each process

   // announce array
18 shared AnnStruct $Announce[1..n] = \{\langle \texttt{Noop}, 0, 0, 0\rangle, \ldots, \langle \texttt{Noop}, 0, 0, 0\rangle\}$;

19 shared StateStruct $ST = \langle 0, \texttt{AGREE}, \langle 0, \ldots, 0\rangle, \langle 0, \ldots, 0\rangle, \langle 0, \ldots, 0\rangle\rangle$; // graph state

   // adjacency matrix representing the graph
20 shared EdgeStruct $Edges[1..m][1..m] = \{\langle\langle 0, 0\rangle, 0, 0\rangle, \ldots, \langle\langle 0, 0\rangle, 0, 0\rangle\}$;

21 private int $toggle_u = 2^u$;  // there is a copy for each process $p_u, u \in \{1, \ldots, n\}$
```

Our implementation provides linearizable, wait-free operations and d-traversals by using light-weight helping. To achieve it, each `UpdateEdge` or `DynamicTraverse` operation is first *announced* by a process, subsequently *agreed* by all processes, and then it can be *applied* by some process - not necessarily the one that invoked it. Finally, it can terminate and return a response. Furthermore, after being agreed and before being applied, an instance u of `UpdateEdge` may perform a modification on a graph edge, in which case we say that u has *taken effect*. In the same way that an instance of any operation may be applied by a process other than the one that invoked it, u may take effect due to events invoked by a process other than the one that invoked u. In order to achieve the coordination that is necessary in order to apply operations or achieve that updates take effect, the processes collaborate in order to alternate between two types of phases, namely `AGREE` and `APPLY`.

The status of operations on the graph is indicated by ST, an `LL/SC` object of type $StateStruct$ consisting of: (i) $phase$, a boolean variable which indicates whether the execution of `Dense` is in an `AGREE` or an `APPLY` phase at any given moment; (ii) seq, an integer which serves as global version counter. It is incremented each time a process successfully switches the execution phase from `AGREE` to `APPLY`; (iii) $ann[1..n]$, an array implemented as n-bit integer, where $ann[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and whose value is toggled each time an operation by p_u is agreed; (iv) $done[1..n]$, an array implemented as n-bit integer, where $done[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and whose value is set equal to $ann[u]$ each time an operation by p_u is applied to the graph; and (v) $rvals[1..n]$, an array of n elements, where t $rvals[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and which stores the value of seq that p_u uses as read version number, in case it is performing a d-traversal.

The `AGREE` phase is used by processes in order detect which operation information in the announce array corresponds to a pending operation: p_u has a pending operation if the u-th bit of the bitvector is not equal to $done[u]$. In this phase, processes essentially "agree" on a set of operations that they will attempt to apply on the graph in the following `APPLY` phase. Then, the `APPLY` phase that follows is used by processes for attempting to apply those pending operations. As a result, operations are applied to the graph in batches. When an announced operation is carried out by some process, we say that it is applied. Otherwise, it is pending. An applied operation can return a response to the process that invoked it. The status of an operation, i.e. whether it has been already applied or not, is reflected in the values of $ST.ann[u]$ and $ST.done[u]$: An invariant in our implementation is that in configurations in which it holds that $ST.ann[u] = ST.done[u]$, it also holds that the latest agreed operation by p_u has been applied; while in configurations in which $ST.ann[u] \neq ST.done[u]$, it also holds that the latest operation by p_u is pending. A process which completes the actions associated with a phase, attempts to flip it.

We represent the graph G with $Edges$, an adjacency matrix, i.e. a two-dimensional array, where each element (i, j) of the array represents edge between vertices i and j, $i, j \leq m$. Graph edges, i.e. adjacency matrix elements, are `LL/SC` objects of type $EdgeStruct$. This type is a record of three fields: (i) $prev$, an array of n elements (one for each process), where each element

Algorithm 5 `Dense`: Operations Update, DynamicTraverse, and EndTraverse, auxiliary routine Read, for a multi-traverse implementation of a concurrent graph object suitable for dense graphs.

```
void UpdateEdge(int $i$, int $j$, int $value$) // for process $p_u, u \in \{1, \ldots, n\}$
  BTU(UpdateEdge, int $i$, int $j$, int $value$)

void DynamicTraverse() // for process $p_u, u \in \{1, \ldots, n\}$
  BTU(DynamicTraverse, $\perp$, $\perp$, $\perp$);

void EndTraverse() // for process $p_u, u \in \{1, \ldots, n\}$
  noop

int ReadEdge(int $i$, int $j$) // for process $p_u$
  EdgeStruct $edge$
  int $val$, int $seq$, int $rval$
  int $rval = ST.rvals[u]$
  $edge = Edges[i][j]$
  if ($edge.seq > rval$) then
    $\langle val, seq \rangle = edge.prev[u]$
  else $val = edge.w$
  return $val$
```

is a pair $< w, seq >$ of integers. Whenever an update operation modifies the weight of an edge, it stores the current weight and version in $prev[u]$ if process p_u is performing a d-traversal on the graph using as read value, stored in $ST.rvals[u]$, a value that is larger than the current version of the edge; (ii) seq, an integer which stores the current version of the edge; (iii) w, of type $weightval$, which stores the current weight of the edge - if this value is $\perp$, the corresponding edge does not exist.

Recall that `Dense` implements the helping mechanism, where any process p_u that invokes an operation also attempts to apply pending operations by other processes. Operation information is stored by processes in $Announce[1..n]$, an announce array of n elements, where each element $Announce[u]$, $u \in \{1, 2, \ldots, n\}$, is of type $AnnStruct$ and can be written to only by process p_u, but can be read by all processes. The announcing of an operation is complemented by the use of $BitVector$, shared vector of n bits (represented as a n-bit integer) where bit u corresponds to process p_u as follows: In order to indicate a pending operation, after each time p_u writes new operation information in $Announce[u]$, it flips the u-th bit of $BitVector$. It does so with the aid of a local, persistent variable, $toggle_u$, with initial value 2^u. After p_u announces an operation, it inverts the value of $toggle_u$.

Pseudocode Description. Pseudocode for the operations of the graph that are described in Subsection 3.2.1 is presented in Algorithm 5. Operations `UpdateEdge` and `DynamicTraverse` require that the processes that execute them, assist each other. In order to do this, they both invoke auxiliary routine BTU (these initials stand for "*B*egin a *T*raversal or *U*pdate"). BTU implements the phase alternation and is further detailed below. We say that an execution of Dense is in AGREE or APPLY phase during those execution intervals in which $ST.phase$ = AGREE, or $ST.phase$ = APPLY, respectively. Notice that ReadEdge is independent of the phases. Instances of ReadEdge are only invoked by a process following the execution of a DynamicTraverse operation by the same process. They rely on UpdateEdge operations to store possibly useful old edge versions for them in the *prev* arrays of each modified edge. To achieve the synchronization that is necessary for this, d-traversals use the aforementioned concept of a read version number, as follows.

The DynamicTraverse operation that initiates some d-traversal d, obtains as read version number the current value v of $ST.seq$ (this happens when either the process that initiated d or some other process helps to apply this DynamicTraverse operation while executing line 64). An instance r of ReadEdge that is invoked by process p_u on edge $e_{i,j}$ and that is included in d, must check whether the version of $e_{i,j}$ is greater than v (line 32). If this is the case, then $e_{i,j}$ was updated after d started. However, in Dense, d-traversals must not be aware of the modifications of concurrent edge updates and have to return values that the edge weights had just before the d-traversal initiated. For this reason, r must return a previous weight of $e_{i,j}$, and finds this in $e_{i,j}.prev[u]$ (line 33). If the version of $e_{i,j}$ is less than v, then r returns $e_{i,j}$'s current weight (line 34). Notice that although the instances of ReadEdge that are included in a d-traversal are not aware of concurrent UpdateEdge instances (i.e. instances whose execution intervals overlap with that of the d-traversal), those UpdateEdge instances become aware of d-traversals and store the necessary old edge weights for them when they modify edges the graph.

Algorithm 5 presents BTU, which is at the heart of the Dense implementation. It is invoked by UpdateEdge specifying as arguments the operation type, integers i and j, which identify the edge to be modified, and integer *value*, which specifies the weight to be written to this edge. When BTU is invoked by DynamicTraverse, then only the operation type is specified as argument, while the remaining three are $\perp$, as they are not required for the d-traversal.

An instance of BTU by p_u first writes the operation information into element u of the announce array (line 40) and then sets the value of the u-th bit of $BitVector$ (line 41), using the current value of local persistent variable $toggle_u$ bit. It then flips $toggle_u$ (line 42) in order to prepare its value for the next execution of an operation by p_u. The algorithm implements this practice in order to provide a previously mentioned invariant: by comparing $ST.ann[u]$ and $ST.done[u]$, a process is able to detect whether the latest agreed operation by p_u has already been applied or not. Notice that the contents of $BitVector$ are copied into $ST.ann$ by each process that successfully executes an AGREE phase of Dense (lines 45, 48, 67), while they are copied into $ST.done$ by a process that successfully executes an APPLY phase of Dense (lines 45, 66, 67). Therefore, each operation by p_u must correspond to a different $BitVector[u]$ value than the previous one.

BTU carries out any light-weight helping in addition to the execution of the operation that invoked it. To do this, it iterates via a for loop (lines 43-67). An iteration of this for loop

Algorithm 6 `Dense`: ApplyOp routine for a multi-traverse implementation of a concurrent graph object suitable for dense graphs.

```
36 void BTU(OpType $op$, int $i$, int $j$, int $value$) { // for process $p_u, u \in \{1, \ldots, n\}$
37   StateStruct $st$
38   int $lbv$, $opi$, $opj$
39   EdgeStruct $e$

40   $Announce[u] = \langle op, i, j, value \rangle$
41   Add($BitVector$, $toggle_u$)
42   $toggle_u$ = - $toggle_u$

43   for $i$ up to 4 do {
44     $st$ = LL($ST$)
45     $lbv = BitVector$

46     if ($lbv[u] == st.done[u]$) then break

47     if ($st.phase ==$ AGREE) then // AGREE Phase
48       $st.ann[1..n] = lbv[1..n]$
49       $st.seq = st.seq + 1$
50       $st.phase$ = APPLY
51     else // APPLY Phase
52       for (r = 1; r ≤ n; r++) {
53         if ($st.ann[r] \neq st.done[r]$) then
54           if ($Announce[r].op ==$ UpdateEdge) then
55             $opi = Announce[r].i$
56             $opj = Announce[r].j$
57             $e$ = LL($Edges[opi][opj]$)
58             if ($e.seq < st.seq$) then
59               for (k = 1; k ≤ n; k++) {
60                 if ($e.seq < st.rvals[k]$) then
61                   $e.prev[k] = \langle e.w, e.seq \rangle$;
62               $e.w = Announce[r].value$
63               $e.seq = st.seq$
64               SC($Edges[opi][opj], e$)
           else $st.rvals[r] = st.seq$
65       $st.done[1..n] = lbv[1..n]$
66       $st.phase$ = AGREE
67     SC($ST, st$);
```

consists in locally copying ST (line 44), and then attempting to perform the actions that are required by the phase indicated in $ST.phase$. Once these actions have been performed, `BTU` attempts to change the phase by executing the `SC` of line 67. If this `SC` is successful, we say that `BTU` (or, abusing terminology, the process or the operation that invoked it) successfully executed the phase. The execution of this primitive may fail if some instance of `BTU`, executed by a process other than p_u, has already performed the current phase and advanced the execution to the next phase. When executing the `for` loop (lines 43 - 67), `BTU` proceeds as follows, depending on the phase it performs:

- AGREE phase (lines 47-50). This phase updates the status record ST with the newly announced operations, so that all processes can agree on them. So, BTU first records this status locally on st, before using an SC instruction in order to attempt to update it globally on ST. In order to set st, BTU collects information from the $BitVector$ regarding newly announced and therefore possibly pending operations. It does so by copying the contents of $BitVector$ into $st.ann$ (line 48). Notice that for a process $p_l, 1 \leq l \leq n$, that has a newly announced operation, the invariant $st.ann[u] \neq st.done[u]$ must hold. Therefore, a successful assignment of st to ST (through the execution of the SC of line 67) creates the inequality between $ST.ann[u]$ and $ST.done[u]$ and makes all processes "agree" that p_u has a newly announced operation which has not been applied yet. Once the information regarding pending operation for each process has been copied into st, BTU increments seq, the global version counter in st (line 49) and changes the $phase$ field of st from AGREE to APPLY.
- APPLY phase (lines 51 - 66) This phase applies any pending agreed UpdateEdge operation on the edges of the graph, and assigns read version number to any pending agreed DynamicTraverse operation. For this, BTU uses st again, and for each process p_u (line 52) it checks whether such a pending operation exists (line 53), in which case it holds that $st.ann[u] \neq st.done[u]$. Consider the case of a pending UpdateEdge operation by p_u on edge $e_{i,j}$. Since multiple processes may be executing an operation on $e_{i,j}$, these modifications must be synchronized in order to safeguard correctness. For this reason, $e_{i,j}$ is copied locally into e using LL (line 57). If the current version number of $e_{i,j}$, $e.seq$ is greater than $st.seq$ then the specific UpdateEdge operation has already taken effect, namely by some process other than p_u, that has also changed the state. However, if this is not the case, the modification of $e_{i,j}$ is carried out. Before setting the new value for the weight (line 62) and version (line 63) of $e_{i,j}$, a comparison of the current version of $e_{i,j}$ and all read version numbers stored in $st.rvals$ is performed (lines 59 - 61). If the current version of $e_{i,j}$ is less than the read version number for some process p_r, $1 \leq r \leq n$, then the condition $e.seq < st.rvals[r]$ is true. This means that a concurrent d-traversal by process p_r might be in progress. In order to guarantee that an eventual such d-traversal can read mutually consistent values, the current values of $e_{i,j}$'s weight and version are stored in $e.prev[r]$. There, instances of ReadEdge on $e_{i,j}$ that are included in a d-traversal, can later find it if necessary. BTU then attempts to finalize the update of $e_{i,j}$ by using SC to copy e into $e_{i,j}$ (line 64). Whether the SC on the edge is successful or not, at the end of the phase the operation is considered applied.

  If p_u's pending operation is a DynamicTraverse, the read version number must simply be set. This is first recorded in $st.rvals[u]$ (line 64) and is eventually stored in $ST.rvals[u]$ (line 67) by the process that successfully executes the phase. Recall that it is used by a concurrent UpdateEdge operation in order to judge whether to discard the current value of the edge that it is updating or whether to keep it for the ongoing d-traversal of p_u.

If the assignment of line 64 followed by a successful SC on ST is executed more than once for a given DynamicTraverse instance or for the d-traversal that it initiated, then the consistency of the ReadEdge instances of the d-traversal could be compromised. An eventual bad scenario would happen if ReadEdge instances that are invoked before the second execution of those lines and ReadEdge instances that are invoked after the second execution would use a different read version number when reading edges.

Thus, at the end of an APPLY phase, the *done* bits in *st* are set equal to the corresponding *ann* bits (line 66). Then, BTU attempts to change the phase from APPLY back to AGREE (line 66) by switching the *phase* field of *st*, which is reflected on ST if the SC instruction of line 67 is successful.

Notice that an instance of BTU may be slow and end up performing the actions associated with a phase while the execution has already progressed to some following phase. Notice also that in the worst case, an instance of BTU has to perform four iterations of the for loop before the operation that invoked it is applied. Such a worst-case scenario is the following: Let I_{btu} be an instance of BTU that executes the first iteration of the for loop during an AGREE phase and let p_l be the process that successfully flips the phase to APPLY by executing the SC on ST of line 67. Consider however that the execution of line 41 by I_{btu} occurs after p_l executes the LL of line 44, which corresponds to the successful SC on ST. This means that in the following APPLY phase, the operation that invoked I_{btu} will not be executed. In the worst case, all other processes are slow and the process that invoked I_{btu} must perform the actions associated with the APPLY phase itself, during the second iteration of the for loop, as well as the actions required by the following AGREE phase, during the third iteration of its for loop. During this AGREE phase, the *Add* on $BitVector$ by I_{btu} is guaranteed to be observed by the process that performs the successful SC on ST and changes the phase to APPLY. Here again, in the worst case, all other processes are once more slower than the process which invoked I_{btu}, and thus, I_{btu} that performs the actions associated with the APPLY phase, in its fourth iteration of the for loop. This time, however, the operation that invoked it is guaranteed to have been applied.

However, in the common case, the operation may be applied earlier, by some other, helping process. The condition that signals this is expressed on line 46 and is checked at each iteration of the for loop. It consists in verifying whether the toggle bit for p_l, the process executing BTU, in shared array $BitVector$ has the same value as the corresponding bit in the $ST.done$ array. If that is the case, the operation executed by BTU is considered applied and the iteration of the for loop terminates as well.

3.2.3 Proof of Correctness

Let α be an execution of Dense. Such an execution is comprised of instances of operations UpdateEdge and DynamicTraverse, which in turn invoke instances of auxiliary routine BTU, as well as of instances of EndTraverse and auxiliary routine ReadEdge. We may refer to instances of operations UpdateEdge and DynamicTraverse as *requests*. The execution interval of an instance of UpdateEdge begins with its invocation and terminates when it returns. Similarly,

SC_k^{ST}	the k-th successful `SC` on ST.
LL_k^{ST}	the `LL` that corresponds to SC^{ST}.
C_k^{ST}	he configuration resulting from the execution of SC_k^{ST}.
Q_w^u	the w_u-th *Add* of line 41 executed by p_u.
CA_w^u	the configuration after the execution of Q_w^u.
$\mathcal{L}_{\mathcal{DT}}$	the sequence of `DynamicTraverse` operations that have been assigned linearization points in α, based on the order of their linearization points.
$\mathcal{L}_{\mathcal{DT}}^{C}$	the prefix of $\mathcal{L}_{\mathcal{DT}}$ from C_0 up to C, for some configuration C in α.
$\mathcal{L}_{\mathcal{U}}$	the sequence of `UpdateEdge` operations that have been assigned linearization points in α, based on the order of their linearization points.
$\mathcal{L}_{\mathcal{U}} \vert e_{i,j}$	the projection of $\mathcal{L}_{\mathcal{U}}$ on `UpdateEdge` operations that affect edge $e_{i,j}$.
$\mathcal{L}_{\mathcal{U}}^{C} \vert e_{i,j}$	the prefix of $\mathcal{L}_{\mathcal{U}} \vert e_{i,j}$ from C_0 up to C, for some configuration C in α.
$SC_k^{e_{i,j}}$	the k-th successful `SC` operation on $e_{i,j}$.

Table 3.2: Notation used during the proof of `Dense`.

the execution interval of an instance of `DynamicTraverse` (or `EndTraverse`) begins with its invocation and terminates when it returns (see Algorithm 5). The execution intervals of routines `BTU` and `ReadEdge` are defined accordingly. The execution interval of a d-traversal begins with the invocation of the instance of `DynamicTraverse` that initiates it and terminates with the response of the instance of `EndTraverse` that finished it.

Consider an instance U of `UpdateEdge`, with arguments i, j and v and let p_u, $u \in \{1, 2, \ldots, n\}$ be the process executing it. We then say that U *updates* edge $e_{i,j}$ with value v. Let now R be an instance of `ReadEdge` with arguments i and j. When R executes line 31 we say that R *reads* edge $e_{i,j}$.

In the following, we prove that the operations provided by `Dense` are linearizable and wait-free. We start with some technical characteristics of the algorithm, which are then used in order to argue about the claimed properties. Table 3.2 briefly summarizes the notation introduced thus far, as well as some notation that will be introduced later. Note that notation that refers to some configuration starts with the letter C.

Preliminaries. Let p_u, $u \in \{1, 2, \ldots, n\}$ be one of the processes that execute `Dense` in α. Recall that `Dense` relies on the shared variables $Announce[1..n]$, $BitVector[1..n]$, and ST in order to achieve process synchronization. Given that processes have local variables that share denomination, we distinguish between them with a subscript indicating the id of the process they belong to – e.g. local variable lbv of process p_u is referred to as lbv_u. Regarding the shared variables, inspection of the pseudocode shows that the following hold.

Observation 21. *$Announce[u]$ is only modified by the execution of line 40 by an instance of* `BTU` *executed by p_u, $u \in \{1, 2, \ldots, n\}$.*

Observation 22. *$BitVector[u]$ is only modified by the execution of line 41 by an instance of* `BTU` *executed by p_u, $u \in \{1, 2, \ldots, n\}$.*

Observation 23. *ST is only modified by a successful execution of the* `SC` *operation of line 67 by an instance of* `BTU` *executed by some process p_u, $u \in \{1, 2, \ldots, n\}$.*

We start by proving some useful properties of ST. We refer to the `SC` operation of line 67 as $st-sc$ and the `LL` operation of line 44 as $st-ll$. Denote by $SC_1^{ST}, SC_2^{ST}, \ldots$ the sequence of such successful operations on ST in α and by $LL_1^{ST}, LL_2^{ST}, \ldots$ the sequence of corresponding $st - ll$ operations. We denote the initial configuration by C_0. Let C_k^{ST} be the configuration resulting from the execution of SC_k^{ST}, $k > 0$. By Observation 67 and the definition of $SC_1^{ST}, SC_2^{ST}, \ldots$, it is straight-forward to show the following lemma.

Lemma 24. *ST is not modified in the execution interval between C_k^{ST} and (but not including) C_{k+1}^{ST}, $k > 0$.*

Assuming that the initial value of *ST.phase* is `AGREE`, then:

Lemma 25. *If ST.phase has the value* `AGREE` *in the configuration just before SC_k^{ST}, $k > 0$, is executed, then it has the value* `APPLY` *in C_k^{ST}. Conversely, if ST.phase has the value* `APPLY` *in the configuration just before SC_k^{ST}, $k > 0$, is executed, it has the value* `AGREE` *in C_k^{ST}.*

Proof. We prove the claim by contradiction.

Fix a $k > 0$ and assume first that in the configuration in which SC_k^{ST} is executed, $ST.phase =$ `AGREE`. Let p_u, $u \in \{1, 2, \ldots, n\}$, be the process that executes SC_k^{ST}. To arrive at a contradiction, assume that when LL_k reads ST, the value of *ST.phase* is not `AGREE`. This is a contradiction, since, by Observation 67, ST only changes through successful `SC` operations of line 67, and by definition, SC_k^{ST} is such an instance. By similar reasoning, that is also the value that *ST.phase* has in the configuration just before SC_k^{ST} is executed.

By inspection of the pseudocode, if p_u executes LL_k and finds that $ST.phase =$ `AGREE`, (line 47), it executes lines 48 - 50 before executing SC_k^{ST} on line 67. Notice that any successful $st - sc$ assigns the value of local variable *st* to ST. Notice also that *st.phase* is assigned the value `APPLY` on line 50. Therefore, in C_k^{ST}, $ST.phase =$ `APPLY` and the claim holds.

By analogous reasoning, we prove that if $ST.phase =$ `APPLY` in the configuration just before SC_k^{ST}, then $ST.phase =$ `AGREE` at C_k^{ST}. □

The previous lemma implies the following corollary:

Corollary 26. *Any SC_k^{ST} such that $k \mod 2 = 1$ changes ST.phase from* `AGREE` *to* `APPLY`*. Any SC_k^{ST} such that $k \mod 2 = 0$ changes ST.phase from* `APPLY` *to* `AGREE`*.*

Lemma 27. *For any $k > 0$,*

1. *At C_k^{ST}, $k > 0$, the value of ST.seq is $\lceil \frac{k}{2} \rceil$*

2. *The value of ST.seq does not change between C_k^{ST} and the configuration in which SC_{k+2}^{ST} is executed, for $k > 0$, $k \mod 2 = 1$.*

Proof. Recall that, by Observation 67, ST, and therefore $ST.seq$, is only modified by the `SC` instruction of line 67 and that a successful `SC` operation assigns to it the value of local variable st. The value of st for each successful `SC` is determined either in the `for` or the `else` branch of the `for` loop of lines 47, 51.

By Lemma 67 we have that each SC_k^{ST} toggles $ST.phase$ from `AGREE` to `APPLY` and vice versa. Recall that inspection of the pseudocode shows that, since each $st - ll$ copies the value of ST into the local variable st of the process p executing $st - ll$ (line 44), this also holds for LL_k^{ST}. Since, according to the pseudocode, $st.seq$ is incremented only during the `AGREE` phase (line 49), this means that it is not modified during the `APPLY` phase. Then, it is incremented by those SC_k^{ST} which toggle $ST.phase$ from `AGREE` to `APPLY`. By assumption, initially it holds that $ST.phase =$ `AGREE`. It follows that $SC_1^{ST}, SC_3^{ST}, \ldots$, increment $ST.seq$, proving the claims. □

Lemma 67 implies the following corollary.

Corollary 28. *$ST.seq$ is monotonically increasing in α.*

Toggle bits, Done bits, and BitVector. We proceed by examining how the values of $BitVector[1..n]$, as well as $ST.ann[1..n]$ and $ST.done[1..n]$ change during α.

Observation 29. *Each request invokes one instance of* `BTU`.

We denote by m_u the number of requests executed by a process p_u, $u \in \{1, 2, \ldots, n\}$, in α. Each process p_u has a persistent local variable $toggle_u$. Let req_u^w be the w-th request invoked by p_u. Let the initial value of $toggle_u$ be 2^u and let $toggle_u^w$ be the value of $toggle_u$ in the configuration right after request req_u^w has been executed.

Observation 30. *For any w, $0 \leq w \leq m_u$, the following holds:*

1. *if $w \mod 2 = 0$, $toggle_u^w = 2^w$*
2. *if $w \mod 2 = 1$, $toggle_u^w = -2^w$*

By inspection of the pseudocode, we have that local variable $toggle_u$ is added by p_u to $BitVector[1..n]$ by the execution of the Add primitive of line 41.

Let C be some configuration in α. Then the following lemma holds.

Lemma 31. *For each $u \in \{1, 2, \ldots, n\}$, if p_u has executed $w_u \geq 0$ Add on $BitVector[1..n]$ by C, it holds that $BitVector[u] = w_u \mod 2$ at C.*

Proof. Fix any $u \in \{1, \ldots, p\}$. We prove the claim by induction on w_u.

Base case ($w_u = 0$). By the way $BitVector$ is initialized and by Observation 22, it follows that $BitVector[u] = 0$ at C_0. Since p_u has not performed any request by C, it holds that $w_u \mod 2 = 0$ and the claim follows.

Induction hypothesis. Fix any $w_u > 0$ and assume that the claim holds.

Induction step. We prove that the claim holds for $w_u + 1$.

First assume that $w_u + 1 \mod 2 = 1$. Then, $w_u \mod 2 = 0$, and, by the induction hypothesis, $BitVector[u] = 0$ in the configuration right after the w_u-th *Add* by p_u is executed. In that configuration, by Observation 30, it also holds that $toggle^u_{w_u} = 2^{w_u}$. By inspection of the pseudocode, we have that this still holds in the configuration in which the (w_u+1)-th *Add* by p_u is executed. By Observation 22, in that configuration, it also still holds that $BitVector[u] = 0$. Then, the $(w_u + 1)$-th *Add* by p_u set the u-th bit to 1 while leaving all other bits unchanged. Thus, if p_u has executed $(w_u + 1)$ *Add* on $BitVector[1..n]$ by C, where $(w_u + 1) \mod 2 = 1$, then $BitVector[u] = 1$, i.e. $BitVector[u] = (w_u + 1) \mod 2$ at C. Therefore, the claim holds.

The case where $w_u \mod 2 = 0$ is symmetric. □

Let Q^u_w be the w_u-th *Add* of line 41 executed by p_u in α and let CA^u_w be the configuration that results from that. Then, an immediate consequence of Lemma 31 is the following.

Corollary 32. *For each w_u, $0 \leq w_u \leq m_u$, the following claims hold:*

1. *$BitVector[u] = w_u \mod 2$ at CA^u_w;*

2. *$BitVector[u]$ has the same value between $CA^u_{w_u}$ and the configuration in which $Q^u_{w_u+1}$ is executed.*

We proceed to examine the behavior of $ST.ann[1..n]$ and $ST.done[1..n]$.

Inspection of the pseudocode (lines 48, 67) shows that a successful $st - sc$ executed during an `AGREE` phase assigns to $ST.ann[1..n]$ the value of local variable *lbv*. Conversely, a successful $st - sc$ executed during an `APPLY` phase assigns to $ST.done[1..n]$ the value of local variable *lbv* (lines 66, 67). In conjunction with Lemma 25, this leads to the following observation.

Observation 33. *$ST.ann[1..n]$ is only modified by those SC^{ST}_k for which $k \mod 2 = 1$. $ST.done[1..n]$ is only modified by those SC^{ST}_k for which $k \mod 2 = 0$.*

This observation, as well as further inspection of the pseudocode (lines 45, 48, 66, 67) and Lemma 25 imply the following lemma.

Lemma 34. *Let p_l be the process that executes SC^{ST}_k. Let C be the last configuration in which p_l reads BitVector before executing SC^{ST}_k. If $k \mod 2 = 1$, then in C^{ST}_k, $ST.ann[1..n]$ has the value that $BitVector[1..n]$ had at C. If $k \mod 2 = 0$, then in C^{ST}_k, $ST.done[1..n]$ has the value that $BitVector[1..n]$ had at C.*

Linearizability. Recall that an execution interval of α during which *ST.phase* has the value `AGREE` is referred to as `AGREE` *phase*, while an execution interval in which *ST.phase* has the value `APPLY`, is referred to as `APPLY` *phase*. Inspection of the pseudocode shows that if the `LL` of line 44 of some process occurs during an `AGREE` phase, then lines 47 to 50 are executed. This observations, as well as Observation 33 and Lemma 34, indicate that if $ST.ann[u]$, $u \in \{1, 2, \ldots, n\}$, is modified by an SC^{ST}_k, $k > 0$, then this SC^{ST}_k toggles *ST.phase* from `AGREE` to `APPLY`. If such a modification of $ST.ann[u]$ occurs in C^{ST}_k, then we say that some operation by process p_u has been *agreed* in C^{ST}_k.

By similar reasoning, if the LL of line 44 occurs during an APPLY phase, the process executes lines 51 to 66 and toggles the phase from APPLY to AGREE, while also potentially modifying $ST.done[u]$, $u \in \{1, 2, \ldots, n\}$. If such a modification of $ST.done[u]$ occurs in C_k^{ST}, then we say that some operation by process p_u has been *applied* in C_k^{ST}.

Considering that at least one process is crash-free in α, we have the following lemma.

Lemma 35. *Any announced request is agreed at most once during its execution interval.*

Proof. Let req_u be a request by p_u that is announced in some configuration C_u in α. We prove the claim by contradiction.

Assume first that req_u is never agreed in α. Corollaries 26 and 28 imply that this cannot be due to the fact that the phase does not change. Thus, the phases alternate, but by assumption, there is no phase in which req_u is agreed. Let p_l, $1 \leq l \leq n$ be a process that executes SC_k^{ST} for some k, that changes the phase from AGREE to APPLY. Inspection of the pseudocode shows that in order to do so, it executes line 48. So, in C_k^{ST}, $BitVector[u] = ST.ann[u]$. By Lemmas 31 to 34, we have that if an operation is not agreed, then $ST.ann[u] = ST.done[u]$ and $BitVector[u] \neq ST.ann[u]$ – a contradiction. Thus req_u is announced at least once during its execution interval.

Assume now that req_u is agreed at least one more time after C_k^{ST}. By definition of the phases, this can only happen in a configuration C_l^{ST}, $k < l$, that results from some subsequent AGREE phase. By Corollary 26, we have that at least one APPLY phase occurs between C_k^{ST} and C_l^{ST}. Notice that by Lemma 34, at the end of an APPLY phase, it holds that $ST.ann[u] = ST.done[u]$ and those values are equal to $BitVector[u]$. Inspection of the pseudocode (line 46) shows that if this is the case, BTU terminates its execution and returns a response to req_u. By Observation 29, if $BitVector[u] \neq ST.done[u]$ in some subsequent configuration, then this can only hold because p_u has invoked a subsequent request – a contradiction with the assumption that req_u is agreed more than once. Thus, the claim holds. □

By similar reasoning, we have the following.

Lemma 36. *Let req be a request that is agreed in configuration C_k^{ST} in α. Then req is applied at most once during its execution interval, namely in C_{k+1}^{ST} if it is included in α.*

We proceed to examine the modification of edges. Inspection of the pseudocode leads to the following observation.

Observation 37. *The weight and sequence number of an edge $e_{i,j}$ can only be modified by a successful execution of the* SC *operation of line 64 during an* APPLY *phase of* Dense*.*

Let U be an instance of an UpdateEdge operation by p_u that writes v to edge $e_{i,j}$ and that is agreed upon in some configuration C_k^{ST} in α. If after C_k^{ST}, some process p_l successfully executes the SC of line 64 on $e_{i,j}$ with the parameter v of U resulting in configuration C, we say that U *takes effect* in C.

Lemma 38. *For any instance U of an* `UpdateEdge` *operation in α, there is at most one configuration C in α in which U takes effect.*

Proof. We prove the claim by contradiction. Assume that there are two such configurations, C and C' in α and let $C < C'$, without loss of generality. Let U write v on edge $e_{i,j}$. Furthermore, let C be immediately preceded by step sc, an `SC` that is successfully executed by some process p_u on $e_{i,j}$, and let C' be immediately preceded by step sc', also a successful `SC` executed by process p_l on $e_{i,j}$. Denote by ll and ll' the corresponding `LL` on $e_{i,j}$ for sc and sc', respectively. We proceed by case analysis.

First, consider that $sc < ll'$. Since sc is a successful `SC` on $e_{i,j}$, inspection of the pseudocode (lines 58 - 64) implies that the condition of the `if` statement of line 58 evaluates to `true`, i.e. that $st_u.seq$ is greater than $e_{i,j}.seq$. Further inspection of the pseudocode (lines 62 - 64) shows that the value of $e_{i,j}.seq$ in C is the same as that of $st_u.seq$. By Observation 37, C occurs in an `APPLY` phase of `Dense`. Consider first that ll' occurs in the same `APPLY` phase. Inspection of the pseudocode shows that ll' is a step taken when p_l executes line 57. Since ll' occurs in the same `APPLY` phase as C, it must hold that $st_l.seq = st_u.seq$. Since sc' is the successful execution by p_l of line 64, it must hold that the evaluation by p_l of the condition of the `if` statement of line 58 is `true`, i.e. it must hold that in the configuration in which this statement is evaluated, $e_{i,j}.seq$ is less than $st_l.seq$. Inspection of the pseudocode (lines 57 - 58) shows that this configuration follows ll'. Since ll' follows C, and since $st_l.seq = st_u.seq$, the condition of the `if` statement of line 58 is evaluated to `false` by p_l and line 64 is not executed – a contradiction. Thus ll' does not occur in the same `APPLY` phase as C.

Since $sc < ll'$, it must then hold that ll' occurs in a subsequent `APPLY` phase. Let p_k be the process that invokes U. Lemmas 34 and 35 imply that at the end of the `APPLY` phase in which sc takes place, $ST.done[k] = ST.ann[k]$. However, inspection of the pseudocode (lines 53 - 64) show that (given the assumption that it executes ll' and sc') p_l evaluates the condition of the `if` statement of line 53 to `true`, also a contradiction. Therefore, it cannot hold that $sc < ll'$.

Consider now that $ll' < sc$. Since by definition $sc < sc'$, it follows that $ll' < sc < sc'$. This in turn implies that sc' is a successful `SC` on $e_{i,j}$, although sc, i.e. another successful `SC` on $e_{i,j}$, is interposed between ll' and sc'. By the definition of the `LL/SC` primitive, this is a contradiction.

It follows that there can be no more than one configuration in which U takes effect and the claim holds. □

We assign linearization points to instances of `UpdateEdge` and `DynamicTraverse` as follows:

- `UpdateEdge`. Let U be an `UpdateEdge` operation executed by process p_u that writes to edge e_{ij} of the graph. Let U be agreed upon in configuration C_k^{ST}, $k > 0$. Let p_l be the process that executes the first successful `SC` of line 64 on e_{ij} after U has been agreed upon (notice that it is possible that $l = u$). If this step occurs in an iteration of the `for` loop of line 52, in which it holds that $r_l = u$, then, the linearization point $*_U$ of U is placed in the resulting configuration. In that case, we refer to U as a *visible* `UpdateEdge`. Conversely, if the step occurs in an iteration of the `for` loop of line 52, in which it holds that $r_l \neq u$, then $*_U$ point is placed in the configuration just before the step is executed. In that case, we refer to U as an *invisible* `UpdateEdge`. In case several invisible `UpdateEdge` operations have their linearization point in the same configuration, ties are broken based on the ID number of the process.

- `DynamicTraverse`. Let DT be a `DynamicTraverse` operation executed by process p_u and let DT be agreed upon in configuration C_k^{ST}, $k > 0$. The linearization point $*_D$ for DT is placed in configuration C_{k+1}^{ST}, i.e. in the configuration in which DT is applied.

By inspection of the pseudocode (line 25), we see that an instance DT of `DynamicTraverse` also invokes exactly one instance of `BTU`. Thus, the next lemma follows as a direct consequence of Lemma 35 and the definition of the linearization point of `DynamicTraverse`.

Lemma 39. *The linearization point of an instance DT of* `DynamicTraverse` *is included in its execution interval.*

We prove this property also for `UpdateEdge` operations.

Lemma 40. *The linearization point of an instance U of* `UpdateEdge` *is included in its execution interval.*

Proof. Let U be executed by some process p_u, $u \in \{1, 2, \dots, n\}$ and assume that it is invoked to update edge $e_{i,j}$. After U is invoked, it in turn invokes an instance I of `BTU` (line 23). By inspection of the pseudocode, we have that U invokes exactly one instance I of `BTU` and that it terminates only after I returns. We proceed by case analysis.

First, assume that p_u is the process that executes the first successful `SC` of line 64 on $e_{i,j}$, after the configuration C_k^{ST} in which U is announced.

Let U be linearized in the configuration resulting from the first successful execution of the `SC` instruction of line 64 by p_u (while executing I). As this line is executed before I terminates and given that U terminates only after I returns, the claim holds.

Next, let U be linearized in the configuration just prior the configuration resulting from the successful execution of the `SC` instruction of line 64 by some other process p' (while executing an instance I' of `BTU`). By definition, the `SC` instruction executed by I in this case is unsuccessful, i.e. between the execution of the `LL` instruction of line 57 by p and the `SC` instruction of line 64 by p, p' has executed a successful `SC` instruction. Given that lines 57 and 64 are executed by p before I returns, and thus, before U terminates, the claim holds also in this case. □

Consider the `DynamicTraverse` operations that are assigned linearization points in α and let $\mathcal{L}_{\mathcal{DT}}$ be the sequence of those operations, based on the order of their linearization points.

Lemma 41. *Let C be any configuration in α and let $\mathcal{L}_{\mathcal{DT}}^{\mathcal{C}}$ be the prefix of $\mathcal{L}_{\mathcal{DT}}$ that denotes the sequence of operations that are assigned linearization points in the execution interval α' between C_0 and C. Then, the value of $ST.rvals[u]$ at C, for some $u \in \{1, 2, \dots, n\}$, is equal to the value that $ST.seq$ had in the configuration C_k^{ST} in which the last* `DynamicTraverse` *in $\mathcal{L}_{\mathcal{DT}}^{\mathcal{C}}$ by p_u was* *linearized in α'. If no* `DynamicTraverse` *by p_u is linearized in α', then the value of $ST.rvals[u]$ is equal to the initial value.*

Proof. We prove the claim by contradiction. Assume first that there is an instance dt_u of `DynamicTraverse` by p_u that is linearized last in $\mathcal{L}^{\mathcal{C}}_{\mathcal{DT}}$ and let the linearization point be placed in configuration C^{ST}_l, $l \mod 2 = 0$. Assume that SC^{ST}_l assigns value v to $ST.rvals[u]$ and, to arrive at a contradiction, assume that at C, $ST.rvals[u] = v'$, $v' \neq v$.

Inspection of the pseudocode (line 64) shows that $ST.rvals[u]$ can only be modified by those $st - sc$ that toggle $ST.phase$ from `APPLY` to `AGREE`, i.e., by Corollary 26, those SC^{ST}_k where $k \mod 2 = 0$. Since, $ST.rvals[u]$ may only be modified by some SC^{ST}_k such that $k \mod 2 = 0$, this in turn implies that v' is a value assigned by some SC^{ST}_k where $k > l$. By the way linearization points are assigned, an instance of `DynamicTraverse` by p_u that has been agreed upon in C^{ST}_{k-1}, is linearized in C^{ST}_k, $k \mod 2 = 0$. This in turn implies that some instance of `DynamicTraverse` is linearized after dt_u, a contradiction. Thus, at C, $ST.rvals[u] = v$. Notice that by inspection of the pseudocode (line 64, line 67), we have that when $ST.rvals[u]$ is assigned a value, this value corresponds to the value that $ST.seq$ had in the immediately preceding `APPLY` phase. Thus, v is the value that $ST.seq$ has in the configuration in which dt_u is linearized and the claim holds.

The argument for the case in which there is no such dt_u in α' is analogous. □

Consider the `UpdateEdge` operations that are assigned linearization points in α and let $\mathcal{L}_{\mathcal{U}}$ be the sequence of those operations, based on the order of their linearization points. Let $\mathcal{L}_{\mathcal{U}}|e_{i,j}$ be the projection of $\mathcal{L}_{\mathcal{U}}$ on the instances of `UpdateEdge` which modify edge $e_{i,j}$. Let $SC^{e_{i,j}}_1, SC^{e_{i,j}}_2, \ldots$ be the sequence of successful `SC` operations (line 64) on $e_{i,j}$ in α that $\mathcal{L}_{\mathcal{U}}|e_{i,j}$ imposes. Denote by $CB^{e_{i,j}}_k$ the configuration in which $SC^{e_{i,j}}_k$ is executed and denote by $CA^{e_{i,j}}_k$ the resulting configuration.

Lemma 42. *Let C be any configuration in α and let $\mathcal{L}^{\mathcal{C}}_{\mathcal{U}}|e_{i,j}$ be the prefix of $\mathcal{L}_{\mathcal{U}}|e_{i,j}$ that denotes the sequence of operations that are assigned linearization points in the execution interval α' between C_0 and C. Then, the value of $e_{i,j}.seq$ at C is equal to the value that $ST.seq$ had in the configuration in which the last* `UpdateEdge` *in $\mathcal{L}^{\mathcal{C}}_{\mathcal{U}}|e_{i,j}$ was linearized in α'. If there is no such instance* `UpdateEdge`*, then the value of $e_{i,j}.seq$ is equal to the initial value.*

Proof. We prove the claim by contradiction. Assume first that there is an instance $u_{i,j}$ of `UpdateEdge` by some process p_u that writes to $e_{i,j}$ and is the instance of $\mathcal{L}^{\mathcal{C}}_{\mathcal{U}}|e_{i,j}$ that is linearized last in α' and that writes v to $e_{i,j}.seq$, and, to arrive to a contradiction, assume that the value of $e_{i,j}.seq$ at C is $v' \neq v$.

Since the value of $e_{i,j}.seq$ is other than v at C, this means that it was modified in the execution interval between the configuration in which $u_{i,j}$ was linearized and C. By Observation 37, we have that in that interval, a successful `SC` was executed on $e_{i,j}$. Notice that by definition, when this occurs, an instance of an `UpdateEdge` operation on $e_{i,j}$ is applied. Furthermore, by the way linearization points are assigned, if an instance of an `UpdateEdge` operation is applied, it is linearized in the configuration following the SC that applies it. This implies that a further instance of UpdateEdge on $e_{i,j}$ is linearized in the execution interval between the linearization point of $u_{i,j}$ and C. By the definition of $u_{i,j}$, this is a contradiction. Therefore, the value of $e_{i,j}.seq$ at C is the value written by the SC that applies $u_{i,j}$.

Let p_l be the process that applies $u_{i,j}$. Inspection of the pseudocode (lines 63 - 64) shows that a successful SC on $e_{i,j}$ assigns to $e_{i,j}.seq$ the value of $st_l.seq$ and further inspection of the pseudocode (line 44) shows that this is the value that $ST.seq$ has during the APPLY phase in which $u_{i,j}$ is applied. Thus, the claim holds.

The argument for the case where no instance of UpdateEdge on $e_{i,j}$ is linearized in α is analogous. □

Lemma 43. *Let $SC_k^{e_{i,j}}$, $k > 0$, be an* SC *that applies some instance of* UpdateEdge *to $e_{i,j}$ and, for some u, $u \in 1, 2, \ldots, p$, let it hold that $ST.rvals[u] > e_{i,j}.seq$ at $CB_k^{e_{i,j}}$. Then, at $CA_k^{e_{i,j}}$ it holds that $e_{i,j}.prev[u]$ contains that weight-sequence number pair that is written to $e_{i,j}$ by that applied instance U of* UpdateEdge *in $\mathcal{L}_{\mathcal{U}}|e_{i,j}$, which is the last to be linearized before $SC_k^{e_{i,j}}$.*

Proof. We prove the claim by contradiction. Let the SC that applies U write to $e_{i,j}$ the weight-sequence number pair $\langle v, s\rangle$, i.e. in the configuration following this SC, it holds that $e_{i,j}.w = v$ and $e_{i,j}.seq = s$. To arrive at a contradiction, assume that at $CA_k^{e_{i,j}}$, $e_{i,j}.prev[u] = \langle v', seq'\rangle$, where $v' \neq v$ and $s' \neq s$.

By inspection of the pseudocode (lines 59 - 61, lines 57 - 64) we see that $SC_k^{e_{i,j}}$ assigns to $e_{i,j}.prev[u]$ the values that $e_{i,j}.w$ and $e_{i,j}.seq$ had in the configuration in which the LL corresponding to $SC_k^{e_{i,j}}$ was executed.

By Observation 37, we have that $e_{i,j}$ may only be modified by successful executions of SC on it. This means that after the configuration in which U is linearized and before the LL corresponding to $SC_k^{e_{i,j}}$ is executed, a successful SC on $e_{i,j}$ takes place. By the way linearization points are assigned, the configuration resulting from the execution of this SC is a configuration in which some instance of an UpdateEdge operation is linearized – a contradiction to the definition of U. Thus, the claim holds. □

We now proceed to prove that instances of ReadEdge that are invoked by a process during a dynamic traversal, read edge values that are mutually consistent.

Lemma 44. *Consider an instance R of* ReadEdge *with arguments i and j, executed by p_u and let r be the executed by R on line 31. Let DT be the last instance of* DynamicTraverse *executed by p_u before R. Then, R returns as the weight for edge $e_{i,j}$ the value v, which is the weight written to $e_{i,j}$ by U, where U is the last instance of* UpdateEdge *with arguments i, j, v, that was linearized before the linearization point of DT.*

Proof. To arrive at a contradiction, let R return the value v' written by another instance of UpdateEdge, U'. Also, let s be the value of $ST.seq$ in the configuration in which U is applied and let s' be the value of $ST.seq$ in the configuration in which U' is applied. Inspection of the pseudocode (lines 32 - 34) shows that after R executes r, it either assigns to local variable val – which is the value that `ReadEdge` returns – the value of $e_{i,j}.w$ or the value of $e_{i,j}.prev[u].w$. Thus, we distinguish two cases.

Case 1. Consider that R returns the value of $e_{i,j}.w$, i.e. that line 34 is executed. By assumption, this value is v'. Inspection of the pseudocode shows that line 34 is executed only if the condition of the `if` statement of line 32 evaluates to `false`. For this to be the case, it must hold that in the configuration in which r is executed, $e_{i,j}.seq \leq ST.rvals[u]$, i.e. $s' \leq ST.rvals[u]$. By Lemma 42 we have that in that configuration, the value of $e_{i,j}.seq$ is the value that $ST.seq$ had in the configuration in which the last `UpdateEdge` that is applied on $e_{i,j}$ was linearized. By Lemma 41, we also have that $ST.rvals[u]$ has the value that $ST.seq$ had in the configuration in which DT was linearized. Since by Corollary 28 the value of $ST.seq$ only increases in α, it must either hold that $s \leq s'$ or that $s' \leq s$. If $s \leq s'$, then, since it holds that $s' \leq ST.rvals[u]$, it must hold that U' is the last instance of `UpdateEdge` to be linearized before DT, a contradiction. If $s' \leq s$, then it can either hold that $s' \leq s \leq ST.rvals[u]$ or that $s' \leq ST.rvals[u] \leq s$. In case that $s' \leq s \leq ST.rvals[u]$, then in the configuration in which line 34 is executed, $e_{i,j}.seq$ does not have the value of $ST.seq$ in the configuration in which U, i.e. the last instance of `UpdateEdge` on $e_{i,j}$ was executed. By Lemma 42, this is a contradiction. In case that $s' \leq ST.rvals[u] \leq s$, then it must hold that the last instance of `UpdateEdge` that is linearized on $e_{i,j}$ before the linearization point of DT, is linearized in a configuration that had a greater value on $ST.seq$. By Corollary 28 this also is a contradiction.

Case 2. Consider now that R returns the value of $e_{i,j}.prev[u].w$, i.e. that line 33 is executed. By assumption, this value is v'. Inspection of the pseudocode shows that line 33 is executed only if the condition of the `if` statement of line 32 evaluates to `true`. Since this is the case, it holds that in the configuration in which r is executed, $e_{i,j}.seq > ST.rvals[u]$. Let U'' be the last instance of `UpdateEdge` that is applied and linearized on $e_{i,j}$ before this configuration. Lemmas 42 and 41 as well as Corollary 28 imply that DT was linearized in a configuration which precedes the configuration in which U'' was linearized. Lemma 43 implies that in that configuration, $e_{i,j}.prev[u]$ contains the weight-sequence number pair that was written to $e_{i,j}$ by the last instance of `UpdateEdge` that was applied on $e_{i,j}$ and linearized before DT, i.e. it contains $\langle v, s\rangle$ written by U. Since we have assumed that the execution of line 33 finds $\langle v', s'\rangle$ in $e_{i,j}.prev[u]$, this is a contradiction. □

The previous lemma implies the following corollary for all those instances of `ReadEdge` that are executed by some process during a graph traversal.

Corollary 45. *Dynamic traversals provided by* `Dense` *have a linearization point inside their execution interval and return a consistent view.*

The previous lemmas and corollaries support the following theorem[1].

Theorem 46. `Dense` *is a linearizable concurrent graph implementation with dynamic traversals.*

[1] Notice that both in the correctness as well as the progress proofs, we have omitted mention of operation `EndTraverse`, since its function and pseudocode trivially comply with the claimed properties.

3.2.4 Proof of Progress

In this section, we show that operations and dynamic traversals of `Dense` are wait-free.

Let α be an execution of `Dense`. By inspection of the pseudocode, lines 32 - 34, we see that `ReadEdge` has no loops and that it performs two accesses to shared memory locations – namely, a read of ST (line 30) and a read of a graph edge (line 31). Thus, we obtain the following straight-forward lemma.

Lemma 47. *An instance of a* `ReadEdge` *operation of* `Dense` *terminates after* $O(1)$ *steps.*

In a similar straight-forward manner, by applying the previous lemma to an infinite execution, we obtain the following theorem.

Theorem 48. `Dense` *provides wait-free* `ReadEdge` *operations with* $O(1)$ *step complexity.*

Contrary to `ReadEdge`, operations `DynamicTraverse` and `UpdateEdge` consist in an invocation of `BTU`. Therefore, their complexity depends on that of `BTU`.

Lemma 49. *An instance of a* `BTU` *of* `Dense` *terminates after* $O(k)$ *steps.*

Proof. Inspection of the pseudocode, lines 53 - 64, shows that the shared object accesses, namely the executions of `SC` on edges during the execution of an iteration of the `for` loop (line 43) of an instance of `BTU` are at most k, where $1 \leq k \leq n$ is the number of "active processes", i.e., the number of processes that have a pending operation during the execution intervals in which the instance of `BTU` performs an `APPLY` phase. Furthermore, the loop is executed a finite number of times, namely at most 4 times. Inspection of the pseudocode shows that it then returns. Thus, an instance of `BTU` performs $O(k)$ `SC` on shared objects before it returns, proving the claim. □

The previous lemma implies the following corollary.

Corollary 50. `DynamicTraverse` *and* `UpdateEdge` *operations in* `Dense` *have a step complexity of* $O(k)$*, where k is the number of pending operations during an* `APPLY` *phase.*

By Lemma 35, in an infinite execution, each instance of one of those operations is announced inside its execution interval. By Lemmas 38 and 36, we have that they are applied at most once inside their execution interval. We now proceed to prove that they are applied exactly once inside their execution interval.

Lemma 51. *In an infinite execution α of* `Dense`*, for any request req_u operation by process p_u, agreed in configuration C in α, there is exactly one configuration C' inside req_u's execution interval in α, with $C < C'$, in which req_u is applied.*

Proof. Since we have assumed that α is infinite, this means that there is at least one non-faulty process which invokes and executes `Dense` operations. Recall that, by definition, *d-traversals* in `Dense` contain a finite number of instances of `ReadEdge`. Thus, for α to be infinite, it must contain an infinite number of instances of `UpdateEdge` or of `DynamicTraverse`. In either case,

α then contains an infinite number of instances of BTU that are invoked after C, and therefore, an infinite number of $st - sc$ instances. Notice that it is impossible for all instances of $st - sc$ that occur during any Dense phase (either AGREE or APPLY) to fail, since for this to happen, at least one $sc - st$ that is executed during that phase must succeed, leading us to a contradiction. Thus, we infer that after C, α contains an infinite number of successful $st - sc$, meaning an infinite number of SC_k^{ST}.

We now prove the claim by contradiction. Assume that req_u is agreed in configuration C in α, in which by definition $ST.ann[u] \neq ST.done[u]$m but that α does not contain any configuration after C, in which $ST.ann[u] = ST.done[u]$. Since req_u is agreed in C, by the definition of an operation being agreed, this means that C occurs in α right after an SC_k^{ST}, for some $k > 0$, which changes the phase from AGREE to APPLY. For a subsequent $st - sc$ to be successful, its corresponding $st - ll$ must after C in α. By inspection of the pseudocode (lines 65 - 67), we have that this subsequent successful $st - sc$ then writes into $ST.done[1..n]$ the contents that $ST.ann[1...]$ had in C. Since we have assumed that there is no configuration in α after C, such that $ST.ann[u] = ST.done[u]$, this implies that either no process executes an $st - sc$ after C in α, or no further $st - sc$, executed after C by any process, can be successful.

Assuming that no process executes an $st - sc$ after C in α, we arrive at a contradiction, since for this to happen, either all processes execute only ReadEdge operations, contradicting the definition of *d-traversals*, or all processes stop invoking both UpdateEdge and DynamicTraverse operations, contradicting the assumption that α is infinite.

Assume now that processes do execute $st - sc$ after C, but that no further of those $st - sc$, executed after C by any process, is successful. By the definition of LL/SC and configuration C, for this to happen, all $st - ll$ corresponding to the $st - sc$ must occur before C and also, in the last iteration of the for loop of the instance of BTU executed by each of the processes executing the $st - sc$. Since α is infinite, this implies that processes invoke further instances of UpdateEdge or DynamicTraverse after C - a contradiction, since we have assumed that the $st - ll$ for all $st - sc$ that occur after C in α, occur before C.

Thus, the claim holds. □

The previous lemmas, corollaries and observations imply the following theorem.

Theorem 52. Dense *provides wait-free* UpdateEdge *and* DynamicTraverse *operations that have $O(k)$ step complexity, where k is the number of active processes, i.e. processes that have pending operations during an* APPLY *phase.*

3.3 Related Work

The present section aims at providing a clearer image of the context in which WFR-TM and Dense have been elaborated. While STM and concurrent data structure implementation are vast areas of research, for the purposes of this thesis we limit our literature review to publications that cover those aspects that concern our algorithms.

Transactional Memory In WFR-TM, a read-only transaction T_r starts by announcing itself, so that update transactions may become aware of it. In case an update transaction T_w wants to update a t-variable x after the announcement of T_r (and thus probably after T_r has read x), it may only commit after T_r has committed. So, before an update transaction T_w completes, it waits for all read-only transactions that have been initiated and not yet completed at some point of T_w's execution, to commit. For these cases, T_w stores this value in its local write-set and allows T_r to obtain it from there; this behavior, in which read-only transactions read t-variable values from the write-set of some update transaction, is referred to as snooping. We remark that it is not necessary to know in advance whether a transaction is read-only; any transaction is read-only when it is initiated and becomes an update transaction the first time it accesses a t-variable for write. Update transactions employ fine-grained locking for accessing t-variables, so that those of them that do not conflict can commit in parallel; a conflict occurs between two concurrent update transactions when they access the same t-variable and at least one of them writes it.

On the contrary, in current pessimistic TM algorithms [AMS12, MS12], the updaters use a single coarse-grain lock for accessing shared data. This is a design characteristic that allows those algorithms to bypass the well-known theoretical result of [BGK12a], which implies that wait-freedom cannot be achieved by any TM algorithm, since this result implicitly refers to TM algorithms which do not employ coarse-grained locking or extensive helping mechanisms. Popular lock-based TM implementations, which, like WFR-TM use fine-grained locking on each t-variable that they update, include [ST95, DSS06, FFMR10, FFR08]. However, in those algorithms, read-only transactions may be aborted spuriously and thus they are not wait-free.

In [FC11], a multi-version TM algorithm is introduced which supports wait-free read-only transactions by keeping a list for each t-variable, where each value that it has had is recorded; read-only transactions can find values for the t-variables that they read that are mutually consistent. In [PFK10], a property, called multi-version (MV-) permissiveness, is introduced which requires that read-only transactions never abort. MV-permissive TM algorithms that maintain multiple versions of each t-variable are also presented in [PFK10, PBLK11] enhanced with efficient garbage collection for deallocating obsolete versions of t-variables. WFR-TM ensures multi-version permissiveness while being single-version, i.e. it does not maintain multiple versions of t-variables. Thus, WFR-TM is more space efficient in comparison to multi-version algorithms. We remark that in WFR-TM read-only transactions not only never abort, but additionally, they always complete (by committing).

Although WFR-TM does not maintain multiple versions, each update transaction that locks a t-variable x, must also maintain the value that x had before the transaction locked it. Thus, at any given configuration in an execution of WFR-TM, up to two distinct values for x may be maintained. We remark WFR-TM is in accordance with the theoretical result presented in [KR15], which studies the cost of providing wait-freedom for read-only transactions while ensuring that update transactions commit only if they are executed in the absence of concurrency. The result finds that a TM algorithm with those characteristics must keep unbounded values for each t-variable, in case read-only transactions are required to be invisible. Notice however that in WFR-TM, read-only transactions are visible in the announce array.

Attiya and Hillel present in [AH12] PermiSTM, a `TM` algorithm that ensures multi-version permissiveness without actually maintaining multiple versions of t-variables. Instead, transactions that read a t-variable x announce their presence by incrementing a dedicated read-counter linked to x; this is done by repeatedly executing `CAS` until one of these `CAS` primitives succeeds. So, if it executes concurrently with update transactions that read x, a read-only transaction may repeatedly fail to increment the read-counter of x. This means that read-only transactions in [AH12] are obstruction-free; obstruction-freedom does not ensure that a transaction completes unless the thread executing it runs solo for a sufficient number of steps after some point during the transaction's execution. Each read-only transaction in PermiSTM executes, at best, twice as many expensive synchronization primitives (like `CAS`) as the number of t-variables it reads. PermiSTM pays this cost in order to ensure disjoint-access parallelism; roughly speaking, disjoint-access parallelism guarantees that transactions that do not conflict do not interfere with each other by accessing common base objects. It has been proved in [AHM09] that in disjoint-access parallel `TM` implementations with wait-free read-only transactions, a read-only transaction that reads m t-variables has to perform non-trivial operations on at least $m - 1$ base objects; a non-trivial operation may change the status of the object on which it is applied. In WFR-TM, read-only transactions perform only two writes on base objects and no expensive synchronization operations at all. However, WFR-TM is not disjoint-access parallel.

Similarly to WFR-TM, PermiSTM supports parallelism among update transactions; update transactions are executed speculatively and they may abort. In PermiSTM, a write-transaction does not proceed in updating the t-variables until all read-only transactions that are accessing it are committed (after decrementing the read counter of the t-variable). Thus, update transactions writing to a t-variable may face a never-decrementing read-counter for this t-variable, leading them to run forever. WFR-TM avoids this by having update transactions waiting for the completion of only a limited number of read-only transactions.

Snooping into a transaction's write-set in order to read t-variable values has also been used in other algorithms, such as WSTM [FH07] and OSTM [FH07]. However, WFR-TM combines this with a waiting mechanism where update transactions let read-only transactions terminate, in order to guarantee their wait-freedom. Similar waiting techniques, where update transactions may not commit until some read-only transactions that are concurrent with them have committed, have also been used in [AH12, AMS12].

Concurrent Data Structures with Iterators A great body of work on the concurrent implementation of graph algorithms tackles common graph-related issues (e.g. [CKK$^+$08, NP11, PMP12]) and focuses either on parallelizing existing sequential algorithms or on providing concurrency through the use of locks on well-known sequential algorithms. Then, liveness guarantees are rather relaxed, as most of these implementations are blocking. In contrast, we are interested in the graph as a general-purpose, concurrent data structure and are especially concerned with providing wait-freedom and linearizability.

Work on concurrent data structures has been devoted to commonly-used ones, such as queues, stacks, or trees, with the focus on providing interesting progress properties – initially simply by avoiding locks (e.g. [MS96, SLS06]), and recently a step further, by proposing wait-free implementations. Notably, in [KP11], the implementation of a wait-free queue is proposed. It uses an announce array to facilitate helping and builds on the CAS-based lock-free queue implementation of [MS96]. This method is elaborated upon in [KP12] and, together with a "fast path, slow path" methodology [TBKP12], previously used for the implementation of a wait-free linked list out of well-known lock-free design [Har01], proposed as a generalized methodology of designing wait-free concurrent data structures, given a lock-free implementation. Our method is "stand-alone", providing wait-freedom without requiring a lock-free design as base.

Recently, techniques that provide iterators of concurrent data structures have been proposed. An iterator parses a data structure in order to obtain a consistent view. In [PT13], a methodology is proposed for enhancing lock-free or wait-free set-based data structures with a CAS-based implementation of a wait-free iterator. It entails reporting data structure updates to any active snapshot, so that they can be taken into account, depending on the order of linearization. In [PBBO12], update and read operations on a trie are aware of an ongoing iterator, and copy – and thus, effectively rebuild – the parts of the trie that they access, leaving intact the albeit obsolete version that the iterator is parsing. The complexity is divided among updates and reads, while the snapshot occurs in constant time.

We, however, are interested in a partial view of the graph, which, furthermore, is dynamically defined. Thus, we want to avoid the overhead that is induced by iterating over the entire data structure. Arguably, the implementation in [PBBO12] does not induce it, having a constant-time snapshot. However, to achieve that, it must employ either DCSS primitives, or a custom-made, CAS-like software primitive, unlike our method, which simply relies on LL/SC. Moreover, those works take advantage of the structural regularity of the underlying data structure. In contrast, a graph usually has irregular characteristics. Our work is more akin to partial snapshots, such as [AGR08, IR09], because we use an adjacency matrix to represent the graph. However, partial snapshots are more restrictive than our model as they require a priori knowledge of the component subset to be scanned.

The required dynamicity can be provided by using transactional memory to access a graph. Indeed the dynamic traversal provided by our model resembles a read-only transaction. However, efficient TM algorithms commonly rely on locks, while even obstruction-free or non-blocking ones commonly burden reads and updates with the processing overhead necessary for conflict-detection and resolution (cf. with [FIKK15] for a survey on TM algorithmic techniques). We wish to avoid these issues, as well as the commit/abort semantics inherent in TM, but unusual for data structures. The recent impossibility result in [BGK12b] further implies that, even if commit/abort semantics are included in our model, the TM progress property equivalent to wait-freedom cannot be achieved.

Chapter 4

Data Structures for Many-core Architectures without Cache-coherence Support

In this chapter, we present a collection of algorithms that implement distributed data structures, intended to facilitate their use on many-core architectures, i.e. architectures that rely on message-passing for process synchronization. We assume that processes cannot suffer from crash failures and examine two different approaches of data structure (DS) design, which we base on the client-server model. In more detail, we assume that out of the entirety of cores, NS act as *servers*, while the remaining ones act as *clients*. A servers may store parts of the data structure in its memory module or manage the access to the data structure in co-ordination with other servers. However, we assume that the application using the data structure is executed on clients and so, a data structure operation is invoked on a client, which in turn communicates with the appropriate servers in order to carry it out.

The first design approach that we present in this chapter is based on the assumption that (a subset of) the servers implement a *directory* service for the storage of the data structure. We present the *directory-based* designs of a stack and a queue in Section 4.1. The second design approach consists in adopting the use of a token. The token is assigned to one server at a time and this *token server* is in charge of serving client requests for access to the data structure. The token is passed to a subsequent server if the token server storage fills up or empties. We present the token-based designs of a stack, a queue, and an unsorted list in Section 4.2, as well as a sorted list design in Section 4.3. We make brief mention of experimental results that were obtained for some of those data structures, in Section 4.4. Section 4.5 gives an overview on related literature.

Author's contribution. Contents of this chapter are a joint work that has been published as Technical Report in [FKKS15]. The author provided all the proofs of correctness presented in this chapter and contributed to the design of the distributed lists of Sections 4.2.3 and 4.3.

4.1 Design Paradigm I: Directory-based Data Structures

The directory is a data structure that supports the operations `DirInsert`, `DirDelete`, and `DirSearch`. Although the directory can be implemented with several different ways,we employ a simple highly-efficient distributed hash table implementation (also met in [Dev93, Haz, Sha14]) where hash collisions are resolved by using hash chains, called *buckets*. Each server stores a number of buckets. For simplicity, we consider a simple hash function which employs mod and works even if the key is a negative integer. The hash function returns an index which is used to find the server where a request must be sent, as well as the appropriate bucket at this server in which the element resides (or must be stored). Then (to apply the request), a message to this server is sent; the server locally processes the request and responds to the process that initiated it. One of the servers, denoted s_s, acts as the *synchronizer*. Its main function is to assign a unique sequence number k to each element e inserted in the data structure DS; this number serves as the key of e.

4.1.1 The Directory

Algorithm 7 Insert, search and delete operations of a client of the directory.

```
1  boolean DirInsert(int $cid$, Data $data$, int $key$) {
2    $sid$ = hash_function($key$);
3    send($sid$, $\langle \mathsf{INSERT}, data, key, cid \rangle$);
4    $status$ = receive($sid$);
5    return $status$;
   }

6  boolean DirSearch(int $cid$, int $key$) {
7    $sid$ = hash_function($key$);
8    send($sid$, $\langle \mathsf{SEARCH}, \perp, key, cid \rangle$);
9    $status$ = receive($sid$);
10   return $status$;
   }

11 boolean DirDelete(int $cid$, int $key$) {
12   $sid$ = hash_function($key$);
13   send($sid$, $\langle \mathsf{DELETE}, \perp, key, cid \rangle$);
14   $status$ = receive($sid$);
15   return $status$;
   }
```

Algorithm 7 presents pseudocode for the client's side directory operations. Let c be a client that requests the insertion of an element e into the directory by invoking `DirInsert`. In order to determine the hash table server to which the request should be sent, the hash function is applied on the key of e (line 2). The result of this hashing gives the server's id, which is then used in order to send the request to the server (line 3). Such a request may not always be

Algorithm 8 Events triggered in a directory server.

```
16 HashTable $buckets = \varnothing$;

17 a message $\langle op, key, data, cid\rangle$ is received:
18   if ($op$ == INSERT) {
19     $status$ = insert($buckets, key, data$);
20     send($cid, status$);
21   } else if ($op$ == DELETE) {
22     $status$ = delete($buckets, key$);
23     send($cid, status$);
24   } else if ($op$ == SEARCH) {
25     $status$ = search($buckets, key$);
26     send($cid, status$);
     }
```

successful, as it may happen that the server's allocated hash table memory chunk is full. For this reason, `DirInsert` blocks while waiting for a response from the hash table server (line 4), before finishing by returning the response of the hash table server (line 5). Similarly, `DirDelete` (and `DirSearch`) finds the hash value of the key to be deleted (searched, respectively) and sends a request to the appropriate server.

The server locally processes the request and responds to the process that initiated the request. Algorithm 8 presents event-driven pseudocode for the server's side of the directory operations. We consider a standard implementation of hash table functions such as `insert` (line 19), `delete` (line 21), and `search` (line 24). Those implementations return $\perp$ in case the requested operation is not successful.

4.1.2 A Directory-based Stack

To implement a stack, the synchronizer s_s maintains a variable *top_key* which stores the key of the topmost element of the stack at each point in time. A client c sends a PUSH (POP) request to s_s to obtain a key k. When s_s processes such a PUSH (POP) request, *top_key* is incremented (decremented) and sent as k to c. Then, c uses k as the input argument to `DirInsert` (`DirDelete`). We describe the algorithm in more detail below.

4.1.2.1 Algorithm Description

Pseudocode for the client's side *DS* operations is presented in Algorithms 9 and 10. Push and pop operations are carried out by `ClientPush()` and `ClientPop()` respectively. An operation *op* is invoked on a client c by invoking one of these routines. Subsequently, c sends a message to the synchronizer s_s (line 3 in `ClientPush()`, line 9 in `ClientPop()`) and awaits the response.

The synchronizer receives, processes, and responds to clients' messages. The messages have an *op* field that represents the operation to be performed (PUSH or POP), and a *cid* field that uniquely identifies the client, so that the synchronizer can communicate with it. Event-drive

Algorithm 9 Push operation for a client of the directory-based stack.

```
1  void ClientPush(int $cid$, Data $data$)
2    $sid$ = get the synchronizer id
3    send($sid$, $\langle \mathsf{PUSH}, cid \rangle$)
4    $key$ = receive(sid)
5    $status$ = DirInsert($key, data$)
6    return $status$
```

Algorithm 10 Pop operation for a client of the directory-based stack.

```
7  Data ClientPop(int $cid$)
8    $sid$ = get the synchronizer id
9    send($sid$, $\langle \mathsf{POP}, cid \rangle$)
10   $key$ = receive($sid$)
11   if ($key$ == NACK) then
12     $status = \perp$
13   else
14     do
15       $status$ = DirDelete($key$)
16     while ($status == \perp$)
17   return $status$
```

pseudocode for the synchronizer's side handling of operations is presented in Algorithm 11. As mentioned, s_s uses *top_key* in order to assign keys to the stack elements.

More specifically, if *op* is a push operation, s_s increments *top_key* by one and then sends this value to c (lines 20 - 22). After c receives this value (line 4), it has to use it in order to perform the insertion in the directory itself (line 5) by invoking `DirInsert`. Notice that c may do so lazily. The operation terminates after c receives the response of the directory.

Pop operations proceed in a similar fashion. If *op* is a pop operation, then an important difference is that s_s has to handle the case where the stack is empty (lines 24 - 25). This is indicated by the fact that the value of *top_key* equals -1. In that case, s_s responds with a NACK (line 25) to c. When c receives it (line 11), pop terminates, returning $\perp$ (lines 12, 17).

If the stack is not empty, s_s sends the value stored in *top_key* to c (line 27) and decrements it by one afterwards (line 28). After c receives this value (line 10), it has to use it in order to perform the deletion the directory itself by invoking `DirInsert`. However, since clients insert elements into the directory lazily, it may occur that the key that c attempts to remove, has not yet been inserted into the directory. For this reason, `DirDelete`() is invoked repeatedly (line 15) while the required key is not yet in the directory, in which case the directory responds with the value $\perp$ (line 16).

4.1.2.2 Proof of Correctness

Let α be an execution of the directory-based stack implementation. We assign linearization points to push and pop operations in α as follows:

- Let c be a client invoking a push operation *op* with *key* argument k. Let s be the hash table

Algorithm 11 Events triggered in the synchronizer of the directory-based stack.

```
18 int $top\_key = -1$

19 a message $\langle op,\ cid\rangle$ is received:
20   if ($op$ == PUSH) then
21     $top\_key$ + +
22     send($cid$, $top\_key$)
23   else if ($op$ == POP) then
24     if ($top\_key$ == $-1$) then
25       send($cid$, NACK)
26     else
27       send($cid$, $top\_key$)
28       $top\_key$ − −
```

server that is indicated by the hash function for input argument k. The linearization point of *op* is placed in the configuration resulting from the execution of line 5 of Algorithm 7 for *op* by s.

- Let c be a client invoking a pop operation *op*. If line 25 of Algorithm 11 is executed for *op* by s_s, then the linearization point is placed in the resulting configuration. If line 27 of Algorithm 11 is executed by s_s, then we distinguish two cases. Let op' be that push operation, which inserts into the directory the element that *op* removes. If the linearization point of op' occurs before or at the execution of line 27 for *op*, then *op* is linearized in the configuration resulting from the execution of this line. Otherwise, the linearization point of *op* is placed right after the linearization point of op'.

Notice that in the proposed implementation, s_s does not communicate directly with the directory, nor does it receive feedback when a client successfully inserts or deletes an element with a certain key from it. Instead, s_s serves client requests oblivious to the actions of a client after it has sent it a key value. This may lead to the following scenario: a client c_1 invokes a push operation and receives value k as the key from s_s. However, it stalls right after receiving it. A different client c_2 invokes a pop operation and receives value k for the key as well. Since c_1 is stalling and has not performed the insertion in the directory yet, c_2 has to loop and wait for an element with key k to be inserted in the directory. Yet another client c_3 invokes a push operation and once more, receives value k from s_s. If c_3 inserts an element with key k into the directory before c_1 does, then c_2 may remove from the directory an element that was inserted by a push operation that was invoked *after* c_2 invoked its pop operation. In the following section we prove that, given that the operation intervals overlap, this does not violate the linarizability of the stack's operations.

Lemma 53. *The linearization point of a push (pop) operation op is placed within its execution interval.*

Proof. Inspection of the pseudocode easily shows that the claim holds for push operations, as the execution of the line after which the linearization point is placed, takes place after the invocation and before the response of the operation.

Assume now that *op* is a pop operation invoked by client c and assume that *op* removes an element with key k from the directory. We consider two cases. First, assume that the linearization point of *op* is placed in the configuration resulting from the execution of line 25 for *op* by s_s. Inspection of the pseudocode shows that this line is executed by s_s for *op* after s_s receives from c the message that is sent by executing line 9, i.e. after `ClientPop` is invoked. Further inspection shows that c blocks (line 10) until it receives from s_s the message sent on line 25. This means that `ClientPop`, and therefore, *op*, does not respond before line 25 is executed. The above implies that the linearization point of *op* is included in its execution interval. The argument is analogous if we assume that *op* is linearized in the configuration resulting from the execution of line 27, i.e. that c receives a response from s_s because s_s executes line 27.

Let op' be the push operation that inserts the element with key k that *op* removes, in the directory. Let C be the configuration in the last `do-while` loop iteration of lines 14 - 16 executed during *op*, i.e the iteration in which the execution of `DirDelete` does not return $\perp$. Let C' be the configuration resulting from the execution of `DirInsert` on line 5 by op', after which the element with key k is inserted in the directory by op'. Recall that by the way that the linearization points are assigned, the linearization point of op' is placed in C'. Assume next that the linearization point of *op* is also placed in C'. By definition, C' follows the execution of line 27 for *op* by s_s. Following the same argumentation as for the previous case, we have that the execution of that line occurs in the execution interval of *op*, i.e after *op* is invoked. From the definitions of C and C', we further have that C' happens before C, since the element that op' inserts in the directory by using `DirInsert`, is the element that *op* removes from the directory in C. Recall that by the way that the linearization points are assigned, the linearization point of op' is placed in C'. Since C is included in the execution interval of *op* and C' occurs after the execution of line 27 and before C, and given that the linearization point of *op* is in this case also placed in C', it follows that the linearization point for *op* is included in its execution interval.

Thus, the claim holds for all cases. □

Notice that since only s_s executes Algorithm 11, we have the following.

Observation 54. *Instances of Algorithm 11 are executed sequentially, i.e. their execution intervals do not overlap.*

Further inspection of the pseudocode of Algorithm 11 indicates that the value of *top_key* is incremented before an element is inserted into the directory and decremented before one is removed from the directory. This implies the following observation.

Observation 55. *When the value of top_key is equal to* -1 *, then for each positive integer that* s_s *has sent as key to a push operation, there is a pop operation that has been assigned the same integer as key. The value of top_key is greater than* -1 *in any other case.*

Denote by L the sequence of operations (which have been assigned linearization points) in the order determined by their linearization points. Let C_i be the configuration in which the i-th operation op_i of L is linearized. Denote by α_i, the prefix of α which ends with C_i and let L_i be the prefix of L up until the operation that is linearized at C_i. We denote by top_i the value of the local variable *top_key* of s_s at configuration C_i; let $top_0 = -1$. Denote by S_i the sequence of elements in the sequential stack that results if the operations of L_i are applied sequentially to an initially empty stack. Denote by d_i the number of elements in S_i. We associate a sequence number with each stack element such that the elements from the bottommost to the topmost are assigned $1, \ldots, d_i$, respectively. Denote by sl_{d_i} the d_i-th element of S_i. Denote by λ the empty sequence.

Lemma 56. *For each integer $i > 0$, it holds that if op_i is a pop operation, then it returns the value of the field data of $sl_{d_{i-1}}$ if $S_{i-1} \neq \lambda$, or $\perp$ if $S_{i-1} = \lambda$.*

Proof. We prove the claim by induction on i.

Base case. We prove the claim for $i = 1$. Recall that at C_0, since no operation has been linearized, the equivalent sequential stack is empty. Recall also that at C_0 it holds that *top_key* $= -1$. If op_1 is a push operation, the claim holds vacuously. Let then op_1 be a pop operation. We prove that op_1 is linearized in the configuration that results when s_s executes line 25.

Assume by the way of contradiction that op_1 is not linearized in that configuration. Then, by the way linearization points are assigned, s_s does not execute line 25 for op_1. Thus, when s_s evaluates *top_key* on line 24, its value is not equal to -1. By Observation 55, the value of *top_key* is greater than -1. Thus, by the way linearization points are assigned, op_1 is linearized either at the execution of line 27 by s_s or at an even later configuration. By assumption, op_1 is the first operation to be linearized. This means that there is no linearization point for some push operation that is placed in a configuration preceding the execution of line 27 by s_s for op_1. Then, by definition, if op_1 is linearized at a configuration later than this, then it is linearized together with the push operation whose value op_1 returns. Then, however, op_1 is not the first operation to be linearized – a contradiction. Therefore, $S_0 = \lambda$ and op_1 is linearized at the execution of line 25 by s_s.

Hypothesis. Fix any i, $i > 0$ and assume that the claim holds for all C_j, $j \leq i$.

Induction step. We prove that the claim also holds at C_{i+1}. If op_{i+1} is a push operation, the claim holds vacuously. Let then op_{i+1} be a pop operation. We proceed by case analysis.

First, assume that op_{i+1} is linearized in the configuration immediately following the execution of line 25 by s_s. This implies that s_s evaluates the `if` condition of line 24, to `true`. Let ℓ be the number of push operations that s_s has processed up to C_{i+1}. Since *top_key* $= -1$, this means that s_s has processed ℓ or more pop operations up to C_{i+1}. Notice that for each of these pop operations, s_s has executed either line 25 or line 27 before C_{i+1}. Assume that ℓ' of those ℓ push operations are linearized before C_{i+1}. Then, by the way linearization points are assigned, the corresponding pop operations have been linearized before C_{i+1} as well. It follows that at C_{i+1}, S_i is empty and that the claim holds.

Next, assume that op_{i+1} is linearized in the configuration right after the execution of line 27 by s_s. By definition, this means that op_{i+1} removes an element from the directory that has been inserted into the directory by a push operation op_j, $j \leq i$, which has been linearized before the execution of this line, due to the way linearization points are assigned. We distinguish two cases.

First assume that op_i is a push operation and assume that k_i is the value of *top_key* that it has received by s_s, i.e., op_i inserts into the directory an element with key k_i. Since op_{i+1} is linearized in the configuration following the execution of line 27, Lemma 53 implies that op_i is linearized before the execution of line 27 by s_s for op_{i+1} and by Observation 54, we have that at the end of the execution of the instance of Algorithm 11 by s_s for op_i, it holds that $top\_key = k_i$. Inspection of Algorithm 11 shows that a pop operation that follows a push operation receives the same value of *top_key* as the one that was sent to the push operation. Therefore, if no further instance of Algorithm 11 is executed for some other operation by s_s after it executes it for op_i and before it executes it for op_{i+1}, then the claim follows straight-forwardly. Assume now that between C_i and C_{i+1}, more instances of Algorithm 11 are executed by s_s for other operations. Let op' be that out of those operations for which Algorithm 11 is executed last before C_{i+1} and assume that it is a push. Let k' be the value of *top_key* at the end of this instance of Algorithm 11. Then, at C_{i+1}, s_s sends k' to the client that invoked op_{i+1}. Then this client attempts to remove from the directory an element with key k'. However, since there is no further operation linearized between C_i and C_{i+1}, this element is not in the directory at C_{i+1}. Thus, the push operation that inserts in the directory the value which op_{i+1} removes, is linearized after C_{i+1} – a contradiction to the definition of linearization points. If op' is a pop operation and it receives k' as the value of *top_key* from s_s, then op_{i+1} receives $k' - 1$ as value of *top_key*. Then, op_{i+1} attempts to remove from the directory an element with key $k' - 1$. Let op'' be the push operation that inserts an element with this key. If op'' is linearized after C_{i+1}, once more we arrive at a contradiction. If op'' is linearized before C_{i+1}, then by the induction hypothesis, implies that each of the pop operations between C_i and C_{i+1} removes the top-most element of the sequential stack. Thus, at C_{i+1}, the element inserted by op'' is the top-most one and the claim holds.

Finally, assume that op_{i+1} is linearized right after the linearization point of that push operation op' whose value it removes from the directory. In this case, since no further operation is linearized between op_{i+1} and op', this means that the value inserted by op' is indeed the top-most of S_i when it is removed by op_{i+1} and the claim holds. □

From the above lemmas we have the following.

Theorem 57. *The directory-based distributed stack implementation is linearizable.*

4.1.3 A Directory-based Queue

The directory-based distributed queue implementation follows similar ideas as those of the directory-based stack implementation of Section 4.1.2. To implement a queue, s_s maintains two counters, *head_key* and *tail_key*, which store the key associated with the first and the last,

Algorithm 12 Enqueue operation for a client of the directory-based queue.

```
29 void ClientEnqueue(int $cid$, Data $data$)
30   $sid$ = get the server id
31   send($sid$, $\langle \mathsf{ENQ}, cid \rangle$)
32   $tail\_key$ = receive($sid$)
33   DirInsert($tail\_key, data$)
```

respectively, element in the queue. A client c sends an enqueue (dequeue) request to s_s to obtain a key k. Then, it uses k as the input argument to `DirInsert` (`DirDelete`). When s_s receives an enqueue (dequeue) request from c, it sends the value stored in *tail_key* (*head_key*) to c and increments *tail_key* (*head_key*). In case of a dequeue request on an empty queue (i.e. if *head_key* = *tail_key*), s_s sends NACK to c without changing *head_key*.

4.1.3.1 Algorithm Description

Pseudocode for the client's side *DS* operations is presented in Algorithms 12 and 13. Enqueue and dequeue operations are carried out by `ClientEnqueue()` and `ClientDequeue()`, respectively. `ClientEnqueue()`, performs similar steps as those presented in Algorithm 9 of the directory-based stack: An operation *op* is invoked on a client c by invoking one of these routines. Subsequently, c sends a message to s_s (line 31 of `ClientEnqueue()`, line 36 of `ClientDequeue()`) and awaits the response.

The synchronizer receives, processes and responds to clients' messages. The messages correspond to enqueue and dequeue requests. Message fields are similar as in the case of the stack of Section 4.1.2. Event-driven pseudocode for s_s is presented in Algorithm 14.

More specifically, if *op* is an enqueue operation, then s_s receives an ENQ message (line 46) and sends to c a message containing the current value of *tail_key*. Then it increments *tail_key* by one (line 48). After c receives this value (line 32), it calls `DirInsert` to insert the new element in the directory (line 47). As in the case of the stack, it may do so lazily. The operation terminates after c receives the directory's response.

If *op* is a dequeue operation, it proceeds in similar fashion, with the difference being that the case of the empty queue must be taken into account. This is indicated by the fact that the values of *head_key* and *tail_key* are equal. So, when a DEQ message is received, s_s first checks if the values of *head_key* and *tail_key* are the same (line 50) and if they are, then it responds to c with a NACK message (line 51). Otherwise, it sends the current value of *head_key* to c (line 53) and then increments its value by one (line 54).

If c receives a NACK response from s_s (line 38), then the queue is empty and the operation returns $\perp$. Otherwise, c uses the value of *head_key* that it has received as the key of the element to remove from the directory (line 41). As in the case of the directory-based stack, `DirDelete` is invoked repeatedly while it returns $\perp$, meaning that the insertion of the key to be deleted is still pending. The operation terminates when `DirDelete`() returns a value different than $\perp$, which is the data associated with *head_key*.

Algorithm 13 Dequeue operation for a client of the directory-based queue.

```
34 Data ClientDequeue(int $cid$)
35   $sid$ = get the server id
36   send($sid$, $\langle$DEQ, $cid\rangle$)
37   $head\_key$ = receive($sid$)
38   if($head\_key$ == NACK) then
39     return $\perp$
40   do
41     $status$ = DirDelete($head\_key$)
42   while ($status == \perp$)
43   return $status$
```

Algorithm 14 Events triggered in the synchronizer of the directory-based queue.

```
44 int $head\_key = 0, tail\_key = 0$

45 a message $\langle op, cid\rangle$ is received:
46   if ($op$ == ENQ) then
47     send($cid, tail\_key$)
48     $tail\_key$++
49   else if ($op$ == DEQ) then
50     if ($head\_key == tail\_key$) then
51       send($cid$, NACK)
52     else
53       send($cid, head\_key$)
54       $head\_key$++
```

4.1.3.2 Proof of Correctness

Let α be an execution of the directory-based queue implementation. We assign linearization points to enqueue operations to which the synchronizer has sent a key as a response to their message in α. Then, the linearization point of an enqueue operation *op* is placed in the configuration resulting from the execution of line 47 for *op* by s_s. Similarly, we assign linearization points to dequeue operations to which the synchronizer has sent a key or NACK as a response to their message in α. Then, the linearization point of a dequeue operation *op* is placed in the configuration resulting from the execution of either line 51 or line 53 for *op* (whichever is executed) by s_s.

Lemma 58. *The linearization point of an enqueue (dequeue) operation op is placed within its execution interval.*

Proof. Assume that *op* is an enqueue operation and let c be the client that invokes it. After the invocation of *op*, c sends a message to s_s (line 32) and awaits a response from it. Recall that routine `receive()` (line 32) blocks until a message is received. The linearization point of *op* is placed at the configuration resulting from the execution of line 47 for *op* by s_s. This line is executed after the request by c is received, i.e. after c invokes `ClientEnqueue`. Furthermore, it is executed before c receives the response by the synchronizer and thus, before `ClientEnqueue` returns. Therefore, the linearization point is placed in the execution interval of enqueue.

The argumentation regarding dequeue operations is similar. □

Denote by L the sequence of operations which have been assigned linearization points in α in the order determined by their linearization points. Let C_i be the configuration in which the i-th operation op_i of L is linearized; denote by C_0 the initial configuration. Denote by α_i, the prefix of α which ends with C_i and let L_i be the prefix of L up until (and including) the operation that is linearized at C_i. We denote by $head_i$ the value of the local variable *head_key* of s_s at configuration C_i, and by $tail_i$ the value of the local variable *tail_key* of s_s at C_i. By the pseudocode, we have that the initial values of *tail_key* and *head_key* are 0; therefore, we consider that $head_0 = tail_0 = 0$.

Let L_e be the subsequence of L that contains all enqueue operations in L, excluding all dequeue operations. For each operation in L_e, we define an equivalent enqueue operation e_j, $j > 0$, such that e_j corresponds to the j-th enqueue operation in L_e, and such that e_j enqueues a pair $\langle key, data\rangle$ to a sequential queue, such that $data$ is the argument of the j-th enqueue operation in L_e and that $key = j - 1$. Denote by L' the sequence of operations that results when each enqueue operation in L is replaced by the corresponding e_j. Similarly, denote by L'_i the sequence of operations that results if all enqueue operations in L_i are substituted by the corresponding e_j. Denote by Q_i the sequence of elements in the sequential queue that results if the operations of L'_i are applied sequentially to an initially empty queue. Denote by d_i the number of elements in Q_i. Denote by sl_i^j the j-th element of Q_i, $1 \leq j \leq d_i$.

Consider a sequence of elements S. If e is the first element of S, we denote by $S \setminus e$ the suffix of S that results by removing only element e from the first position of S. If e is an element not included in S, we denote by $S' = S \cdot e$ the sequence that results by appending element e to the end of S.

As the execution interval of an instance of an algorithm executed in α, we consider that subsequence of α that starts with the configuration right after which the algorithm instance takes its first step and ends with the configuration resulting from the last step of the algorithm instance in α. Notice that since only s_s executes Algorithm 14, we have the following.

Observation 59. *Instances of Algorithm 14 are executed sequentially, i.e. their execution intervals do not overlap.*

By inspection of Algorithm 14, we have that for some instance of it, either lines 46-48, or lines 50-51, or lines 52-54 are executed. Then, by the definition of C_i, by the way linearization points are assigned, and by Observation 59, we have the following.

Observation 60. *Given two configurations C_i, C_{i+1}, $i \geq 0$, in α, there is at most one step in the execution interval between C_i and C_{i+1} that modifies either head_key or tail_key.*

By further inspection of the pseudocode, we have that each enqueue or dequeue operation sends one single request to s_s (line 31, line 36). Inspection of the pseudocode executed by s_s shows that when it serves an enqueue request, it only modifies *tail_key* (lines 46-48). Similarly, when s_s serves a dequeue request, it only modifies *head_key* (lines 49-54). So we have the following observation.

Observation 61. *A dequeue operation does not cause s_s to modify head_key. An enqueue operation does not cause s_s to modify tail_key.*

Lemma 62. *For each integer $i \geq 1$, the following hold at C_i:*

1. *If $i > 1$ and op_{i-1} is an enqueue operation, then $tail_i = tail_{i-1} + 1$ and $head_i = head_{i-1}$; if $i = 1$, then $tail_i = tail_{i-1}$.*
2. *If $i > 1$, $head_{i-1} \neq tail_{i-1}$ and op_{i-1} is a dequeue operation, then $head_i = head_{i-1} + 1$ and $tail_i = tail_{i-1}$; if $i = 1$, then $head_i = head_{i-1}$.*

Proof. Fix any $i \geq 1$. The linearization point of op_i may be placed at the configuration resulting from the execution of line 47, line 51 or line 53, whichever is executed by s_s for it. By inspection of the pseudocode, we have that in either case, the execution of neither of these lines, nor the ones preceding it in the instance of Algorithm 14 executed for op_i, modify *tail_key* or *head_key*. Notice also that because of Observation 59 no process other than s_s modifies neither *tail_key* nor *head_key* between C_{i-1} and C_i.

We proceed by case analysis. First, consider the case where $i = 1$. Recall that $tail_0 = head_0 = 0$. Because of the preceding argument, $tail_1 = tail_0 = 0$ and $head_1 = head_0 = 0$. Thus, the claims hold.

Next, consider the case where $i > 1$. Let op_{i-1} be an enqueue operation. By the pseudocode (line 48), *tail_key* is incremented after the linearization point of op_{i-1}, i.e. between configurations C_{i-1} and C_i. Thus, $tail_i = tail_{i-1} + 1$. The value of *head_key* is not modified by enqueue operations (lines 46-48), therefore $head_i = head_{i-1}$.

Now let op_{i-1} be a dequeue operation that is linearized at the execution of line 51. By inspection of the pseudocode (line 50), this occurs only in case $head_{i-1} = tail_{i-1}$. By the pseudocode (lines 50-51) and by Observation 59, it follows that in this case *head_key* is not modified in the execution interval between C_{i-1} and C_i. Therefore, $head_i = head_{i-1}$. Since, by Observation 61, a dequeue operation does not modify *tail_key*, it also holds that $tail_i = tail_{i-1}$.

Finally, let op_{i-1} be a dequeue operation that is linearized at the execution of line 53. By the pseudocode, line 54 and by Observation 59, *head_key* is incremented by 1 after the linearization point of op_{i-1}, i.e. between configurations C_{i-1} and C_i. Thus, $head_i = head_{i-1} + 1$. The value of *tail_key* is not modified by dequeue operations (lines 50-54), therefore $tail_i = tail_{i-1}$. □

The previous lemma implies the following corollary.

Corollary 63. *Let op_i be a dequeue operation in L. Let op_j, $j < i$, be the last dequeue operation that precedes op_i in L. Then, $head_i = head_j + 1$. If no such op_j exists, then $head_i = head_0$.*

Let op_i be an enqueue operation in L. Let op_j, $j < i$, be the last enqueue operation that precedes op_i in L. Then, $tail_i = tail_j + 1$. If no such op_j exists, then $tail_i = tail_0$.

We denote the *key* field of the $\langle key, data \rangle$ pair that comprises some element sl_i^j, $0 < j \leq d_i$, of Q_i by $sl_i^j.key$. By the way Q_i is defined, we have that if sl_i^j has been enqueued by the

ℓ-th enqueue operation in L'_i, then $sl_i^j.key = \ell - 1$. By Corollary 63, we have the following observation.

Observation 64. *If L_i contains ℓ enqueue operations, and if op_i is an enqueue operation, then $tail_i = \ell - 1$.*

By inspection of the pseudocode (lines 46-48), we see that, when op_i is an enqueue operation, $tail_i$ is sent by s_s to the client c that invoked op_i. By further inspection of the pseudocode (lines 32-33), we see that c uses $tail_i$ as the *key* field of the element it enqueues. When op_i is a dequeue operation, by inspection of the pseudocode (lines 50-51), we have that when $head\_key = tail\_key$, s_s sends NACK to c, and that when c receives NACK, it does not enqueue any element and instead, returns $\perp$(lines 38-39). When $head\_key \neq tail\_key$, s_s sends $head_i$ to c (lines 52-54) and c uses $head_i$ as the *key* field in order to determine which element to dequeue (lines 41-43).

Observation 65. *Let c be the client that invoked op_i. If op_i is an enqueue operation, then c initiates the insertion of a pair with key $= tail_i$ into the directory. If op_i is a dequeue operation then, if $head_i \neq tail_i$, c initiates the removal of a pair with key $= head_i$ from the directory; if $head_i = tail_i$, it does not initiate the removal of any pair from the directory.*

Lemma 66. *At C_i, $i \geq 1$, the following hold:*

1. *If op_i is an enqueue operation, then $tail_i = sl_i^{d_i}.key$.*
2. *If op_i is a dequeue operation, then if $Q_{i-1} \neq \lambda$, $head_i = sl_{i-1}^1.key$. If $Q_{i-1} = \lambda$, then $head_i = tail_i$.*

Proof. We prove the claims by induction.

Base case. We prove the claim for $i = 1$. Consider the case where op_1 is an enqueue operation. Then, by Lemma 62 and since $tail_0 = 0$, it follows that $d_1 = 1$ and Q_1 contains only one element, namely $sl_1^{d_1} = \langle 0, data \rangle$. By Observation 59, it is the first operation in α for which an instance of Algorithm 14 is executed by s_s. Therefore, by Lemma 62, $tail_1 = tail_0 = 0$. Thus, $tail_1 = sl_1^{d_1}.key$

Now consider the case where op_1 is a dequeue operation. By Observation 59, op_1 is the first operation in α for which an instance of Algorithm 14 is executed by s_s. Notice that then, $Q_1 = \lambda$, given that Q_i is defined as the sequence of elements that results if operations are applied to an initially empty queue. Therefore, by Lemma 62, $head_1 = head_0 = 0$. By the same reasoning, $tail_1 = tail_0 = 0$. Thus, $head_1 = tail_1$, so Claim 2 holds.

Hypothesis. Fix any i, $i > 0$ and assume that the lemma holds at C_i.

Induction step. We prove that the claims also hold at C_{i+1}. Assume that op_{i+1} is an enqueue operation. Then, its corresponding enqueue operation in L'_{i+1} is also an enqueue operation, and thus, $d_{i+1} = d_i + 1$. We examine two cases. First, consider that op_i is an enqueue operation as well. Since $i > 0$, it holds that $i + 1 > 1$. By Lemma 62, we have that $tail_{i+1} = tail_i + 1$. By Observation 65, we have that the client c that initiated op_{i+1} inserts a pair with $key = tail_{i+1} = tail_i + 1$ into the directory. By definition, and by the semantics of the sequential queue, $sl_{i+1}^{d_{i+1}}.key = sl_i^{d_i}.key + 1$. By the induction hypothesis, $sl_i^{d_i}.key = tail_i$. Thus, $sl_{i+1}^{d_{i+1}}.key = tail_i + 1$, and Claim 1 holds.

Next, consider that op_i is a dequeue operation. By Lemma 62 and Observation 61, dequeue operations do not modify *tail_key*, and by Corollary 63, $tail_{i+1} = tail_j + 1$, where op_j is the last enqueue operation preceding op_{i+1} in L_{i+1}. By definition of Q_j and by Observation 65, op_j enqueues a pair with $key = tail_j$ to Q_j. Furthermore, by definition of op_j, all other operations in L_{i+1} that have a linearization point between that of op_j and op_{i+1}, are dequeue operations. By definition, the same holds for L'_{i+1}. Therefore, no further element is appended to the sequential queue by operations that are linearized between C_j and C_{i+1}, i.e. $sl_j^{d_j} = sl_i^{d_i}$. Notice that, by Observation 64 and the definition of the semantics of the sequential queue, $sl_j^{d_j}.key = tail_j$. By Observation 65, c inserts a pair with $key = tail_{i+1}$ into the directory. Also, by the definition of L'_{i+1}, it holds that $sl_{i+1}^{d_{i+1}}.key = sl_i^{d_i}.key + 1$. Thus, since $tail_{i+1} = tail_j + 1$, it follows that $sl_{i+1}^{d_{i+1}}.key = tail_j + 1 = sl_j^{d_j}.key + 1 = sl_i^{d_i}.key + 1$, and Claim 1 holds.

Now let op_{i+1} be a dequeue operation. Again we examine two cases. First, consider that op_i is a dequeue operation as well. Assume that $Q_{i-1} = \lambda$. Then, since the induction hypothesis holds at C_i, it holds that $tail_i = head_i$. By Observation 61, the value of *tail_key* is not modified neither by op_i nor by op_{i+1}. By Lemma 62, *head_tail* is not modified either. Therefore, also at C_{i+1}, we have that $head_{i+1} = tail_{i+1}$, so Claim 2 holds. Assume now that $Q_{i-1} \neq \lambda$. By the induction hypothesis, Claim 2 holds at C_i, which implies that when op_i is applied to the sequential queue to obtain Q_i, it dequeues an element with key $sl_{i-1}^1.key = head_i$. By the definition of Q_i and of L'_i, the keys of the elements in the key-data pair sequence that is Q_i, take consecutive values as well. This implies that, since $sl_{i-1}^1.key = head_i$, it must hold that $sl_i^1.key = sl_{i-1}^1.key + 1 = head_i + 1$. By Lemma 62, $head_{i+1} = head_i + 1$. Thus, op_{i+1} removes from the sequential queue the element with key equal to $head_i + 1$. Since Claims 1 and 2 hold at C_i by the induction hypothesis, we have that this element is sl_i^1, i.e. Claim 2 also holds at C_{i+1}.

Next consider that op_i is an enqueue operation. By Lemma 62 and Observation 61, enqueue operations do not modify *head_key*. By Corollary 63, we further have that $head_{i+1} = head_j + 1$, where op_j is the last dequeue operation preceding op_{i+1} in L_{i+1}. By definition and by Observation 65, op_j dequeues from the sequential queue Q_{j-1} a pair with $key = tail_j$. Furthermore, by definition of op_j, all other operations in L_{i+1} that have a linearization point between that of op_j and op_{i+1}, are enqueue operations. By definition, the same holds for L'_{i+1}. Therefore, no further element is removed from the sequential queue between C_j and C_{i+1}, i.e. $sl_j^1 = sl_i^1$. Notice that $sl_j^1.key = head_j$. By Observation 65, the client c that invoked op_{i+1} removes a pair with $key = head_{i+1}$ from the directory and by definition, $sl_{i+1}^1.key = sl_i^1.key + 1$. Thus, since $head_{i+1} = head_j + 1$, it follows that $sl_{i+1}^1.key = head_j + 1 = sl_j^1.key + 1 = sl_i^1.key + 1$, and Claim 2 holds. □

By Lemma 62 and by inspection of the pseudocode, we have that at C_i, $i > 0$, the value of *tail_key* indicates the number of enqueue operations on Q_i that have been linearized in α_i, and the value of *head_key* indicates the number of successful dequeue operations (i.e. dequeue operations that do not return $\perp$) on Q_i that have been linearized in α_i. Thus, the following corollary holds.

Corollary 67. *$Q_i = \lambda$ if and only if $head_i = tail_i$.*

Lemma 68. *If op_i is a dequeue operation, then it returns the value of the field data of sl^1_{i-1} or $\perp$ if $Q_{i-1} = \lambda$.*

Proof. Consider the case where $Q_{i-1} \neq \lambda$. By definition of Q_i, we have that $Q_i = Q_{i-1} \setminus \{sl^1_{i-1}\}$. Let op_j be the enqueue operation that is linearized before op_i and inserts an element with key $head_i$ to the queue. Notice by the pseudocode, line 41, that the parameter of `DirDelete` is $head_i$. By the semantics of `DirDelete`, if at the point that the instance of `DirDelete` is executed in the `do - while` loop of lines 41-43 for op_i, the instance of `DirInsert` of op_j has not yet returned, then `DirDelete` returns $\langle \perp, - \rangle$.

By Lemma 66, and since *head_key* is not modified by the execution of line 53 by the server, $head_i$ is the *key* of the first pair sl^1_{i-1} in Q_{i-1}. Therefore, when `DirDelete` returns a $status \neq \perp$, it holds that it returns the *data* field of sl^1_{i-1}, the first element in Q_{i-1}, as the return value of op_i, i.e. the claim holds.

Now consider the case where $Q_{i-1} = \lambda$. Since, by Corollary 67, when this is the case, $head_i = tail_i$, NACK is sent to the client that invoked op_i and, by inspection of the pseudocode, op_i returns $\perp$, i.e. the claim holds. □

From the above lemmas we have the following:

Theorem 69. *The directory-based queue implementation is linearizable.*

4.2 Design Paradigm II: Token-based Data Structures

We assume that the servers are numbered from 0 to NS -1 and form a logical ring. Each server has allocated a chunk of memory (e.g. one or a few pages) of a predetermined size, where it stores elements of the implemented *DS*. A *DS* implementation employs (at least) one token which identifies the server s_t, called the token server, at the memory chunk of which newly inserted elements are stored. (A second token is needed in cases of queues and deques.) When the chunk of memory allocated by the token server becomes full, the token server gives up its role and appoints another (e.g. the next) server as the new token server. A client remembers the server that served its last request and submits the next request it initiates to that server; so, each response to a client contains the id of the server that served the client's request. Servers that do not have the token for handling a request, forward the request to subsequent servers; this is done until the request reaches the appropriate token server. A server allocates a new (additional) chunk of memory every time the token reaches it (after having completed one more round of the ring) and gives up the token when this chunk becomes full.

Algorithm 15 Push operation for a client of the token-based stack.

```
1   $sid = 0$   // the client stores the id of the first server with id=0.

2   Data ClientPush(int $cid$, Data $data$)
3     send($sid$, $\langle \mathsf{PUSH}, data, cid, \perp \rangle$)
4     $\langle status, sid \rangle$ = receive()
5     return $status$
```

Algorithm 16 Pop operation for a client of the token-based stack.

```
6   $sid = 0$   // the client stores the id of the first server with id=0.

7   Data ClientPop(int $cid$)
8     send($sid$, $\langle \mathsf{POP}, \perp, cid, \perp \rangle$)
9     $\langle status, sid \rangle$ = receive()
10    return $status$
```

4.2.1 A Token-based Stack

To implement a stack, each server uses its allocated memory chunk to maintain a local stack, *lstack*. Initially, s_t is the server with id 0. To perform a push (or pop), a client c sends a push (or pop) request to the server that has served c's last request (or, initially, to server 0) and awaits for a response. If this server is not the current token server at the time that it receives the request, it forwards the request to its next or previous server, depending on whether its local stack is full or empty, respectively. This is repeated until the request reaches the server s_t that has the token which pushes the new element onto its local stack and sends an ACK to c. If s_t's local stack does not have free space to accommodate the new element, it sends the push request of c, together with an indication that it gives up its token, to the next server. A pop request is treated by s_t in a similar way.

4.2.1.1 Algorithm Description

Pseudocode for the client's side DS operations is presented in Algorithms 15 and 16. Push and pop operations are carried out by `ClientPush()` and `ClientPop()`, respectively. An operation op is invoked on a client c by invoking one of these routines. Subsequently, c sends a message to the token server s_t (line 3 in `ClientPush()`, line 8 in `ClientPop()`) and awaits the response (lines 4 and 9, respectively).

The token server receives, processes, and responds to clients' messages. As with previous data structures, a field in each message indicates the type of operation that is requested. Event-driven pseudocode for the server is presented in Algorithm 17. Initially, the stack elements are stored in the memory space allocated by server s_0, the first server in the ring. At this point, s_0 is the token server, managing the top of the stack. Once the memory chunk of the token server becomes full, the token server notifies the next server in the ring to become the new token server

(s_0 notifies s_1, s_1 does so with s_2, etc, while $s_{\mathsf{NS}-1}$ notifies s_1).

Algorithm 17 Events triggered in a server of the token-based stack.

```
11 LocalStack $lstack = \varnothing$
12 int $my\_sid$ // each server has a unique id
13 int $token = 0$

14 a message $\langle op, data, id, tk \rangle$ is received:
15   switch ($op$)
16     case PUSH:
17       if ($tk ==$ TOKEN) then $token = my\_sid$
18       if ($token \neq my\_sid$) then
19         send($token, \langle op, data, id, tk \rangle$)
20         break;
21       if (!IsFull($lstack$)) then
22         push($lstack, data$)
23         send($id, \langle$ACK$, my\_sid \rangle$)
24       else if ($my\_sid \neq$ NS-1) then
25         $token$ = find_next_server($my\_sid$)
26         send($token, \langle op, data, id,$ TOKEN$\rangle$)
27       else // It's the last server in the order, thus the stack is full
28         send($id, \langle$NACK$, my\_sid \rangle$)
29       break
30     case POP:
31       if ($tk ==$ TOKEN) then $token = my\_sid$
32       if ($token \neq my\_sid$) then
33         send($token, \langle op, data, id, tk \rangle$)
34         break
35       if (!IsEmpty($lstack$)) then
36         $data$ = pop($lstack$)
37         send($id, \langle data, my\_sid \rangle$)
38       else if ($my\_sid \neq 0$) then
39         $token$ = find_previous_server($my\_sid$)
40         send($token, \langle op, data, id,$ TOKEN$\rangle$)
41       else // It's the first server in the order, thus the stack is empty
42         send($id, \langle$NACK$, my\_sid \rangle$)
43       break
```

Each server s_i, $0 \leq i < \mathsf{NS}$, maintains a local variable *token* which identifies whether s_i is the token server. We assume that a local stack implementation, *lstack*, is available for each server. Depending on whether s_i is the token server or not, the messages that it receives are treated accordingly. Each message has four fields: (1) *op* designates the requested operation, (2) *data* contains the data in to be enqueued if $op = \mathsf{ENQ}$ and $\perp$ otherwise, (3) *id* contains the id of the sender, and (4) *tk* is a one-bit flag which is set to TOKEN only when the server has received a forwarded message from another server that also requests a token transition.

Let s_i receive a message. If the message *op* field is PUSH (line 16), then s_i first checks whether the message contains a token transition. This is indicated by tk = TOKEN. If s_i detects this condition, it changes its *token* variable to contain its own id (line 17). If s_i is not a token server, however, it just forwards the message to the next server in the ring (line 19). If s_i is the token server, it checks whether it can perform the push on its local stack (line 21). If this is possible, then s_i responds with ACK to c, the client that initiated the push request. In this implementation, the `push()` function (line 22) does not need to return any value, since the check for memory space has already been performed by the server on line 21, hence `push()` is always successful.

If the local stack of s_i does not have any free space, then s_i must forward c's request to the next server in the ring and also notify it that it must become the token server. More specifically, if $i \neq \mathsf{NS}-1$ (line 24), then s_i sets the tk field of the message to the value TOKEN and forwards the message to s_{i+1} (line 26). On the other hand, if $i = \mathsf{NS} - 1$, then this implies that all other servers in the ring have no memory space available for storing the stack element. In this case the token-based stack is considered full and s_i notifies c by sending a NACK message (line 28).

If s_i receives a message where the op field is POP (line 30), then similar actions take place: s_i checks whether the message contains a token transition and if this is true, then it changes its local variable *token* appropriately. If s_i is not the token server (line 32), then it forwards the message (line 33). If s_i is the token server, however, then it checks whether its local stack is empty (line 35) and if it is not, then s_i can execute the requested pop operation and send the data of the top element to c (line 37). In case s_i has an empty local stack, if $i \neq 0$ (line 38), then s_i it forwards the pop request to s_{i-1}, after setting the tk field to TOKEN (line 40). If $i = 0$, then this implies that the local stacks of all servers are empty and, consequently, the distributed stack is empty. So, s_i responds with NACK to c (line 42).

When c receives a response from s_i, it updates the value of variable *sid* (line 4 of `ClientPush()`, line 9 of `ClientPop()`). This variable represents the id of the server that c considers as token server. Initially, all clients forward their requests to s_0. However, as the server that maintains the top element might change throughout an execution, the clients have to update the value of *sid* and do so through the aforementioned lazy mechanism. In the meanwhile, if c's message was sent to an incorrect server, it is forwarded by the servers till it reaches the server that holds the token. Since that server is going to respond to c after performing the requested operation, c can update the value of *sid*. `ClientPush()` and `ClientPop()` then return the value of variable *status*. This value is either ACK indicating a successful push or pop, or NACK, indicating that the stack is full, in case of a push, and that it is empty, in case of a pop.

4.2.1.2 Proof of Correctness

Let α be an execution of the token-based stack algorithm presented in Algorithms 15, 16, and 17. Let op be any operation in α. We assign a linearization point to op by considering the following cases:

- op is a push operation. Let s_t be the token server that responds to the client that initiated op (i.e. the `receive` of line 4 in the execution of op receives a message from s_t). If op

returns ACK, the linearization point is placed at the configuration resulting from the execution of line 23 by s_t for *op*. Otherwise, the linearization point of *op* is placed at the configuration resulting from the execution of line 28 by s_t for *op*.

- *op* is a pop operation. Let s_t be the token server that responds to the client that initiated *op* (line 9). If the operation returns NACK, the linearization point of *op* is placed at the configuration resulting from the execution of line 42 by s_t for *op*. Otherwise, the linearization point of *op* is placed at the configuration resulting from the execution of line 37 by s_t for *op*.

Denote by L the sequence of operations (which have been assigned linearization points) in the order determined by their linearization points.

Lemma 70. *The linearization point of a push (pop) operation op is placed in its execution interval.*

Proof. Assume that *op* is a push operation and let c be the client that invokes it. After the invocation of *op*, c sends a message to some server s and awaits a response. Recall that routine `receive()` (line 4) blocks until a message is received. The linearization point of *op* is placed either in the configuration resulting from the execution of line 23 by s_t for *op*, where s_t is the token server in this configuration, or in the configuration resulting from the execution of line 28 by s_t for *op*.

Either of these lines is executed after the request by c is received, i.e. after c invokes `ClientPush`. Furthermore, they are executed before c receives the response by s_t and thus, before `ClientPush` returns. Therefore, the linearization point is inside the execution interval of push.

The argumentation regarding pop operations is analogous. □

Each server maintains a local variable *token* with initial value 0 (initially, the server with id equal to 0 is the token server). Whenever some server s_i receives a TOKEN message, i.e. a message with its *tk* field equal to TOKEN (line 17), the value of *token* is set to i. By inspection of the pseudocode, it follows that the value of *token* is set to the id of the next server if the local stack of s_i is full (line 25); then, a TOKEN message is sent to the next server (line 26). Moreover, the value of *token* is set to the id of the previous server if the local stack *lstack* of s_i is empty (line 38); then, a TOKEN message is sent to the previous server (lines 39-40). (Unless the server is s_0 in which case a NACK is sent to the client (line 42 but no TOKEN message to any server.) Thus, the following observation holds.

Observation 71. *At each configuration in α, there is at most one server s_i for which the local variable token has the value i.*

At each configuration C, the server s_i whose *token* variable is equal to i is referred to as the token server at C.

Observation 72. *A* TOKEN *message is sent from a server with id i, $0 \leq i < \mathsf{NS} - 1$, to a server with id $i + 1$ only if the local stack of server i is full. A* TOKEN *message is sent from a* *server with id i, $0 < i \leq \mathsf{NS} - 1$, to a server with id $i - 1$ only when the local stack of server i is empty.*

By the pseudocode, namely the `if` clause of line 18 and the `if` clause of line 32, the following observation holds.

Observation 73. *Whenever a server s_i performs push and pop operations on its local stack (lines 22 and 36), it holds that its local variable token is equal to i.*

Let C_i be the configuration at which the i-th operation op_i of L is linearized. Denote by α_i, the prefix of α which ends with C_i and let L_i be the prefix of L up until the operation that is linearized at C_i. Denote by S_i the sequence of values that a sequential stack contains after applying the sequence of operations in L_i, in order, starting from an empty stack; let $S_0 = \epsilon$, i.e. S_0 is the empty sequence.

Lemma 74. *For each i, $i \geq 0$, if s_{k_i} is the token server at C_i and ls_i^j are the contents of the local stack of server j, $0 \leq j \leq k_i$, at C_i, then it holds that $S_i = ls_i^0 \cdot ls_i^1 \cdot \ldots \cdot ls_i^{k_i}$ at C_i.*

Proof. We prove the claim by induction on i. The claim holds trivially for $i = 0$. Fix any $i \geq 0$ and assume that at C_i, it holds that $S_i = ls_i^0 \cdot ls_i^1 \cdot \ldots \cdot ls_i^{k_i}$. We show that the claim holds for $i + 1$.

We first assume that op_{i+1} is a push operation initiated by some client c. Assume first that $s_{k_i} = s_{k_{i+1}}$. Then, by induction hypothesis, $S_i = ls_i^0 \cdot \ldots \cdot ls_i^{k_i}$. In case the local stack of s_{k_i} is not full, s_{k_i} pushes the value v_{i+1} of field *data* of the request onto its local stack and responds to c. Since no other change occurs to the local stacks of $s_0, \ldots, s_{k_i}$ from C_i to C_{i+1}, at C_{i+1}, it holds that $S_{i+1} = ls_i^0 \cdot \ldots \cdot ls_i^k \cdot \{v_{i+1}\} = ls_i^0 \cdot \ldots \cdot ls_{i+1}^{k_i}$. In case that the local stack of s_{k_i} is full, since $s_{k_i} = s_{k_{i+1}}$ and it is the token server, it follows that $s_{k_i} = s_{\mathsf{NS}-1}$. In this case, s_{k_i} responds with a NACK to c and the local stack remains unchanged. Thus, it holds that $S_{i+1} = ls_i^0 \cdot \ldots \cdot ls_i^k = S_i$.

Assume now that $s_{k_i} \neq s_{k_{i+1}}$. This implies that the local stack of s_{k_i} is full just after C_i. Observation 72 implies that s_{k_i} forwarded the token to s_{k_i+1} in some configuration between C_i and C_{i+1}. Notice that then, $s_{k_i+1} = s_{k_{i+1}}$. Observation 73 implies that the local stack of s_{k_i+1} is empty. Thus, the `if` condition of line 21 evaluates to `true` for server s_{k_i+1} and therefore, it pushes the value v_{i+1} of op_{i+1} onto its local stack. Thus, at C_{i+1}, $ls_{i+1}^{k_i+1} = \{v_{i+1}\}$. By definition, $S_{i+1} = S_i \cdot \{v_{i+1}\}$. Therefore, $S_{i+1} = ls_i^0 \cdot \ldots \cdot ls_{i+1}^{k_i+1}$. And since by Observations 71 and 73, the contents of the local stacks of servers other than $k_i + 1$ do not change, it holds that $S_{i+1} = ls_{i+1}^0 \cdot \ldots \cdot ls_{i+1}^{k_i+1} = ls_{i+1}^0 \cdot \ldots \cdot ls_{i+1}^{k_{i+1}}$.

The reasoning for the case where op_{i+1} is an instance of a pop operation is symmetrical. □

From the above lemmas and observations, we have the following.

Theorem 75. *The token-based distributed stack implementation is linearizable. The time complexity and the communication complexity of each operation op is $O(\mathsf{NS})$.*

4.2.2 A Token-based Queue

The token-based distributed queue implementation follows similar ideas as those of the token-based stack implementation of Section 4.2.1. To implement a queue, two tokens are employed: at each point in time, there is a head token server s_h and a tail token server s_t. Initially, server 0 plays the role of both s_h and s_t. Each server s_i, other than s_t (s_h), that receives a request (directly) from a client c, it forwards the request to the next server to ensure that it will either reach the appropriate token server or return back to s_i (after traversing all servers). Servers s_t and s_h work in a way similar as server s_t in stacks.

To prevent a request from being forwarded forever due to the completion of concurrent requests which may cause the token(s) to keep advancing, each server keeps track of the request that each client c (directly) sends to it, in a *client table* (there can be only one such request per client). Server s_t (and/or s_h) now reports the response to s_i which forwards it to c. If s_i receives a response for a request recorded in its client table, it deletes the request from the client table. If s_i receives the token (tail, or head), it serves each request (enqueue, or dequeue, respectively) in its client array and records its response. If a request, from those included in s_i's client array, reaches s_i again, s_i sends the response it has calculated for it to the client and removes it from its client array. Since the communication channels are FIFO, the implementations ensures that all requests, their responses, and the appropriate tokens, move from one server to the next, based on the servers' ring order, until they reach their destination. This is necessary to argue that the technique ensures termination for each request.

4.2.2.1 Algorithm Description

Pseudocode for the client's side *DS* operations is presented in Algorithm 18. Enqueue and dequeue operations are carried out by `ClientEnqueue()` and `ClientDequeue()`, respectively: An operation *op* is invoked on a client c by invoking one of these routines. Subsequently, c sends a message to the server that it considers to be tail token server, in case of an enqueue operation (line 4), or the server it considers to be head token server, in case of a dequeue operation (line 8), and then awaits the response. Notice that as in the case of the token-based stack, the clients in their initial state consider that s_0 holds the head and tail tokens and they keep track of the changes in token servers in a lazy way.

On the server side as well, the queue implementation is based on similar ideas as the token-based distributed stack of Section 4.2.1: Clients initially send their requests to what they consider to be the token server, servers message each other in order reassign the tokens. Each server maintains a local queue *lqueue* on which it performs the enqueue and dequeue requests, depending on whether that local queue is empty or full, respectively, and on whether the server holds the appropriate tokens.

The token servers receive, process, and respond to clients' messages. As with previous data structures, a field in each message indicates the type of operation that is requested. Event-driven pseudocode for a server is presented in Algorithm 19. Apart from the local queue *lqueue*, each

Algorithm 18 Enqueue and Dequeue operations for a client of the token-based queue.

```
int $enq\_sid = 0$
int $deq\_sid = 0$

Data ClientEnqueue(int $cid$, Data $data$)
  send($enq\_sid$, $\langle$ENQ$, data, cid, -1\rangle$)
  $\langle status, \perp, enq\_sid\rangle$ = receive($enq\_sid$)
  return $status$

Data ClientDequeue(int $cid$)
  send($deq\_sid$, $\langle$DEQ$, \perp, cid\rangle$)
  $\langle status, data, deq\_sid\rangle$ = receive($deq\_sid$)
  return $data$
```

server keeps two boolean flag variables, ($hasHead$ and $hasTail$), in order to monitor whether it has the token or not. Furthermore, it uses a bit flag, $fullQueue$, which indicates whether the local queue is full. A defining difference from the token-based stack is the client array that each server has to maintain. This is implemented as a local array of size n, where n is the maximum number of clients. Initially, the client table of a server is empty. As it receives requests, it stores in the client table all those requests that it has received from clients directly – in contrast to those requests that it received because they were forwarded to it by another server.

The messages that reach a server have five fields: (1) *op* designates the operation (ENQ or DEQ) up to when it is served and after that, it indicates whether it has been successful or not (ACK or NACK), (2) *data* stores the data to be added to the queue in case *op* = ENQ, and $\perp$ otherwise, (3) *cid* is the id of the client that issued the request, (4) *sid* is the id of the server that forwarded the message towards the token server and has the value -1 if that has not occurred, and (5) *tk* takes the values TAIL_TOKEN or HEAD_TOKEN when it is a field of a forwarded messages (of type ENQ or DEQ, respectively), and is used to indicate that a head or tail token transfer is required. When that is not the case, it is equal to $\perp$.

When a server s_i, $0 \leq i <$ NS, that is not the tail token server, receives a message of type ENQ (line 23), it first checks if it contains a token transfer from another server (line 24). Assume first that it does not (line 28). In this case, s_i forwards the request to the next server in the ring (line 29, lines 32 and 34) so that it can eventually reach s_t. A bad scenario that could occur is that the client request may be transmitted indefinitely from a server to the next without ever reaching the appropriate token server. This can happen if, in the meanwhile, both the head and the tail tokens are forwarded indefinitely along the ring. To avoid this, if s_i receives a message from a client (line 30) but cannot serve it, then s_i updates its clients table, storing in it information about the request (line 31), before it forwards the message towards s_t.

If the message reaches s_t, then it attempts to serve the request. If the local queue *lqueue* of s_i is not full, then s_t it enqueues the given data. Recall that the message may reach s_t either because it was sent directly from the client or because it was forwarded to it from another server, let it be s_i. If the former is the case, then s_t responds with an ACK directly to the client (lines 37 - 38). If the latter is the case, then, deviating from the stack implementation, once s_t serves the request it does not respond to the client directly. Instead, it sends an ACK message

Algorithm 19 Events triggered in a server of the token-based queue.

```
11 int $my\_sid$
12 LocalQueue $lqueue = \varnothing$
13 LocalArray $clients = \varnothing$ // Array of three values: <op, data, isServed>
14 boolean $fullQueue$ = false// True when tail and head are in the same server and tail is before head
15 boolean $hasHead$ // Initially hasHead and hasTail are true in server 0, and false in the rest
16 boolean $hasTail$

17 a message $\langle op, data, cid, sid, tk \rangle$ is received:
18   if (!clients[$cid$] AND clients[$cid$].$isServed$) then // If message has been served earlier.
19     send($cid$, ⟨ACK, clients[$cid$].$data$, $my\_sid$⟩)
20     clients[$cid$] = $\perp$
21   else
22     switch ($op$)
23       case ENQ: // The message contains an enqueue request
24         if ($tk$ == TAIL_TOKEN) then
25           $hasTail$ = true
26           if ($hasHead$) $fullQueue$ = true
27           ServeOldEnqueues()
28         if (!$hasTail$) then // Server does not have token
29           $nsid$ = find_next_server(($my\_sid$)
30           if ($sid == -1$) then // From client.
31             clients[$cid$] = ⟨ENQ, $data$, false⟩
32             send($nsid$, ⟨ENQ, $data, cid, my\_sid, \perp$⟩)
33           else // From server.
34             send($nsid$, ⟨ENQ, $data, cid, sid, \perp$⟩)
35         else if (!IsFull($lqueue$)) then // Server can enqueue.
36           enqueue($lqueue, data$)
37           if ($sid == -1$) // From client.
38             send($cid$, ⟨ACK, $\perp, my\_sid$⟩)
39           else// From server.
40             send($sid$, ⟨ACK, $\perp, cid, my\_sid, \perp$⟩)
41         else if ($fullQueue$) then // Global Queue full
42           if ($sid$ == -1) // From client.
43             send($cid$, ⟨NACK, $\perp, my\_sid$⟩)
44           else // From server
45             send($sid$, ⟨NACK, $\perp, cid, my\_sid, \perp$⟩)
46         else // Server moves the tail token to the next server
47           $nsid$ = find_next_server($my\_sid$)
48           $fullQueue$ = false
49           $hasTail$ = false
50           send($nsid$, ⟨$op, data, cid, my\_sid$, TAIL_TOKEN⟩)
51         break
```

```
52     case DEQ: // The message contains an dequeue request
53       if ($tk$ == HEAD_TOKEN) then
54         $hasHead$ = true
55         ServeOldDequeues()
56       if (!$hasHead$) then
57         $nsid$ = find_next_server($my\_sid$)
58         if ($sid == -1$) then // From client
59           clients[$cid$] = $\langle$DEQ, $\perp$, false$\rangle$
60           send($nsid$, $\langle$DEQ, $\perp$, $cid$, $my\_sid\rangle$)
61         else // From server
62           send($nsid$, $\langle$DEQ, $\perp$, $cid$, $sid$, $\perp\rangle$)
63       else if (!IsEmpty($lqueue$)) then // Server can dequeue.
64         $data$ = dequeue($lqueue$)
65         if ($sid == -1$) then // From client
66           send($cid$, $\langle$ACK, $data$, $my\_sid\rangle$)
67         else // From server
68           send($sid$, $\langle$ACK, $data$, $cid$, $my\_sid$, $\perp\rangle$)
69       else if ($hasTail$ AND !$fullQueue$) then // Queue is empty
70         if ($sid == -1$) then // From client
71           send($cid$, $\langle$NACK, $\perp$, $my\_sid\rangle$)
72         else // From server
73           send($sid$, $\langle$NACK, $\perp$, $cid$, $my\_sid$, $\perp\rangle$ )
74       else // Server moves the head token to the next server
75         $nsid$ = find_next_server($my\_sid$)
76         $hasHead$ = false
77         send($nsid$, $\langle op$, $\perp$, $cid$, $my\_sid$, HEAD_TOKEN$\rangle$)
78       break
79     case ACK:
80       clients[$cid$] = $\perp$
81       send($cid$, $\langle$ACK, $data$, $sid\rangle$)
82       break
83     case NACK:
84       clients[$cid$] = $\perp$
85       send($cid$, $\langle$NACK, $\perp$, $sid\rangle$)
86       break
```

back to s_i (lines 39-40) and it is s_i that responds to the client with an ACK message. In order to keep the clients up-to-date with the approximate location of the token, this message also includes the id of the server that currently holds the token.

If the token-based queue is full, as indicated when $fullQueue$ is true, then s_t sends a NACK message to the client or the server that the message was sent from (line 41-45). It may however be that the $lqueue$ of s_t is full but that the token-based queue is not. In that case, s_t moves the enqueue request to the next server in the ring (line 47), encapsulating in it a token transfer, by setting tk = TAIL_TOKEN and $hasTail$ = false (lines 46 - 50).

Assume now that a server s_i, that is not tail token server, receives a request, forwarded to it by another server and that the message does include a token transfer. Then, s_i sets its token *hasTail* to `true` (line 25) and if it also had the head token from a previous message, then it changes the *fullQueue* flag to `true` as well (line 26). Then, s_i serves all pending enqueue requests that it has stored in its client table (line 27).

Algorithm 20 Auxiliary functions for a server of the token-based queue.

```
87 void ServeOldEnqueues(void)
88   if (!$fullQueue$) then
89     for each $cid$ such that clients[$cid$].$op$ == ENQ do
90       if (!IsFull($lqueue$)) then
91         enqueue($lqueue$, clients[$cid$].$data$)
92         clients[$cid$].$isServed$ = true

93 void ServeOldDequeues(void)
94   for each $cid$ such that clients[$cid$].$op$ == DEQ do
95     if (!IsEmpty($lqueue$)) then
96       clients[$cid$].$data$ = dequeue($lqueue$)
97       clients[$cid$].$isServed$ = true
```

In order to deal with requests that are stored in the client table of a server, additional mechanisms are required. Let s_i received a message of type ACK (line 79) or NACK (line 83) for that request. In this case, s_i sets the entry *cid* of its client table to $\perp$ (lines 80, 84) and sends an ACK (line 81) or a NACK (line 85) to that client. In either of those cases, the request has been served by another server and can be deleted from the client table (lines 19, 20). However, recall that the request may do a round-trip on the server ring and reach s_i again without having been served. When this happens, then s_i obtains the tail token as well. Then s_i has to serve all its pending enqueue requests, indicated on the client table. In order to do this, it uses `ServeOldEnqueues()`.

The actions that are performed by a server in the case of a dequeue request are analogous. In the dequeue case, pending requests are handled through `ServeOldDequeues()`.

Functions `ServeOldEnqueues()` and `ServeOldDequeues()` are described in more detail in Algorithm 20. `ServeOldEnqueues()` (line 87) processes all enqueue requests stored in the client table, if the local queue has space (line 90). Similarly, `ServeOldDequeues()` (line 93) processes all dequeue requests stored in the client table, if the local queue is not empty (line 95).

4.2.2.2 Proof of Correctness

Let α be an execution of the token-based queue algorithm presented in Algorithms 18, 19, and 20. Each server maintains local boolean variables *hasHead* and *hasTail*, with initial values `false`. Whenever some server s_i receives a TAIL_TOKEN message, i.e. a message with its *tk* field equal to TAIL_TOKEN (line 24), the value of *hasTail* is set to `true` (line 25). By inspection of the pseudocode, it follows that the value of *hasTail* is set to `false` if the local queue of s_i is full (line 35, 46- 49); then, a TAIL_TOKEN message is sent to the next server (line 50). The same holds for *hasHead* and HEAD_TOKEN messages, i.e. messages with their *tk* field equal to

HEAD_TOKEN. Thus, the following observations holds.

Observation 76. *At each configuration in α, there is at most one server for which the local variable hasHead (hasTail) has the value* `true`*.*

Observation 77. *In some configuration C of α,* TAIL_TOKEN *message is sent from a server s_j, $0 \leq j < \mathsf{NS} - 1$, to a server s_k, where $k = (j+1) \mod \mathsf{NS}$ only if the local queue of s_j is full in C. Similary, a* HEAD_TOKEN *message is sent from s_j to s_k only if the local queue of s_j is empty in C.*

By inspection of the pseudocode, we see that a server performs an enqueue (dequeue) operation on its local queue *lqueue* either when executing line 36 (line 55) or when executing `ServeOldEnqueues` (`ServeOldDequeues`). Further inspection of the pseudocode (lines 24-27, lines 35-41, as well as lines 56-62, lines 63-69), shows that these lines are executed when $hasTail =$ `true`. Then, the following observation holds.

Observation 78. *Whenever a server s_j performs an enqueue (dequeue) operation on its local queue, it holds that its local variable hasTail (hasHead) is equal to* `true`*.*

By a straight-forward induction, the following lemma can be shown.

Lemma 79. *The mailbox of a client in any configuration of α contains at most one incoming message.*

If $hasTail =$ `true` ($hasHead =$ `true`) for some server s in some configuration C, then we say that s has the tail (head) token. The server that has the tail token is referred to as tail token server. The server that has the head token is referred to as head token server.

Let *op* be any operation in α. We assign a linearization point to *op* by considering the following cases:

- If *op* is an enqueue operation for which a tail token server executes an instance of Algorithm 19, then it is linearized in the configuration resulting from the execution of either line 36, or line 91, or line 43, whichever is executed for *op* in that instance of Algorithm 19 by the tail token server.
- If *op* is a dequeue operation for which a head token server executes an instance of Algorithm 19, then it is linearized in the configuration resulting from the execution of either line 64, or line 96, or line 66, whichever is executed for *op* in that instance of Algorithm 19 by the head token server.

Lemma 80. *The linearization point of an enqueue (dequeue) operation op is placed in its execution interval.*

Proof. Assume that *op* is an enqueue operation and let c be the client that invokes it. After the invocation of *op*, c sends a message to some server s (line 4) and awaits a response. Recall that routine `receive()` (line 5) blocks until a message is received. The linearization point of *op* is placed either in the configuration resulting from the execution of line 36 by s_t for *op*, in the configuration resulting from the execution of line 43 by s_t for *op*, or in the configuration resulting from the execution of line 91 by s_t for *op*. Notice that either of these lines is executed after the request by c is received, i.e. after c invokes `ClientEnqueue`, and thus, after the execution interval of *op* starts.

By definition, the execution interval of *op* terminates in the configuration resulting from the execution of line 6. By inspection of the pseudocode, this line is executed after line 5, i.e. after c receives a response by some server. In the following, we show that the linearization point of *op* occurs before this response is sent to c.

Let s_j be the server that c initially sends the request for *op* to. By observation of the pseudocode, we see that c may either receive a response from s_j if s_j executes lines 38 or 43, or if s_j executes lines 80-81 or lines 84-85, or if s_j executes line 19. To arrive at a contradiction, assume that either of these lines is executed in α before the configuration in which the linearization point of *op* is placed. Thus, a tail token server s_t executes lines 36, 91, or 43 in a configuration following the execution of lines 38, or 43, or 80-81 or 84-85, or line 19 by s_j. Since the algorithm is event-driven, inspection of the pseudocode shows that in order for a tail token server to execute these lines, it must receive a message containing he request for *op* either from a client or from another server.

Assume first that a tail token server executes the algorithm after receiving a message containing a request for *op* from a client. This is a contradiction, since, on one hand, c blocks until receiving a response, and thus, does not sent further messages requesting *op* or any other operation, and since *op* terminates after c receives the response by s_j, and on the other hand, any other request from any other client concerns a different operation op'.

Assume next that a tail token server executes the algorithm after receiving a message containing the request for *op* from some other server. This is also a contradiction since inspection of the pseudocode shows that after s_j executes either of the lines that sends a response to c, it sends no further message to some other server and instead, terminates the execution of that instance of the algorithm.

The argumentation regarding dequeue operations is analogous. □

Denote by L the sequence of operations which have been assigned linearization points in α in the order determined by their linearization points. Let C_i be the configuration at which the i-th operation op_i of L is linearized. Denote by α_i, the prefix of α which ends with C_i and let L_i be the prefix of L up until the operation that is linearized at C_i. Denote by Q_i the sequence of values that a sequential queue contains after applying the sequence of operations in L_i, in order, starting from an empty queue; let $Q_0 = \epsilon$, i.e. Q_0 is the empty sequence. In the following, we denote by s_{t_i} the tail token server at C_i and by s_{h_i} the head token server at C_i.

Lemma 81. *For each i, $i \geq 0$, if lq_i^j are the contents of the local queue of server s_j at C_i, $h_i \leq j \leq t_i$, at C_i, then it holds that $Q_i = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i}$ at C_i.*

Proof. We prove the claim by induction on i. The claim holds trivially at $i = 0$.

Fix any $i \geq 0$ and assume that at C_i, it holds that $Q_i = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i}$. We show that the claim holds for $i+1$.

First, assume that op_{i+1} is an enqueue operation by client c. Furthermore, distinguish the following two cases:

- Assume that $t_i = t_{i+1}$. Then, by the induction hypothesis, $Q_i = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i}$. In case the local queue of s_{t_i} is not full, s_{t_i} enqueues the value v_{i+1} of the *data* field of the request for op_{i+1} in the local queue (line 36 or line 91). Notice that, by Observation 78 changes on the local queues of servers occur only on token servers. Notice also that those changes occur only in a step that immediately precedes a configuration in which an operation is linearized. Thus, no further change occurs on the local queues of $s_{h_i}, s_{h_i+1}, \ldots, s_{t_i}$ between C_i and C_{i+1}, other than the enqueue on lq_i^t. Then, it holds that $Q_{i+1} = Q_i \cdot v_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot v_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_{i+1}^{t_i} = lq_{i+1}^{h_i} \cdot lq_{i+1}^{h_i+1} \cdot \ldots \cdot lq_{i+1}^{t_i}$, and if the head token server does not change between C_i and C_{i+1}, then $h_{i+1} = h_i$ and $Q_{i+1} = lq_{i+1}^{h_{i+1}} \cdot lq_{i+1}^{h_{i+1}+1} \cdot \ldots \cdot lq_{i+1}^{t_{i+1}}$ and the claim holds. If the head token server changes, i.e., if $h_{i+1} \neq h_i$, then by Observation 77, $lq_{i+1}^{h_i} = \emptyset$ and the claim holds again.

  In case the local queue of s_{t_i} is full and since by assumption, $s_{t_i} = s_{t_{i+1}}$, it follows by inspection of the pseudocode (line 41) and the definition of linearization points, that $s_{t_{i+1}} = s_{h_{i+1}}$. In this case, $s_{t_{i+1}}$ responds with a NACK to c and the local queue remains unchanged. Since no token server changes between C_i and C_{i+1}, $Q_{i+1} = Q_i = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} = lq_{i+1}^{h_{i+1}} \cdot lq_{i+1}^{h_{i+1}+1} \cdot \ldots \cdot lq_{i+1}^{t_{i+1}}$ and the claim holds.

- Next, assume that $t_i \neq t_{i+1}$. This implies that the local queue of s_{t_i} is full just after C_i. Observation 77 implies that s_{t_i} forwarded the token to s_{t_i+1} in some configuration between C_i and C_{i+1}. Notice that then, $s_{t_i+1} = s_{t_{i+1}}$. If the local queue of $s_{t_{i+1}}$ is not full, then the condition of line 35 evaluates to `true` and therefore, line 36 is executed, enqueueing value v_{i+1} to it. Then at C_{i+1}, $lq_{i+1}^{t_{i+1}} = v_{i+1}$. By definition, $Q_{i+1} = Q_i \cdot v_{i+1}$, and therefore, $Q_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot v_{i+1} = lq_{i+1}^{h_{i+1}} \cdot lq_{i+1}^{h_{i+1}+1} \cdot \ldots \cdot lq_{i+1}^{t_i} \cdot v_{i+1} = lq_{i+1}^{h_{i+1}} \cdot lq_{i+1}^{h_{i+1}+1} \cdot \ldots \cdot lq_{i+1}^{t_i} \cdot lq_{i+1}^{t_{i+1}}$ and the claim holds. If the local queue of $s_{t_{i+1}}$ is full, then the condition of line 35 evaluates to `false` and therefore, line 45 is executed. The operation is linearized in the resulting configuration and NACK is sent to c. Notice that in that case, the local queue of the server is not updated. Then, $Q_{i+1} = Q_i = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot lq_{i+1}^{t_{i+1}} = lq_{i+1}^{h_{i+1}} \cdot lq_{i+1}^{h_{i+1}+1} \cdot \ldots \cdot lq_{i+1}^{t_i} \cdot lq_{i+1}^{t_{i+1}}$, and the claim holds.

The reasoning for the case where op_{i+1} is an instance of a dequeue operation is symmetrical. □

From the above lemmas and observations we have the following theorem.

Theorem 82. *The token-based distributed queue implementation is linearizable. The time complexity and the communication complexity of each operation op is* $O(\mathsf{NS})$.

4.2.3 A Token-based Unsorted List

In order to implement a list, its elements are stored in the local memory modules of several of the available servers, potentially spreading among all of them, if its size is large enough. The proposed implementation follows a token-based approach for implementing insert operations: At each point in time, there is a server (not necessarily always the same), denoted by s_t, which holds the insert token, and serves insert operations. Initially, server s_0 has the token, thus the first element to be inserted in the list is stored on server s_0. Further element insertions are also performed on it, as long as the space it has allocated for the list does not exceed a threshold. In case s_0 has to serve an insert but its space is filled up, it forwards the token by sending a message to the next server, i.e. server s_1. The token may propagate to subsequent servers in that manner.

In case the token reaches s_0 again, then, if the allocated memory chunk of s_0 is still full, s_0 allocates more memory for storing more list elements. The token might go through the server ring again without having any upper-bound restrictions concerning the number of round-trips. In order for a server to know whether the token has performed a round-trip on the ring, and hence all servers have stored list elements, it deploys a variable to count the number of ring round-trips it knows that the token has performed.

4.2.3.1 Algorithm Description

Pseudocode for the client's side DS operations is presented in Algorithm 21. Insert operations are carried out by invoking `ClientInsert()`, search operations by invoking `ClientSearch()`, and delete operations by invoking `ClientDelete()`. It is notable that insert operations in the proposed implementation are executed in sequence and must necessarily pass through server 0 and be forwarded through the server ring, if necessary due to space constraints. Search and delete operations, on the contrary, are executed in parallel.

In more detail, after a client invokes `ClientInsert()` (line 41), it sends an INSERT message (line 45) to server 0, regardless of which server holds the token in any given configuration, and then blocks waiting for a response (line 46). If the client receives ACK from a server, then the element was inserted correctly. If the client receives NACK, then the insertion failed, due to either limited space, or the existence of another element with the same key value.

For a search operation the client invokes `ClientSearch()` (line 57), which sends a SEARCH request to all servers (line 62) and waits to receive a response message (line 64) from each server (`do while` loop of lines 63-67). The requested element is in the list if the client receives ACK from some server (line 65). A delete operation proceeds similarly to `ClientSearch()`. It is initiated by a client by sending a DELETE request to all servers (line 74). The client then waits to receive a response message (line 76) from each server (`do while` loop of lines 75-79). The requested element has been found in the list of some client and deleted from there, if the client receives ACK from some server s.

Event-driven code for the server is presented in Algorithm 22. Each server s maintains a

Algorithm 21 Insert, Search and Delete operation for a client of the distributed list.

```
boolean ClientInsert(int $cid$, int $key$, data $data$)
  boolean $status$
  send(0, $\langle \textsf{INSERT}, cid, key, data, \texttt{false}, -1 \rangle$)
  $status$ = receive()
  return $status$

boolean ClientSearch(int $cid$, int $key$)
  int $sid$
  int $c = 0$
  boolean $status$
  boolean $found$ = false
  send_to_all_servers($\langle \textsf{SEARCH}, cid, key, \perp, \texttt{false}, -1 \rangle$)
  do
    $\langle status, sid \rangle$ = receive()
    if ($status$ == ACK) then $found$ = true
    $c{+}{+}$
  while ($c <$ NS)
  return $found$

boolean ClientDelete(int $cid$, int $key$)
  int $sid$
  int $c = 0$
  boolean $status$
  boolean $deleted$ = false
  send_to_all_servers($\langle \textsf{DELETE}, cid, key, \perp, \texttt{false}, -1 \rangle$)
  do
    $\langle status, sid \rangle$ = receive()
    if ($status$ == ACK) $deleted$ = true
    $c{+}{+}$
  while ($c <$ NS)
  return $deleted$
```

local list (*llist* variable) allocated for storing list elements, a *token* variable which indicates whether s currently holds the token, and a variable *round* to mark the ring round-trips the token has performed; *round* is initially 0, and is incremented after every transmission of the token to the next server.

Each message a server receives has five fields: (1) *op* that denotes the operation to be executed, (2) *cid* that holds the id of the client that initiated a request, (3) *key* that holds the value to be inserted, (4) *mloop* stands for "message loop", a boolean value that denotes if the message has traversed the whole server sequence and (5) *tk* that is set when a forwarded message also denotes a token transition from one server to the other.

When a message is received, the server s first checks its type. If the message is of type INSERT (line 5), s first checks whether the message has the *tk* field marked. If it is marked (line 6), s sets a local variable *token* equal to its own id (line 7) and allocates additional space for its local part of the list (line 8).

Algorithm 22 Events triggered in a server of the distributed unsorted list.

```
30 List $llist = \varnothing$
31 int $my\_id$, $next\_id$, $token = 0$, $round = 0$

32 a message $\langle op, cid, key, data, mloop, tk \rangle$ is received:
33   switch ($op$)
34     case INSERT:
35       if ($tk ==$ TOKEN) then
36         $token = my\_id$
37         allocate_new_memory_chunk($llist, round$)
38       $status_1$ = search($llist$, $key$)
39       if ($status_1$) then send($cid$, NACK)
40       else
41         if ($token \neq my\_id$) then
42           $next\_id$ = get_next($my\_id$)
43           if ($my\_id \neq$ NS $- 1$) then
44             send($next\_id$, $\langle op, cid, key, data, mloop, tk \rangle$)
45           else   send($next\_id$, $\langle op, cid, key, data,$ true$, tk \rangle$)
46         else
47           if (($my\_id \neq$ NS $- 1$) AND ($round > 0$) AND !($mloop$)) then
48             $next\_id$ = get_next($my\_id$)
49             send($next\_id$, $\langle op, cid, key, data, mloop, tk \rangle$)
50           else
51             $status_2$ = insert($llist, round, key, data$)
52             if ($status_2 ==$ false) then
53               $round++$
54               $token$ = get_next($my\_id$)
55               send($token$, $\langle op, cid, key, data, mloop,$ TOKEN$\rangle$)
56             else send($cid$, ACK)
57       break
58     case SEARCH:
59       $status_1$ = search($llist$, $key$)
60       if ($status_1$) then send($cid$, $\langle$ACK$, my\_id \rangle$)
61       else send($cid$, $\langle$NACK$, my\_id \rangle$)
62       break
63     case DELETE:
64       $status_1$ = delete($llist$, $key$)
65       if ($status_1$)then send($cid$, ACK)
66       else send($cid$, NACK)
67       break
   }
```

Afterwards, s searches the part of the list that it stores locally, for an element with the same key (key variable in the algorithm) as the one to be inserted (line 9). Searching$llist$ for the element has to be performed independently of whether the server holds the token or not. Since this design does not permit duplicate entries, if such an element is found, the server responds with NACK to the client (line 12). Otherwise (line 17), s checks whether the new element can be stored in $llist$.

In case s does not hold the token (line 20), it is not allowed to perform an insertion, therefore it must forward the message to the next server in the ring. If s is not $s_{\mathsf{NS}-1}$ (line 43), it forwards to the next server the request (line 22). In case s is $s_{\mathsf{NS}-1}$, it means that all servers have been searched for the element and the element was not found. Server s sends the message to the next server (in order to eventually reach the token server), after marking the *mloop* field of the message as `true`, to indicate that the message has completed a full round-trip on the ring (line 45).

On the other hand, if s holds the token (line 23), it must first check whether there is room in *llist* to insert the element in it. If there is room in *llist* and the local variable *round* of s equals to `false` (which means that the list does not expand to the next servers)or the message has already performed a round-trip on the ring, then s inserts the element and returns ACK. If however, $round > 0$ and the message has not performed a round trip on the ring ($mloop ==$ `false`), s continues forwarding the message.

If the token server's local memory is out of sufficient space (line 25) (i.e. the `insert()` function was unsuccessful), s forwards the message to the next server the *tk* field with TOKEN (line 28) to indicate that this server will become the new token server after s. Also, s increments *round* by one to count the number of times the token has passed from it. The *round* variable is also used by function `allocate_new_memory_chunk()` that allocates additional space for the list (line 8).

Notice that, contrary to other token-based implementations presented in previous sections, the token server of the unsorted list does not need to rely on client tables in order to stop a message from being incessantly forwarded from one server to another, without ever being served. By virtue of having clients always sending their insert requests to s_0, an insert request r_j that arrives at s_0 before some other insert request r_k, is necessarily served before r_k. The scenario where insert requests constantly arrive at the token server before r_j, making the token travel to the next server before r_j can be served, is thus avoided.

Upon receiving a SEARCH request from a client (line 31), a server searches for the requested element in its local list (line 32) and sends ACK to the server if the element is found (line 33) and NACK otherwise (line 34).

Upon receiving a DELETE request from a client (line 36), a server attempts to delete the requested element from its local list (line 37) and sends ACK to the server if the deletion was successful (line 38). Otherwise it sends NACK (line 39).

4.2.3.2 Proof of Correctness

We sketch the correctness argument for the proposed implementation by providing linearization points. Let α be an execution of the distributed unsorted list algorithm presented in Algorithms 21 and 22. We assign linearization points to insert, delete and search operations in α as follows:

- Insert. Let *op* be any instance of `ClienInsert` for which an ACK or a NACK message is sent by a token server. Then, if ACK is sent by a token server for *op* (line 29), the linearization point is placed in the configuration resulting from the execution of line 24 that successfully inserted the required element into the server's local list. If NACK is sent for *op* (line 12), then the linearization point is placed in the configuration resulting from the execution of line 9, where the search operation on the local list of the server returned `true`.

- Let *op* be any instance of `ClientDelete` for which an ACK or a NACK message is sent by a server. Then, if ACK is sent by a server s for *op*, the linearization point is placed in the configuration resulting from the execution of line 37 by the server that sent the ACK. Otherwise, if the key k that *op* had to delete was not present in any of the local lists of the servers in the beginning of the execution interval of *op*, then the linearization point of *op* is placed at the beginning of its execution interval. Otherwise, if k was present but was deleted by a concurrent instance op' of `ClientDelete`, then the linearization point is placed right after the linearization point of op'.
- Let *op* be any instance of `ClientSearch` for which an ACK or a NACK message is sent by a server. Then, if ACK is sent by a server s for *op*, the linearization point is placed in the configuration resulting from the execution of line 32 by the server that sent the ACK. Otherwise, if the key k that *op* had to find was not present in the list in the beginning of its execution interval, then the linearization point is placed there. Otherwise, if k was present but was deleted by a concurrent instance op' of `ClientDelete`, then the linearization point is placed right after the linearization point of op'.

Lemma 83. *Let op be any instance of an insert, delete, or a search operation executed by some client c in α. Then, the linearization point of op is placed in its execution interval.*

Proof. Let *op* be an instance of an insert operation invoked by client c. A message with the insert request is sent on line 45, after the invocation of the operation. Recall that routine `receive()` blocks until a message is received. Notice that both line 24 as well as line 9 are executed by a server before it sends a message to the client. Therefore, whether *op* is linearized at the point some server sends it a message on line 29 or on line 12, it terminates only after receiving it. Notice also that the operation terminates only after the client receives it. Thus, the linearization point is included in its execution interval.

By similar reasoning, if *op* is an instance of a delete operation that is linearized in the configuration resulting from the execution of line 37 or a search operation that is linearized in the configuration resulting from the execution of line 32, then the linearization point is included in the execution interval of *op*.

Let *op* be an instance of a delete operation that deletes key k and that terminates after receiving only NACK messages on line 76. If k is not present in the list in the beginning of the execution interval of *op*, then *op* is linearized at that point and the claim holds.

Consider the case where k is included in the list when *op* is invoked. By observation of the pseudocode (lines 36-40), we have that when a server receives a delete request by a client, it traverses its local part of the list and deletes the element with key equal to k (line 37), if it is included in it. By further observation of the pseudocode (lines 74-80), we have that after c invokes *op*, it sends a delete request to all servers (line 74) and then awaits for a response from all of them (`do while` loop of lines 75-79). By assumption, all servers responds with NACK. Notice that this implies that between the execution of line 76 and 78 the element with key k is removed from the local list of s because of some other concurrent delete operation op' invoked by some client c'. By scrutiny of the pseudocode, we have that a server that deletes an element from its local list, does so on line 37, which occurs before the server sends a response to the delete request. By definition, then, op' is linearized at the point s executes line 37, before it sends an ACK message to c'. Since op' causes the element with key k to be removed from the local list of s between the execution of lines 76 and 78 by c, its linearization point is included in the execution interval of op. Since we place the linearization point of op right after the linearization point of op', the claim holds.

The argument is similar for when op is an instance of a search operation for key k that terminates after receiving a NACK message from all the servers on lines 63-67. □

Each server maintains a local variable *token* with initial value 0. Let some server s receive a message m in some configuration C. If the field `tk` of m is equal to TOKEN, we say that receives a token message. Observe that when s receives a token message (line 17), the value of *token* is set to s. Furthermore, when s executes line 27, where the value of *token* changes from s to $s+1$, s also sends a token message to $s+1$ (line 28). Notice that s can only reach and execute this line if the condition of the `if` clause of line 20 evaluates to `false`, i.e. if $token = s$. Then, the following holds:

Observation 84. *At each configuration in α, there is at most one server s for which the local variable token has the value s.*

This server is referred to as token server. By the pseudocode, namely the `if else` clause of lines 20, 23, and by line 24, the following observations holds.

Observation 85. *A server s performs insert operations on its local list in α only during those subsequences of α in which it is the token server.*

Each server maintains a local list collection, *llist*. By observation of the pseudocode, lines 9 and 12, we have that if an insert operation attempts to insert key k in either of the lists of a server s, but an element with that key already exists, then no second element for k is inserted and the operation terminates. Thus, the following holds:

Observation 86. *The keys contained in the list collection of s in any configuration C of α form a set.*

We denote this set by ll^s. By scrutiny of the pseudocode, we see that a new list object is allocated in *llist* each time a server receives a token message (lines 6-8). The new object is identified by the value of local variable *round*. By observation of the pseudocode, we further have that each time a server inserts a key into ll^s, it does so on the list object identified by *round* (line 24). We refer to this object as current list object. Then, based on lines 25-28 we have the following:

Observation 87. *A token message is sent from a server s to a server $((s+1) \mod \mathsf{NS})$ in some configuration C only if the current local list object of server s is full at C.*

Further inspection of the pseudocode shows that the local list object of a server is only accessed by the execution of line 9, 24, 32, or 37. From this, we have the following observation.

Observation 88. *If an operation op modifies the local list object of some server, then this occurs in the configuration in which op is linearized.*

Let C_i be the configuration in which the i-th linearization point in α is placed. Denote by α_i, the prefix of α which ends just after C_i and let L_i be the sequence of linearization points that is defined by α_i. Denote by S_i the set of keys that a sequential list contains after applying the sequence of operations that L_i imposes. Denote by $S_i = \epsilon$ the empty sequence (the list is empty).

Lemma 89. *Let k be the token server in some configuration C in which it receives a message m for an insert operation op with key k invoked by client c. Then at C, no element with key k is contained in the local list set of any other server $s \neq k$.*

Proof. By inspection of the pseudocode, when a client c sends a message m to some server either on line 45, line 62, line 74, or line 78, the *mloop* field of m is equal to `false`. This field is set to `true` when server $s_{\mathsf{NS}-1}$ executes line 45. Notice that in the configuration in which this line is executed by $s_{\mathsf{NS}-1}$, it is not the token server (otherwise the condition of line 20 would not evaluate to `true` and the line would not be executed).

Consider the case where m reaches a server s at some configuration C and let ll^s contain an element with key k in C. By inspection of the pseudocode (lines 9-12) we have that in that case, m is not forwarded to a subsequent server.

Furthermore, by lines 20-45, we have that if s is not the token server and not $s_{\mathsf{NS}-1}$, and provided that ll^s does not contain an element with key k, then s forwards m without modifying the *mloop* field. This implies that the *mloop* field of m is changed at most once in α from `false` to `true`, and that by server $\mathsf{NS}-1$, in a configuration C' in which k is not contained in $ll^{\mathsf{NS}-1}$. □

Lemma 90. *Let C_i, $i \geq 0$, be a configuration in α in which server s_{t_i} is the token server. Let ll_i^j be the local list set of server s_j, $0 \leq j < \mathsf{NS}$, in C_i. Then it holds that $S_i = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j$.*

Proof. We prove the claim by induction on i.

Base case $(\mathbf{i = 0})$. The claim holds trivially at C_0.

Hypothesis. Fix any $i > 0$ and assume that at C_i, it holds that $S_i = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j$. We show that the claim holds for $i+1$.

Induction step. Let op_{i+1} be the operation that corresponds to the linearization point placed in C_{i+1}. We proceed by case study.

Let op_{i+1} be an insert operation for key k. Assume first that the linearization point of op_{i+1} is placed at the execution of line 9 by $s_{t_{i+1}}$ for it. Notice that when this line is executed, k is searched for in the local list of $s_{t_{i+1}}$. Recall that, by the way linearization points are assigned, the client c that invoked op_{i+1} receives NACK as response. Notice also that $s_{t_{i+1}}$ sends NACK as a response to c if k is present in the local list of $s_{t_{i+1}}$, and thus $status_1 = \texttt{true}$. In that case, lines 20 to 29 are not executed, and therefore, no new element is inserted into the local list of $s_{t_{i+1}}$ (line 24). Thus $ll_{i+1}^{s_{t_{i+1}}} = ll_{i}^{s_{t_{i+1}}}$. By the induction hypothesis, $S_i = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j$. By Observation 88 it follows that for any other server s_j, where $j \neq t_{i+1}$, $ll_{i+1}^{s_j} = ll_i^{s_j}$ as well. Then, $\bigcup_{j=0}^{\mathsf{NS}-1} ll_{i+1}^j = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j$. Notice that since the server responds with NACK, $S_{i+1} = S_i$ by definition. Thus, $S_{i+1} = \bigcup_{j=0}^{\mathsf{NS}-1} ll_{i+1}^j$ and the claim holds.

Now, assume that op_{i+1} is linearized at the execution of line 24 by the token server for it. By the way linearization points are assigned, this implies that when this line is executed, $status_2 = \texttt{true}$, and the insertion of an element with key k into the local list of s_t was successful. This in turn implies that at C_{i+1}, $ll_{i+1}^{s_t} = ll_i^{s_t} \cup \{k\}$. By Observation 88 it follows that for any other server s_j, where $j \neq t_{i+1}$, $ll_{i+1}^{s_j} = ll_i^{s_j}$ as well. Notice that since the server responds with ACK, by definition the insertion is successful and thus $S_{i+1} = S_i \cup \{k\}$. Since by the hypothesis, $S_i = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j$, it holds that $S_{i+1} = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j \cup \{k\} = \bigcup_{j=0}^{\mathsf{NS}-1} ll_{i+1}^j$, thus, the claim holds.

Now consider that op_{i+1} is a delete operation for key k. Assume first that some server s_d responds with ACK, by executing line 38, to the client c that invoked op_{i+1}. Then op_{i+1} is linearized at the execution of this line by s_d. Notice that this line is executed by a server if $status_1 = \texttt{true}$, i.e. if the server was successful in locating and deleting an element with key k from its local list. Thus, $ll_{i+1}^{s_t} = ll_i^{s_t} \setminus \{k\}$. Furthermore, by definition, $S_{i+1} = S_i \setminus \{k\}$. By the induction hypothesis, $S_i = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j$ and since by Observation 88 no other modification occurred on the local list of some other server between C_i and C_{i+1}, it follows that $S_{i+1} = S_i \setminus \ \{k\} = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j \setminus \{k\} = \bigcup_{j=0}^{\mathsf{NS}-1} ll_{i+1}^j$.

Assume now that op_{i+1} is a delete operation for which no server responds with ACK to the invoking client. Recall that in this case, by definition, $S_{i+1} = S_i$. By inspection of the pseudocode, it follows that no server finds an element with key k in its local list when it is executing line 37 for op_{i+1}. We examine two cases: (i) either no element with key k is contained in any local list of any server in the beginning of the execution interval of op_{i+1}, or (ii) an element with key k is contained in the local list of some server s_d in the beginning of op_{i+1}'s execution interval, but s_d deletes it while serving a different delete operation op', before it executes line 37 for op_{i+1}.

Assume that case (i) holds. Then, the linearization point is placed in the beginning of the execution interval of op_{i+1}. Notice that in this case, the invocation (nor in fact the further execution) of op_{i+1} has no effect on the local list of any server. Thus, between C_i and C_{i+1} no server local list is modified and, by the induction hypothesis, the claim holds.

Assume now that case (ii) holds. By Lemma 83, we have that a concurrent delete operation op' removes the element with key k from the local list of s_d during the execution interval of op_{i+1}. By the assignment of linearization points, Observation 88 and Lemma 83, it further follows that $op' = op_i$. Notice that in this case (ii) also, op_{i+1} has no effect on the local list of any server. Thus, since by the induction hypothesis it holds that $S_i = \bigcup_{j=0}^{\mathsf{NS}-1} ll_i^j$, it also holds that $S_i = \bigcup_{j=0}^{\mathsf{NS}-1} ll_{i+1}^j$, and since $S_i = S_{i+1}$, the claim holds. Since a search operation does not modify the local list of any server, the argument is analogous as for the case of the delete operation. □

From the above lemmas and observations, we have the following.

Theorem 91. *The distributed unsorted list is linearizable. The insert operation has time and communication complexity $O(\mathsf{NS})$. The search and delete operations have communication complexity $O(1)$.*

4.2.4 A variation on the Unsorted List

In order to avoid the serial nature of Insert operations, we present a variant of the unsorted list implementation, in which insert operations avoid traversing the entire server ring by default.

Event-driven code for the server is presented in Algorithm 23. Each server s maintains a local list (*llist* variable) allocated for storing list elements, a *token* variable which indicates whether s currently holds the token, and a variable *round* to mark the ring round-trips the token has performed; *round* is initially 0, and is incremented after every transmission of the token to the next server. The pseudocode of the client is presented in Algorithm 24.

A client c sends an insert request for an element with key k to all servers in parallel and awaits a response. If any of the servers contains k in its local list, it sends ACK to c and the insert operation terminates. If no server finds k, then all reply NACK to c. In addition, the token server s_t encapsulates its id in the NACK reply. After that, c sends an insert request for k to s_t only. If s_t can insert it, it replies ACK to c. If k has in the meanwhile been inserted, s_t replies NACK to c. If s_t is no longer the token server, it forwards the request along the server ring until it reaches the current token server. Servers along the ring should check whether they contain k or not, and if some server does, then it replies NACK to c. Let s_t' be a token server that receives such a request. It also checks whether it contains k or not. If not, it attempts to insert k into its local list. Otherwise it replies NACK. When attempting to insert the element in the local list, it may occur that the allocated space does not suffice. In this case, the server forwards the request as well as the token to the next server in the ring, and increments the value of *round* variable. If the insertion at a token server is successful, the server then replies ACK to c.

Delete and Search operations are the same as in the previous version of the unsorted list.

4.3 A Distributed Sorted List

We briefly propose a modification of the distributed unsorted list of Section 4.2.3 that converts it into a sorted list. Clients use the same functions as for the unsorted list in order to access the sorted list. However, the servers have to perform more complex handling of messages and communication among them. Each server s has a memory chunk of predetermined size where it maintains a part of the implemented list so that all elements stored on server s_i have smaller

Algorithm 23 Events triggered in a server of the distributed unsorted list variant.

```
List $llist = \varnothing$;
int $my\_id$, $next\_id$, $token = 0$, $round = 0$;

a message $\langle op, cid, key, data, tk \rangle$ is received:
  switch ($op$) {
    case INSERT:
      if ($tk$ == TOKEN) {
        $token = my\_id$;
        allocate_new_memory_chunk($llist, round$);
      }
      $status_1$ = search($llist$, $key$);
      if ($tk == -2$) {
        if ($status_1$) {
          if ($token == my\_id$) send($cid$, $\langle$ACK, true$\rangle$);
          else send($cid$, $\langle$ACK, false$\rangle$);
        } else {
          if ($token == my\_id$) send($cid$, $\langle$NACK, true$\rangle$);
          else send($cid$, $\langle$NACK, false$\rangle$);
        }
      } else {
        if ($status_1$) send($cid$, NACK);
        else {
          if ($token \neq my\_id$) {
            $next\_id$ = get_next($my\_id$);
            send($next\_id$, $\langle op, cid, key, data, tk \rangle$);
          } else {
            $status_2$ = insert($llist, round, key, data$);
            if ($status_2$ == false) {
              $round++$;
              $token$ = get_next($my\_id$);
              send($token$, $\langle op, cid, key, data,$ TOKEN$\rangle$);
            } else send($cid$, ACK);
          }
        }
      }
      break;
    case SEARCH:
      $status_1$ = search($llist$, $key$);
      if ($status_1$) send($cid$, $\langle$ACK, $my\_id\rangle$);
      else send($cid$, $\langle$NACK, $my\_id\rangle$);
      break;
    case DELETE:
      $status_1$ = delete($llist$, $key$);
      if ($status_1$) send($cid$, ACK);
      else send($cid$, NACK);
      break;
  }
```

keys than those stored on server s_{i+1}, $0 \leq i < \mathsf{NS} - 1$. Because of this sorting property, an element with key k is not appended to the end of the list, so a token server is useless in this case. This is an essential difference with the unsorted list implementation.

Algorithm 24 Insert, Search and Delete operation for a client of the distributed list variant.

```
boolean ClientInsert(int $cid$, int $key$, data $data$) {
    boolean $status$;
    boolean $found$ = false;
    int $tid$;
    send_to_all_servers($\langle$INSERT$, cid, key, \perp, -2\rangle$);
    do {
      $\langle status, sid, is\_token\rangle$ = receive();
      if ($status$ == ACK) $found$ = true;
      if ($is\_token$) $tid = sid$;
      $c++$;
    } while ($c <$ NS);
    if ($found$ == true) return false;
    send($tid$, $\langle$INSERT$, cid, key, data, -1\rangle$);
    $status$ = receive();
    if ($status$ == NACK) return false;
    else return true;
  }
  boolean ClientSearch(int $cid$, int $key$) {
    int $sid$;
    int $c = 0$;
    boolean $status$;
    boolean $found$ = false;
    send_to_all_servers($\langle$SEARCH$, cid, key, \perp, -1\rangle$);
    do {
      $\langle status, sid\rangle$ = receive();
      if ($status$ == ACK) $found$ = true;
      $c++$;
    } while ($c <$ NS);
    return $found$;
  }
  boolean ClientDelete(int $cid$, int $key$) {
    int $sid$;
    int $c = 0$;
    boolean $status$;
    boolean $deleted$ = false;
    send_to_all_servers($\langle$DELETE$, cid, key, \perp, -1\rangle$);
    do {
      $\langle status, sid\rangle$ = receive();
      if ($status$ == ACK) $deleted$ = true;
      $c++$;
    } while ($c <$ NS);
    return $deleted$;
  }
```

4.3.1 Algorithm Description

Event-driven pseudocode for the server is presented in Algorithm 25 and 26. Similarly to the unsorted case, a client sends an insert request for key k to server s_0. The server searches its local part of the list for a key that is greater than or equal to k. In case that it finds such an element that is not equal to k, it can try to insert k to its local list, *llist*. More specifically, if the server has sufficient storage space for a new element, it simply creates a new node with key k and inserts it to the list. However, in case that the server does not have enough storage space, it tries to free it by forwarding a chunk of elements of *llist* to the next server. If this is possible, it serves the request. In case s_0 does not find a key that is greater than or equal to k in its *llist*, if forwards the message with the insert request to the next server, which in turn tries to serve the request accordingly.

Notice that this way, a request may be forwarded from one server to the next, as in the case of the unsorted list. However, for ease of presentation, in the following we present a static algorithm where this forwarding stops at $s_{\mathsf{NS}-1}$. In case that an element with k is already present in the *llist* of some server s of the resulting sequence, then s sends an NACK message to the client that requested the insert.

As in the case of the unsorted list, a client performs a search or delete operation for key k by sending the request to all servers. If not handled correctly, then the interleaving of the arrival of requests to servers may cause a search operation to "miss" the key k that it is searching, because the corresponding element may be in the process to be moved from one server to a neighboring one. In order to avoid this, servers maintain a sequence number for each client that is incremented at every search and delete operation. Neighboring servers that have to move a chunk of elements among them, first verify that the latest (search or delete) requests that they have served for each client have compatible sequence numbers and perform the move only then.

When an insert request for key k reaches a server s, s compares the maximal key stored in its local list to k. If k is greater than the maximal key and s is not $s_{\mathsf{NS}-1}$, the request must be forwarded to the next server (line 36). Otherwise, if k is to be stored on s, s checks if *llist* has enough space to serve the insert. If it does, s inserts the element and sends an ACK to the client (line 24-25). If s does not have space for inserts, the operation cannot be executed, hence s must check whether a chunk of its elements can be forwarded to the next server to make room for further inserts. To move a chunk, s calls `ServerMove()` (presented in Algorithm 26) (line 29). If `ServerMove()` succeeds in making room in s's *llist*, the insert can be accommodated (line 30). In any other case, s responds to the client with NACK (line 33).

A server process a search request as described for the unsorted list, but it now pairs each such request with a sequence number (line 41). Delete is processed by a server in a way analogous to search.

In order to move a chunk of *llist* to the next server, a server s_i invokes the auxiliary routine `ServerMove()` (line 29). `ServerMove()` sends a REQC message to server s_{i+1} (line 57). When s_{i+1} receives this request, it sends its client vector to s_i (line 7). Upon reception (line 58), s_i compares its own client vector to that of s_{i+1} and as long as it lags behind s_{i+1} for any client, it services search and delete requests until it catches up to s_{i+1} (lines 59-61). Notice that during this time, s_{i+1} does not serve further client request, in order allow s_i to catch up with it.

As soon as s_i and s_{i+1} are compatible in the client delete and search requests that they have served, s_i sends to s_{i+1} a chunk of the elements in its local list (lines 62-63) and awaits

Algorithm 25 Events triggered in a server of the distributed sorted list.

```
List $llist = \emptyset$; int $my\_id$, $next\_id$, $k_{max}$, $cv[MC]$, $nbr\_cv[MC]$
  data$[0 \ldots CHUNKSIZE]$ $chunk1$, $chunk2$
  boolean $status$ = false, $served$ = false

  a message $\langle op, cid, key, data \rangle$ is received:
    switch ($op$)
      case $REQC$:
        send($cid$, $cv$)
        $chunk2$ = receive($cid$)
        if (there is not enough free space in local list to fit the elements recorded in $chunk2$ ) then
          if ($my\_sid$ == NS $-$ 1) then $status$ = false
          else
            $chunk1$ = getChunkOfElementsFromLocalList($llist$)
            $status$ = ServerMove($next\_id$, $chunk1$)
        else $status$ = true
        if ($status$ == true) then
          insertChunkOfElementsInLocalList($llist, chunk2$)
          send($cid$, ACK)
        else send($cid$, NACK)
        break
      case $INSERT$:
        while ($served \neq$ true) do
          $k_{max}$ = find_max($llist$)
          if ($k_{max} > key$ and isFull($llist$) $\neq$ true) then
            status = insert($llist, key, data$)
            send($cid$, $status$)
            $served$ = true
          else if ($k_{max} > key$) then
            $chunk1$ = getChunkOfElementsFromLocalList($llist$)
            $status$ = ServerMove($next\_id$, $chunk1$)
            if ($status$ == true) then
              removeChunkOfElementsFromLocalList($llist$, $chunk1$)
            else
              send($cid$, NACK)
              $served$ = true
          else
            if ($my\_id \neq$ NS $-$ 1) then send($next\_id$, $\langle$INSERT$, cid, key, data \rangle$)
            else send($cid$, NACK)
            $served$ = true
        break
      case $SEARCH$:
        $cv[cid]++$
        $status$ = search($llist, key$)
        if ($status$ == false) then send($cid$, NACK))
        else send($cid$, ACK)
        break
      case $DELETE$:
        $cv[cid]++$
        $status$ = search($llist, key$)
        if ($status$ == true) then
          delete($llist$, $key$)
          send($cid$, ACK)
        else send($cid$, NACK)
        break
```

Algorithm 26 Auxiliary routine ServerMove for the servers of the distributed sorted list.

```
54 boolean ServerMove(int $cid$, data $chunk1$)
55     boolean $status$
56     data $chunk2$
57     send($next\_id$, $\langle REQC, cid, 0, \perp \rangle$)
58     $nbr\_cv$ = receive($next\_id$)
59     while (for any element $i$, $cv[i] < nbr\_cv[i]$) do
60       receiveMessageOfType($SEARCH$ or $DELETE$)
61       service request
62     $chunk2$ = getChunkOfElementsFromLocalList($llist$)
63     send($next\_id$, $chunk2$)
64     $status$ = receive($next\_id$)
65     if ($status$ == true) then
66       removeChunkOfElementsFromLocalList($llist$, $chunk2$)
67       insertChunkOfElementsInLocalList($llist$, $chunk1$)
68       return true
69     else return false
```

the response of s_{i+1}. We remark that in order to perform this kind of bulk transfer, as the one carried out between a server executing line 64 and another server executing line 8, we consider that remote DMA transfers are employed. This is omitted from the pseudocode for ease of presentation.

If s_{i+1} can store the chunk of elements, then it does so and sends ACK to s_i. Upon reception, s_i may now remove this chunk from its local list (line 30) and attempt to serve the insert request. Notice that if s_{i+1} cannot store the chunk of elements of s_i, then it itself initiates the same chunk moving procedure with its next neighbor (lines 11-13), and if it is successful in moving a chunk of its own, then it can accommodate the chunk received by s_i. Notice that in the static sorted list that is presented here, this protocol may potentially spread up to server $s_{\mathsf{NS}-1}$ (line 10). If $s_{\mathsf{NS}-1}$ does not have available space, then the moving of the chunk fails (line 32). The client then receives a NACK response, corresponding to a full list.

We remark that this implementation can become dynamic by appropriately exploiting the placement of the servers on the logical ring, in a way similar to what we do in the unsorted version.

4.4 Hierarchical Approaches and Experimental Evaluation

In the interest of supporting our theoretical view of the expected behavior of our implementations, we provide a summary of an experimental evaluation that was performed on them. We first sketch the hierarchical approach, a practical variation of the data structure implementations that does not affect correctness or data structure operation, but which is instead intended to provide good performance and scaling behavior. A deeper analysis of the obtained results is included in [FKKS15].

The hierarchical approach. This approach exploits the fast communication between the cores of the same island by organizing them virtually, as follows. In each island i, one process is designated as *island master* m_i. The remaining processes act as clients. The island master gathers requests from clients located on its island and forwards them to the appropriate data structure servers. To minimize the number of messages that are sent to those servers, m_i may batch several requests in one or more memory chunks (or fat messages). Then, m_i may send the memory chunks as messages or choose to transfer them to the servers using DMA. Conversely, a data structure server can either respond to clients individually, or batch the responses to the messages that pertain to requests initiated by clients of island i and send them to m_i (using DMA). If the latter option is followed, then m_i forwards each response to the appropriate client. Batching can be performed based on different rules for different data structures (and design approaches) to optimize performance. A straightforward example is the elimination that can be done in the case of a stack: to implement it, m_i may first collect a number of requests from the clients of its island, then perform elimination among the push and pop requests that it has received, and then batch the remaining requests into a fat message and forward it to a data structure server.

In case the architecture is fully non-cache coherent, then m_i does not gather the client requests from their shared memory module. Instead, the clients send their request to m_i as messages. A timeout delimits how long m_i waits for such messages before batching them and forwarding them to the data structure servers.

In partially cache-coherent architectures, an instance of a combining synchronization algorithm [FK12, HIST10] can be used in each island with all clients of the island participating to the protocol. A combining synchronization algorithm employs a list which stores requests of active clients from the island. After announcing its request by placing a node in the list, a client tries to acquire a global lock. The client that manages to acquire the lock, called the *combiner*, serves, in addition to its own request, other active requests recorded in the list. Thus, at each point in time, the combiner plays the role of the island master. When the island master receives (a batch of) responses from a server, it records each of them in the appropriate element of the request list to inform active clients of the island about the completion of their requests. In the meantime, each such client performs spinning (on its element) until either the response for its request has been fulfilled by the island master or the global lock has been released.

The simple one-level hierarchical scheme of island masters, described above, can easily be generalized to work for more layers of intermediate masters (in a tree-like fashion). The number of intermediate masters and the number of layers can be tuned for achieving the best performance.

Experimental evaluation. The stack and queue implementations were tested experimentally on the Formic-Cube [LKL$^+$12], which is a hardware prototype of a 512 core, non-cache-coherent machine. It consists of 64 boards with 8 cores each (for a total of 512 cores). Each core owns 8 KB of private L1 cache, and 256 KB of private L2 cache. None of these caches is hardware coherent. The boards are connected with a fast, lossless packet-based network forming a 3D-mesh with a diameter of 6 hops. Each core is equipped with its own local *hardware mailbox*, an incoming hardware FIFO queue, whose size is 4 KB. It can be written by any core and read by the core that owns it. One core per board plays the role of the island master(and could be one of the algorithm's servers), whereas the remaining 7 cores of the board serve as clients.

The experiments that were performed on the data structures consisted of executing 10^7 pairs of requests (push and pop or enqueue and dequeue) in total, increasing the number of cores for each experiment. To make the experiment more realistic, a random local work (up to 512 dummy loop iterations) was simulated between the execution of two consecutive requests by the same process. The average throughput of each of the algorithms was measured. The experiments were similar to those presented in [FK11, FK12, MS96].

The stack implementations that were used for the experiments were (i) a centralized stack, where only one core acts as server, while all remaining ones act as clients; (ii) a hierarchical version of the centralized stack where island masters do not batch messages; (iii) a hierarchical, centralized stack where messages are batched by the island masters; (iv) a hierarchical implementation of the directory-based stack; and (v) a hierarchical implementation of the token-based stack.

Since (as further experiments confirmed) the effect of the elimination technique is dominant, the implementations did not perform elimination, in order to give insights into their actual behavior. These experiments confirmed that the centralized implementation does not scale for more than 16 cores and showed the advantages of the hierarchical approach, since the implementations that incorporate it show better scaling. Interestingly, the directory-based stack outperforms the other implementations in experimental settings that use between 32 and 256 cores, showing a decline in scalability after that, when compared to the other implementations. This is attributed to the effect that the particular experimental setting has on the directory service: since pushes and pops are performed in pairs, frequently, the same key is assigned to two subsequent operations. This creates contention on the directory servers and degrades performance.

The queue implementations that were used for the experiments were (i) a centralized queue where only one core acts as server, while all remaining ones act as clients; (ii) a hierarchical version of the centralized queue; (iii) a hierarchical version of the directory-based queue; and (iv) a hierarchical version of the token-based queue. In these experiments, as well, the centralized queue exhibited the less scalability and was outperformed by its hierarchical version. The experiments further supported out theoretical perception that token-based implementations are nicely-suited for queues of relatively small expected size, offering an alternative that scales better than the centralized version as the number of cores increases up to 64. The hierarchical directory-based approach was the one that exhibited the best scalability characteristics also in the case of the queue implementations. We attribute this both to the fact that the synchronizer receives batched messages in this case, i.e. has less message handling to perform, and to the fact that the necessary computation on the synchronizer is minimal, while the actual insertion and deletion to the data structure takes place on the directory servers, allowing for more parallelization of operations.

By monitoring the amount of exchanged messages in each of the experiments, it was observed that they do not necessarily represent an indicative factor of actual performance in the proposed implementations. A low amount of messages may already saturate a server, it it means that the server has to dedicate significant effort in handling them. It seems more important to ensure good load balancing between servers if scalability is the desired outcome. A more in-depth analysis of those factors out of the scope of this thesis and is included in [FKKS15], together with graphical representations of the obtained resuls.

4.5 Related Work

We round up the context in which the work of this chapter was elaborated, by reviewing the related literature. Previous research results [KBI$^+$09, KPR$^+$08, LDK$^+$08, KW94] propose how to support dynamic data structures on distributed memory machines. Some are restricted on tree-like data structures, other focus on data-parallel programs, some favor code migration, whereas other focus on data replication. We optimize beyond simple distributed memory architectures by exploiting the communication characteristics of non cache-coherent multicore architectures. Some techniques from [KBI$^+$09, KPR$^+$08, LDK$^+$08] could be of interest though to further enhance performance and scalability in our implementations.

As in the shared memory context, in the distributed context also, transactional memory can be employed for the implementation of data structures. Distributed transactional memory (DTM) [BAC08, CRCR09, DPR15, GGT12, KAJ$^+$08, MMA06, SR11a, SR11c] is a generic approach for achieving synchronization, so data structures can be implemented on top of them. However, to do so, DTM systems introduce not only significant space overheads by maintaining metadata for every object and every transaction, but also performance overheads whenever reads from or writes to data items take place. Moreover, DTM requires the programmer to write the code in a transactional-compatible way. (When the transactions dynamically allocate data, as when they synchronize operations on dynamic data structures, compilers cannot detect all possible data races without trading performance, by introducing many false positives.) Our work is on a different avenue: towards providing a customized library of highly-scalable data structures, specifically tailored for non cache-coherent machines.

TM^{2}C [GGT12] is a DTM proposed for non cache-coherent machines. The paper presents a simple distributed readers/writers lock service where nodes are responsible for controlling access to memory regions. It also proposes two contention management (CM) schemes (Wholly and FairCM) that could be used to achieve starvation-freedom. However, in Wholly, the number of times a transaction T may abort could be as large as the number of transactions the process executing T has committed in past, whereas in FairCM, progress is ensured under the assumption that there is no drift [AW04, Lam78] between the clocks of the different processors of the non cache-coherent machine. Read-only transactions in TM^{2}C can be slow since they have to synchronize with the lock service each time they read a data item, and in case of conflict, they must additionally synchronize with the appropriate CM module and may have to restart several times from scratch. Other existing DTMs [BAC08, CRCR09, SR11b, SR11c, SR11d], not only impose common DTM overheads, but also may cause livelocks thus not providing strong progress guarantees.

The data structure implementations we propose do not cause any space overhead, read-only requests are fast, since all nodes that store data of the implemented structure search for the requested key in parallel, and the number of steps executed to perform each request is bounded. We remark that, in our algorithms, information about active requests is submitted to the nodes where the data reside, and data are not statically assigned to nodes, so our algorithms follow neither the data-flow approach [BF10, SR11d] nor the control-flow approach [BAC08, SR11b] from DTM research.

Distributed directory protocols [AGM10, AGM15, HS05, SB14, ZR09] have been suggested for locating and moving objects on a distributed system. Most of the directory protocols follow the simple idea that each object is initially stored in one of the nodes, and as the object moves around, nodes store pointers to its new location. They are usually based either on a spanning tree [DH98, ZR09] or a hierarchical overlay structure [AGM10, HS05, SB14]. Remarkably, among them, COMBINE [AGM10] attempts to cope with systems in which communication is not uniform. Directory protocols could potentially serve for managing objects in DTM. However, to implement a DTM system using a directory protocol, a contention manager must be integrated with the distributed directory implementation. As pointed out in [AGM15], this is not the case with the current contention managers and distributed directory protocols. It is unclear how to use these protocols to get efficient versions of the distributed data structures we present in this work.

Distributed data structures have also been proposed [AS03, GBHC00, HBC97, MNN01, AGS08] in the context of peer-to-peer systems or cluster computing, where dynamicity and fault-tolerance are main issues. They tend to provide weak consistency guarantees. Our work is on a different avenue.

Hierarchical lock implementations and other synchronization protocols for NUMA cache-coherent machines are provided in [DMS11, DMS12, FK12, HIST10, LDT$^+$12, LNS06, RH03]. We extend some of the ideas from these papers, and combine them with new techniques to get hierarchical implementations for a non cache-coherent architecture. Tudor et al. [DGT15] attempt to identify patterns on search data structures, which favor implementations that are portably scalable in cache-coherent machines. The patterns they came up with cannot be used to automatically generate a concurrent implementation from its sequential counterpart; they rather provide hints on how to apply optimizations when designing such implementations.

Hazelcast [Haz] is an in-memory data grid middleware which offers implementations for maps, queues, sets and lists from the Java concurrency utilities interface. These implementations are optimized for fault tolerance, so some form of replication is supported. Lists and sets are stored on a single node, so they do not scale beyond the capacity of this node. The queue stores all elements to the memory sequentially before flushing them to the datastore. Like Hazelcast, GridGain [Gri], an in-memory data fabric which connects applications with datastores, provides a distributed implementation of queue from the Java concurrency utilities interface. The queue can be either stored on a single grid node, or be distributed on different grid nodes using the datastore that exists below GridGain.

Chapter 5

Conclusion and Open Problems

We have presented a collection of algorithms that are meant to offer ease of programmability of current multi-core and of emerging future many-core architectures. Those algorithms include a transactional memory and a concurrent graph implementation that are designed assuming a cache-coherent shared memory system, as well as a collection of data structures that are implemented assuming a client-server model over a non-cache-coherent message-passing machine. In the present chapter, we discuss their use and implications.

5.1 Perspectives on Presented Algorithms

Previous chapters were restricted to detailing the defining aspects of our contributions. In order to round up our presentation, we make use of the following paragraphs in order to discuss interesting or important issues that our work has not yet covered.

WFR-TM: Practical Aspects and Future Work. We have presented WFR-TM, our implementation of a transactional memory algorithm, in a theoretical manner, in order to simplify its description and to focus on the correctness and progress properties that it guarantees. This has also allowed us to simplify the necessary formalism that was used in the proof. However, an additional concern that would arise for the implementation of a practical version of WFR-TM would be the optimization for performance. An important such optimization could be the use of a timestamping mechanism as the one that is presented in [DSS06, RFF06]. This mechanism can speed up the read-set validation process: currently, the validation process that we provide requires first obtaining and then de-referencing a pointer to a data item, in addition to the comparison to a local value. On the contrary, the use of timestamping would require the comparison of a local value – the timestamp of the read-set entry – to the current value of the global counter.

Another straight-forward optimization could be obtained by substituting the read-set locking that update transactions perform at commit-time. This locking provides a final read-set validation to determine whether the transaction must abort. Instead, explicit read-set validation could be carried out, once the transaction has locked its write-set. As a positive side effect, only the owner field of a `tvarrec` would then be required to be a `CAS` object, whereas the rest fields could be updated with simple writes.

Dense: Extending the Model. The novel model that we have used in order to present `Dense`, our concurrent graph implementation, is edge-oriented, meaning that operations on the graph do not create or remove vertices, but instead, access and affect the edges of the graph. Implicitly, this means that the resulting implementation assumes a fixed or at least, maximal number of possible vertices out of a specific vertex set. `Dense` operations are oblivious to the values or possible other attributes of those vertices.

Indeed, there are many applications that are concerned with just the connectivity of a graph and only require to access graph edges. Examples include garbage-collection – where objects are represented by graph nodes, while references to them are represented by graph edges – and graph-based video game navigation – where the edges of a graph represent walkable surfaces between obstacles, represented in turn by graph nodes. Nevertheless, an interesting line of future work is to extend the update and traversal capabilities of `Dense` to also provide information about the state or attributes of the visited vertices.

Distributed Data Structures: Perspectives. We have presented two different implementations for basic data structures, intended to facilitate programmability of future many-core architectures. The implementations could be utilized by runtimes of high-productivity languages ported to such architectures. Notably, our implementations correspond to several concurrent data structures supported in Java's concurrency utilities: Specifically, our implementations can be used (directly or with light modifications) in order to provide e.g. different kinds of queues, including static, dynamic, and synchronous ones. The queue implementation can be trivially adjusted to provide the functionality of *delay queues* (or *delay deques*) [Lea06]. Furthermore, our list implementations provide the functionality of sets.

The experimental evaluation shows the performance and scalability characteristics that some of the techniques provide, when used on FORMIC, a non cache-coherent hardware prototype. They also illustrate the scalability power of the hierarchical approach on that machine. While FORMIC is a many-core architecture emulator, we consider that it exhibits behaviors that actual machines will have. For this reason, we believe that the proposed data structure implementations will exhibit the same performance characteristics, if programmed appropriately, on prototypes with similar characteristics as FORMIC, like Tilera or SCC. We expect this also to be true for similar machines that may be commercially available in the future.

5.2 Future Prospects

The algorithms presented in this thesis have been designed following diverse models and assumptions. Nevertheless, their designs are governed by similar principles. Arguably, insights that are gained while studying one design may result useful for analyzing another. This can be nicely illustrated by comparing WFR-TM and Dense.

WFR-TM forces each update transaction to wait for each active read-only transaction it encounters, even if the read-set of the read-only transaction shares no t-variables with the update transaction's write-set. Recall that dynamic traversals in Dense exhibit behavior that is reminiscent of transactional memory. So, in order to avoid the unnecessary waiting of Update operations in the Dense implementation, several previous values of an edge are stored on the edge itself. In ensuring that transactions and dynamic traversals, i.e. complex read-only operations, are correct and wait-free, WFR-TM opts for incurring time overhead, while Dense opts for incurring some space overhead. We have thus here the opportunity to observe the trade-offs offered by different approaches.

With regards to transactional memory in particular, it would also be interesting not only to investigate trade-offs between design choices, but also between correctness and progress. Specifically, given the well-known impossibility result by Bushkov et al. [GK08], which states that a TM implementation cannot ensure that transactions are both opaque and wait-free, it remains to be seen whether TM algorithms with stronger progress properties than those ensured by WFR-TM can be designed by trading opacity with a weaker consistency condition. Conversely, impossibility results such as the aforementioned one [GK08] can help delimit the extent to which transaction-like complex operations can be provided for concurrent data structure implementations such as Dense.

While some data structures can be characterized as *regular* – as is the case with stacks and queues, which allow very specific access patterns – others, such as graphs or trees, exhibit a structural irregularity: This means that it is not easy to predict where and how updates will be made on the data structure. Contrast this with a FIFO queue: "where" on the data structures an update can be made is very specifically defined. Furthermore, depending on what end of the queue the modification is made on, the type of modification, i.e addition of an element (enqueue) or removal of an element (dequeue), is also very specifically defined. This is not the case in irregular data structures. Consequently, this makes the design of complex read-only operations difficult, given that a greater variety of modification patters will have to be taken into account if the read-only operation has to provide consistency. The implementation of Dense addresses this by taking such an irregular data structure and using a regularized representation of it, in order to provide dynamic traversals. An interesting question concerns whether the helping mechanism employed by Dense can be used as a generalized traversal technique. It would be interesting to explore what other irregular or regular data structures (trees, lists, queues, etc) can benefit from it.

Even though the distributed data structures are devised under a different framework than their concurrent equivalents, they provide the same functionality when seen from the programmer's level. For this reason, we consider that the concerns we exhibited previously, regarding read-only operations, also apply to them. Furthermore, the absence of cache-coherence is an additional factor of difficulty, apart from the process asynchrony. Generally speaking, the algorithms that we provide for either paradigm are rather *reader-friendly*: WFR-TM favors read-only transactions and `Dense` burdens the Update operations with the book-keeping of past edge values so that the dynamic traversals can easily construct a consistent view. Similarly, the distributed list implementations that we present provide a parallelized implementation of the Search operation. A step further in terms of functionality would be to enhance our distributed implementations with the capability of taking a total or partial snapshot of the data structure's state. Taking a cue from the `Dense` implementation and from standard practices in distributed computing, the use of vector clocks can be an interesting path to follow for that. A less complex read-only operation that is useful when it comes to list implementations in particular, is the range query. Interestingly, our sorted list implementation can be modified in order to provide it: by making the delete operation visit the servers one after another, i.e. by making it as slow as the insert operation, we could use the search operations in order to provide range queries. However, a more challenging question is how to accomplish this without sacrificing the efficiency that the current update implementation can provide.

The questions and concerns that are mentioned so far are an indicative subset of the challenges that the new machines pose, not only to the average programmer but also to the expert that is tasked with providing programming abstractions and data structure libraries. Arguably, while the architectures that we are concerned with become more and more pervasive, expertise in programming them will increase. The solutions we propose cannot claim to be the "silver bullet" to every kind of programming or performance problem. However, we consider that the presented algorithms can be a valuable contribution to making the programming of those architectures more accessible, while the required expertise is being acquired. We consider that this accessibility allows for the sufficient exploitation of the available computing power, even in the face of lacking programmer specialization. Furthermore, we hope that the study of the behavior of those algorithms can help shed light on more general concurrent computing issues and give insight into future architectures' behavior that will ultimately contribute to the better design of tailor-made applications for them.

If we return to the pebble-in-the-pond metaphor, we can state that while practices in hardware design change and evolve, the manner in which the programming paradigms adapt to them may continue to raise waves. Far from calming the surface, efforts such as ours may add to the turbulence. In fact, we do hope that they may contribute to the better understanding of the characteristics of the emerging hardware and in turn, indirectly contribute to the design of more efficient software. In the meanwhile, we hope that the implementations that we provide may assist the programmer in floating even while the waters are troubled.

Bibliography

[AAD$^+$93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. J. ACM, 40(4):873–890, September 1993.

[ABH$^+$01] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. Parallel Computing, 27(10):1279–1297, 2001.

[AGM10] Hagit Attiya, Vincent Gramoli, and Alessia Milani. A provably starvation-free distributed directory protocol. In Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), pages 405–419, New York, USA, September 2010.

[AGM15] Hagit Attiya, Vincent Gramoli, and Alessia Milani. Directory protocols for distributed transactional memory. In Rachid Guerraoui and Paolo Romano, editors, Transactional Memory. Foundations, Algorithms, Tools, and Applications, volume 8913 of Lecture Notes in Computer Science, pages 367–391. Springer International Publishing, 2015.

[AGR08] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 336–343, NY, USA, 2008. ACM.

[AGS08] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed b-tree. PVLDB, 1(1):598–609, 2008.

[AH12] Hagit Attiya and Eshcar Hillel. A single-version stm that is multi-versioned permissive. Theory of Computing Systems, 51(4):425–446, 2012.

[AHM09] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In Proceedings of the 21st Symposium on Parallelism in Algorithms and Architectures, SPAA '09, pages 69–78, New York, USA, 2009. ACM.

[AMS12] Yehuda Afek, Alexander Matveev, and Nir Shavit. Pessimistic software lock-elision. In Proceedings of the 26th International Symposium on Distributed Computing, DISC'12, pages 297–311, Berlin, Heidelberg, 2012. Springer-Verlag.

[And93] James H. Anderson. Composite registers. In Distributed Computing, pages 15–30, 1993.

[And94] JamesH. Anderson. Multi-writer composite registers. Distributed Computing, 7(4):175–195, 1994.

[AR93] Hagit Attiya and Ophir Rachman. Atomic snapshots in o(n log n) operations. In Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing, PODC '93, pages 29–40, New York, NY, USA, 1993. ACM.

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 384–393, Philadelphia, USA, 2003. SIAM.

[AW04] Hagit Attiya and Jennifer Welch. Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition). John Wiley Interscience, March 2004.

[BAC08] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 247–258, New York, USA, 2008.

[BF10] Annette Bieniusa and Thomas Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pages 1–12, Atlanta, Georgia, USA, April 2010.

[BGK12a] Victor Bushkov, Rachid Guerraoui, and Michal Kapalka. On the liveness of transactional memory. In Proceedings of the 31st ACM Symposium on Principles of Distributed Computing, PODC '12, pages 9–18, New York, USA, 2012. ACM.

[BGK12b] Victor Bushkov, Rachid Guerraoui, and Michal Kapalka. On the liveness of transactional memory. In Proceedings of the 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pages 9–18, NY, USA, 2012. ACM.

[CAB$^+$13] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua B. Fryman, Ivan Ganev, Roger A. Golliver, Rob C. Knauerhase, Richard Lethin, Benoît Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. Runnemede: An architecture for Ubiquitous High-Performance Computing.

In Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 198–209. IEEE Computer Society, 2013.

[CKK+08] Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In 37th International Conference on Parallel Processing (ICPP), pages 536–545, 2008.

[CRCR09] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable Distributed Software Transactional Memory. In Proceedings of the 15th Pacific Rim International Symposium on Dependable Computing (PRDC), Shanghai, China, November 2009.

[Dev93] Robert Devine. Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. In Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO), pages 101–114, 1993.

[DGT15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In Proceeding of the 20th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 631–644, Istanbul, Turkey, March 2015.

[DH98] Michael J. Demmer and Maurice Herlihy. The arrow distributed directory protocol. In Shay Kutten, editor, DISC, volume 1499 of Lecture Notes in Computer Science, pages 119–133. Springer, 1998.

[DMS11] David Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining NUMA locks. In Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 65–74, San Jose, CA, USA, June 2011.

[DMS12] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 247–256, New York, USA, 2012.

[DPR15] Aditya Dhoke, Roberto Palmieri, and Binoy Ravindran. On reducing false conflicts in distributed transactional data structures. In Proceedings of the 2015 International Conference on Distributed Computing and Networking (ICDCN), pages 8:1–8:10, Goa, India, January 2015.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In Proceedings of the 20th international conference on Distributed Computing, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[DT01] William J. Dally and Brian Towles. Route packets, not wires: On-chip inteconnection networks. In Proceedings of the 38th Annual Design Automation Conference, DAC '01, pages 684–689, New York, NY, USA, 2001. ACM.

[FC11] Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP '11, pages 179–188, New York, USA, 2011. ACM.

[FFMR10] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. IEEE Transactions on Parallel and Distributed Systems, 21:1793–1807, 2010.

[FFR08] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'08, pages 237–246, New York, USA, 2008. ACM.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. ACM Trans. Comput. Syst., 25(2), May 2007.

[FIKK15] Panagiota Fatourou, Mykhailo Iaremko, Eleni Kanellou, and Eleftherios Kosmas. Algorithmic techniques in stm design. In Transactional Memory. Foundations, Algorithms, Tools, and Applications, volume 8913, pages 101–126. Springer, 2015.

[FK11] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 325–334, New York, USA, 2011.

[FK12] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (SPAA), pages 257–266, 2012.

[FKKR14] Panagiota Fatourou, Eleni Kanellou, Eleftherios Kosmas, and Md Forhad Rabbi. WFR-TM: wait-free readers without sacrificing speculation of writers. In Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings, pages 420–436, 2014.

[FKKS15] Panagiota Fatourou, Nikolaos D. Kallimanis, Eleni Kanellou, and Christi Symeonidou. Distributed data structures for future many-core architectures. Technical Report TR-447, ICS-FORTH, April 2015.

[GBHC00] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI), pages 22–22, Berkeley, CA, USA, 2000. USENIX Association.

[GGT12] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. TM2C: A Software Transactional Memory for Many-cores. In Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys), pages 351–364, NY, USA, 2012.

[GHKR11] M. Gries, U. Hoffmann, M. Konow, and M. Riepen. Scc: A flexible architecture for many-core platform research. Computing in Science Engineering, 13(6):79–83, Nov 2011.

[GK08] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '08, pages 175–184, New York, USA, 2008. ACM.

[GKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A benchmark for software transactional memory. In Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[Gri] GridGain. Gridgain - in-memory data fabric. `http://www.gridgain.com/`.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In Proceedings of the 15th International Conference on Distributed Computing (DISC), pages 300–314, London, UK, 2001. Springer-Verlag.

[Haz] Hazelcast. Hazelcast the leading in-memory data grid. `http://hazelcast.com/`.

[HBC97] Victoria Hilford, Farokh B. Bastani, and Bojan Cukic. Eh* - extendible hashing in a distributed environment. In Proceedings of the 21 st International Computer Software and Applications Conference (COMPSAC), 1997.

[HDH$^+$10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In Proceedings of the International Solid-State Circuits Conference (ISSCC), pages 108–109, 2010.

[Her91] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., 13(1):124–149, January 1991.

[Hew13] HP ProLiant SL4500 server series overview. Technical report, Hewlett-Packard, 2013.

[HIST10] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In Proceedings of the 22nd Annual ACM

Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 355–364, New York,USA, 2010.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing, PODC'03, pages 92–101, New York, USA, 2003. ACM.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA), New York, USA, 1993.

[HS05] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In Proceedings of the 19th International Conference on Distributed Computing (DISC), pages 324–338. Springer Berlin Heidelberg, 2005.

[HS08] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 12(3):463–492, 1990.

[IR09] Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. In Distributed Computing, volume 5805, pages 142–156. Springer Berlin Heidelberg, 2009.

[KAJ$^+$08] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris C. Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In ICPP, pages 51–58. IEEE Computer Society, 2008.

[KBI$^+$09] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 3–14, New York, USA, 2009. ACM.

[KBLD08] Jakub Kurzak, Alfredo Buttari, Piotr Luszczek, and Jack Dongarra. The playstation 3 for high-performance scientific computing. Computing in Science and Engineering, 10(3):84–87, 2008.

[KK15] Nikolaos D. Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In Principles of Distributed Systems - 19th International Conference, OPODIS 2015, 2015.

[KP11] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 223–234, NY, USA, 2011. ACM.

[KP12] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. SIGPLAN Not., 47(8):141–150, February 2012.

[KPR$^+$08] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In Proceeding of the 13th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008.

[KR15] Petr Kuznetsov and Srivatsan Ravi. On partial wait-freedom in transactional memory. In Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN '15, pages 10:1–10:9, New York, NY, USA, 2015. ACM.

[KW94] Brigitte Kröll and Peter Widmayer. Distributing a search tree among a growing number of processors. In Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, pages 265–276, New York, USA, 1994.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7), 1978.

[LDK$^+$08] D.B. Larkins, J. Dinan, S. Krishnamoorthy, S. Parthasarathy, A. Rountev, and P. Sadayappan. Global trees: A framework for linked data structures on distributed memory parallel systems. In International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–13, Nov 2008.

[LDT$^+$12] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In Proceedings of the 2012 USENIX Conference on Annual Technical Conference, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[Lea06] Douglas Lea. Concurrent Programming in Java(TM): Design Principles and Patterns (3rd Edition). Addison-Wesley Professional, 2006.

[LKL$^+$12] Spyros Lyberis, George Kalokerinos, Michalis Lygerakis, Vassilis Papaefstathiou, Dimitris Tsaliagkos, Manolis Katevenis, Dionisios Pnevmatikatos, and Dimitris Nikolopoulos. Formic: Cost-efficient and scalable prototyping of manycore architectures. In Proceedings of the 2012 IEEE 20th International Symposium on Field-

Programmable Custom Computing Machines (FCCM), pages 61–64, Washington, DC, USA, 2012. IEEE Computer Society.

[LNS06] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A Hierarchical CLH Queue Lock. In WolfgangE. Nagel, WolfgangV. Walter, and Wolfgang Lehner, editors, Euro-Par 2006 Parallel Processing, volume 4128 of Lecture Notes in Computer Science, pages 801–810. Springer Berlin Heidelberg, 2006.

[Lyn96] Nancy A. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS), 9(1):21–65, 1991.

[MMA06] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 198–208, New York, USA, 2006. ACM.

[MNN01] Richard P. Martin, Kiran Nagaraja, and Thu D. Nguyen. Using distributed data structures for constructing cluster-based services. In Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY), 2001.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 267–275, NY, USA, 1996. ACM.

[MS10] Ross McIlroy and Joe Sventek. Hera-jvm: a runtime system for heterogeneous multi-core architectures. In Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 205–222, 2010.

[MS12] Alexander Matveev and Nir Shavit. Towards a fully pessimistic stm model. In 7th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2012.

[NDB$^+$14] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. In Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, pages 3–18. ACM, 2014.

[NGF08] Albert Noll, Andreas Gal, and Michael Franz. CellVM: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. In Workshop on Cell Systems and Applications. Citeseer, 2008.

[NP11] Donald Nguyen and Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 333–344, 2011.

[Ora] Oracle. Java utilities library. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. Journal of the ACM, 26(4):631–653, oct 1979.

[PBBO12] Aleksandar Prokopec, Nathan G. Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. SIGPLAN Not., 47(8):151–160, Feb 2012.

[PBLK11] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. Smv: Selective multi-versioning stm. In David Peleg, editor, DISC, volume 6950 of Lecture Notes in Computer Science, pages 125–140. Springer-Verlag, 2011.

[PFK10] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In Proceedings of the 29th ACM Symposium on Principles of Distributed Computing, PODC '10, pages 16–25, New York, USA, 2010. ACM.

[PMP12] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A system for synthesizing concurrent graph programs. SIGPLAN Not., 47(10):375–394, October 2012.

[PT13] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In Distributed Computing, volume 8205, pages 224–238. Springer Berlin Heidelberg, 2013.

[RFF06] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In Proceedings of the 20th International Symposium on Distributed Computing, DISC'06, pages 284–298, Berlin Heidelberg, 2006. Springer-Verlag.

[RH03] Zoran Radovic and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA), pages 241–252, 2003.

[SB14] Gokarna Sharma and Costas Busch. Distributed transactional memory for general networks. Distrib. Comput., 27(5):329–362, October 2014.

[Sha14] Omid Shahmirzadi. High-Performance Communication Primitives and Data Structures on Message-Passing Manycores. PhD thesis, École Polytechnique fédérale de Lausanne (EPFL), 2014. n° 6328.

[SLS06] William N. Scherer III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. In Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP), NY, USA, 2006. ACM.

[SR11a] Mohamed Saad and Binoy Ravindran. Supporting STM in Distributed Systems: Mechanisms and a Java Framework. In 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2011.

[SR11b] Mohamed Saad and Binoy Ravindran. Transactional forwarding algorithm. Technical report, Virginia Tech, 2011.

[SR11c] Mohamed M. Saad and Binoy Ravindran. HyFlow: A High Performance Distributed Software Transactional Memory Framework. In Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC), pages 265–266, New York, USA, 2011.

[SR11d] Mohamed M. Saad and Binoy Ravindran. Snake: Control Flow Distributed Software Transactional Memory. In Proceedings of 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), pages 238–252, 2011.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 204–213, New York,USA, 1995. ACM.

[TBKP12] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 309–310, NY, USA, 2012. ACM.

[TMG$^+$09] Fuad Tabba, Mark Moir, James R. Goodman, Andrew W. Hay, and Cong Wang. Nztm: Nonblocking zero-indirection transactional memory. In Proceedings of the 21st Symposium on Parallelism in Algorithms and Architectures, SPAA '09, pages 204–213, New York, USA, 2009. ACM.

[Vaj11] Andras Vajda. Introduction. In Programming Many-Core Chips, pages 1–7. Springer US, 2011.

[vB09] C. H. (Kees) van Berkel. Multi-core for mobile phones. In Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09, pages 1260–1265, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[YC97] Weimin Yu and Alan Cox. Java/dsm: A platform for heterogeneous computing. Concurrency: Practice and Experience, 9(11):1213–1224, 1997.

[ZR09] Bo Zhang and Binoy Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS), pages 48–53, Nîmes, France, December 2009.

[ZWL02] Wenzhang Zhu, Cho-Li Wang, and Francis CM Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In 2002 IEEE International Conference on Cluster Computing., pages 381–388. IEEE, 2002.

List of Algorithms

List of Tables

Résumé

À une époque où les processeurs sont omniprésents, les programmer correctement et efficacement est un enjeu important. Les tendances récentes dans la conception de matériel montrent une évolution vers l'intégration de plusieurs cœurs de traitement sur une seule puce. Actuellement, la majorité de ces machines sont fondées sur une mémoire partagée avec cohérence de caches. Des prototypes intégrant de grandes quantités de coeurs, reliés par une infrastructure de transmission de messages, indiquent que, dans un proche avenir, les architectures de processeurs vont probablement avoir ces caractéristiques. Ces deux tendances – mémoire partagé ou transmission de messages – exigent que les processus s'exécutent en parallèle et rendent la programmation concurrente nécessaire. Cependant, la difficulté inhérente du raisonnement sur la concurrence peut avoir un effet négatif: celui de rendre ces nouvelles architectures de processeurs difficiles à programmer.

La programmation concurrente est actuellement considérée comme une discipline réservée aux experts qui maîtrisent la gestion des accès aux ressources partagées. Ce genre de gestion peut exiger des aptitudes différentes, car, selon l'application à programmer, il peut être plus important d' éviter les mauvais effets que les mémoires cachés ont sur la performance, ou bien de pouvoir résister aux crash, ou encore de savoir utiliser de verrous correctement. Pour le cas ou la transmission de messages est utilisée, il peut être plus important de minimiser la totalité de messages qui circulent ou bien d'adapter leur montant à l'architecture de la machine utilisée. Afin de résoudre ce type de problèmes, nous explorons trois approches ayant pour but de faciliter la programmation concurrente.

Notre première approche est fondé sur la mémoire transactionnelle (`TM`), un paradigme de programmation concurrente très prometteur. Une `TM` utilise des transactions afin de synchroniser l'accès aux données partagées, appelées aussi variables transactionnelles. Une transaction peut soit terminer (commit), rendant visibles ses modifications des variables transactionnelles, soit échouer (abort), annulant toutes ses modifications. Étant donné que les échecs de transactions sont considérés comme une perte de puissance de calcul, un important sujet de recherche sur le domaine des `TM` est de savoir comment les minimiser. Typiquement, une transaction peut échouer dans des cas ou elle a un conflit avec une autre transaction. Un conflit se produit quand deux (ou plus) transactions essayent d'accéder à la même variable transactionnelle et qu'au moins une de ces transactions essaie de la modifier. Dans des cas comme celui-ci, l'échec protège la cohérence des données partagées, mais les échecs diminuent les performances si ils sont trop nombreux.

Idéalement, nous voudrions avoir des implémentations de `TM` qui garantissent que toutes les transactions terminent. Pourtant, des résultats théoriques montrent que ce n'est pas possible. Cela pose une restriction importante, surtout quand cela touche aux transactions en lecture seule, c'est à dire, les transactions qui ne modifient pas des variables transactionnelles. Dans de nombreuses applications la majorité des transactions sont en lecture seule, comme par exemple celles ou les transactions sont utilisées pour convertir une implémentation séquentielle de

structure de données en implémentation concurrente. Nous voudrions avoir des transactions en lecture seule qui sont légères à la fois en méta-données et synchronisation.

Nous proposons WFR-TM, un algorithme qui tente d'offrir ces propriétés en combinant des caractéristiques désirables des TM optimistes et pessimistes. Dans une TM pessimiste, aucune transaction n'échoue jamais; néanmoins, pour cela les algorithmes existants utilisent des verrous afin d'exécuter de manière séquentielle les transactions qui contiennent des opérations d'écriture. Cela diminue le degré de parallélisme qui peut être atteint par l'application. À l'inverse, les algorithmes TM optimistes exécutent toutes les transactions en parallèle mais ne les terminent que si elles n'ont pas rencontré de conflit au cours de leur exécution. WFR-TM fournit des transactions en lecture seule qui sont wait-free, avec l'avantage supplémentaire de ne jamais exécuter d'opérations de synchronisation coûteuse (comme par exemple, CAS, LL/SC, etc). Ce résultat est obtenu sans sacrifier le parallélisme entre les transactions d'écriture.

Dans WFR-TM, chaque transaction d'écriture détecte les transactions en lecture seule concurrentes et attend qu'elles terminent avant de terminer elle même afin d'éviter des conflits. Ce mécanisme permet aux transactions de lecture seule de toujours terminer. Par contre, lorsqu'une transaction d'écriture détecte un conflit avec une autre transaction d'écriture, elle peut échouer. Dans ce cas, l'approche optimiste est utilisé pour la synchronisation entre les transactions d'écriture (alors que l'approche pessimiste est utilisé pour synchroniser les transactions d'écriture avec les transactions de lecture). Afin d'offrir au programmeur une implémentation correcte de TM, ce travail contient une démonstration formelle qui prouve que WFR-TM offre des transactions de lecture seule qui terminent toujours, et que l'algorithme satisfait la condition de cohérence *opacité*, qui exige qu'aucune transaction ne lise les valeurs d'un état global incohérent.

La mémoire transactionnelle est un outil facilitant la programmation de modèles génériques de coordination entre des processus qui utilisent une mémoire partagée. Les structures de données concurrentes sont une façon plus spécialisée de faire la même chose. Actuellement, on peut trouver plusieurs implémentations concurrentes de structures de donnés comme par exemple de piles, de files et de listes et la recherche dans cet direction est très active. Une telle implémentation concurrente fournit des algorithmes qui fournissnet les opérations basiques de la structure, mais qui prennent également en compte le fait que plusieurs processus peuvent y accéder en parallèle, et s'occupent de leur synchronisation. Ici, nous sommes intéressés par des structures de données offrant des fonctionnalités améliorées en fournissant des opérations complexes en lecture seule. Contrairement aux opérations basiques d'une structure de données, les opérations complexes en lecture seule sont utiles quand l'objectif est d'obtenir un snapshot, c'est à dire, une vue cohérente, partielle ou totale, de l'état de la structure. Dans le contexte séquentiel, obtenir un snapshot est trivial, mais ce n'est pas le cas dans le contexte concurrent: étant donné que plusieurs processus accèdent à la structure en parallèle, il peut arriver que l'un d'entre eux fasse des modifications sur la structure alors qu'un autre essaie d'obtenir un snapshot, ce qui peut causer des problèmes de cohérence.

Comme solution aux problèmes de ce type, dans ce travail, nous présentons également une implémentation concurrente de graphe qui fournit une opération complexe en lecture seule. Les graphes sont des structures de données polyvalentes qui permettent la mise en oeuvre d'une variété d'applications, comme par exemple les simulations scientifiques ou les jeux vidéo. Cependant, bien que des structures de données tel que des files, des piles, et des arbres aient été largement étudiés et adaptées en versions concurrentes, des applications multi-processus qui utilisent des graphes utilisent encore largement des versions séquentielles où les accès aux données partagées sont synchronisés par l'utilisation de verrous, ce qui entraine des pertes de performance. Nous introduisons un nouveau modèle de graphes concurrents, permettant l'ajout ou la suppression de n'importe quel arc du graphe, ainsi que la traversée atomique d'une partie (ou de l'intégralité) du graphe. Nous présentons également `Dense`, une implémentation concurrente de graphes visant à atténuer les deux inconvénients d'implémentation susmentionnés.

`Dense` offre la possibilité d'effectuer un snapshot partiel d'un sous-ensemble du graphe défini dynamiquement. Des modifications et des traversées atomiques peuvent se faire en parallèle sans violer la cohérence du snapshot obten. Comme le sous-ensemble à visiter est défini dynamiquement, le modèle proposé ressemble à la mémoire transactionnelle. Ayant cette versatilité, il peut être utilisé pour implémenter plusieurs modèles de traversée très variés. Pour autant, les similarités avec la mémoire transactionnelle n'incluent pas les échecs de terminaison qui sont associés aux transactions. Cette caractéristique est importante car elle aide à assurer que les opérations du graphe satisfassent le critère de cohérence *linéarisabilité* et qu'elles soient *wait-free*, c'est à dire qu'elles terminent toujours.

Enfin, nous ciblons les futures architectures et étudions des techniques générales pour implémenter des structures de données distribuées en supposant qu'elles seront utilisées sur des architectures many-core, qui n'offrent qu'une cohérence partielle de caches, voir pas de cohérence du tout. Dans l'intérêt de la réutilisation du code et afin d'offrir un paradigme commun, il existe depuis quelques temps une tentative d'adaptation des environnements d'exécution de logiciel, initialement prévus pour mémoire partagée, à des machines sans cohérence de caches. Un exemple notable est la JVM, l'environnement d'exécution de Java. Les implementations de structure de données distribuées sont des composantes importantes des bibliothèques incorporées dans ces environnements. Afin de contribuer à cet effort, nous présentons différentes implémentations de piles, de files et de listes.

Nous nous concentrons sur deux techniques. Nous présentons d'abord une approche fondée sur le répertoire. Avec cette approche, les éléments qui composent la structure de données sont stockés dans un répertoire distribué. Dans cette technique, un serveur de synchronisation agit comme coordinnateur qui indique où les données doivent être stockés et d'où elles peuvent être récupérées. Cette approche permet d'atteindre l'équilibrage de charge dans les situations où la structure de données est grande. Cependant, la manière dont ceci est réalisé, reste "caché" pour le programmeur.

Le même est vrai pour l'approche basée sur les jetons, la deuxième technique de conception que nous présentons. Dans nos algorithmes à base de jeton, les éléments qui constituent la structure de données sont stockés dans les modules de mémoire de certains de serveurs disponibles.

Ces serveurs forment un anneau. L'un d'entre eux est désigné comme le serveur du jeton et, initialement, le stockage et la récupération des éléments de la structure de données a lieu sur son module de mémoire. Si le module de mémoire de ce serveur devient vide ou se remplit complètement, le jeton est envoyé au serveur suivant ou précédent dans l'anneau. Cette approche exploite la localité des données et pour cette raison elle est mieux adaptée pour les cas où la taille de la structure de données est modérée.

Dans le but de rendre les architectures many-core plus accessible aux programmeurs qui sont habitués à la programmation séquentielle, un avantage supplémentaire de nos implémentations est qu'elles peuvent faciliter la réutilisation des applications qui étaient initialement conçues pour la mémoire partagée. Les algorithmes que nous présentons sont conçue comme une étape vers la création de bibliothèques de structures de données adaptées aux infrastructures de transmission de messages. Des applications à mémoire partagée qui se fondent sur des bibliothèques équivalentes de structures de données pourraient être portées à des environnements qui utilisent la transmission de messages simplement en substituant une bibliothèque par une autre. Beaucoup d'efforts de recherche ont été consacrés à la mise en œuvre des environnements distribués d'exécution pour les langages à forte productivité, tels que Java par exemple. Bien que ces implémentations supposent des mémoires cachés sans cohérence, ils maintiennent néanmoins l'abstraction de la mémoire partagée pour le programmeur. Les structures de données que nous fournissons correspondent à plusieurs des structures de données inclues dans des bibliothèques concurrentes de Java, et pourraient être utilisées pour les remplacer.

Abstract

In an era where processors are ubiquitous, programming them correctly and efficiently is an important issue. Recent trends in hardware design mark a shift towards integrating several processing cores on a single chip. Currently, a majority of those machines relies on shared, cache-coherent memory. Prototypes that integrate large amounts of cores, connected through a message-passing substrate, indicate that architectures of the near future may have these characteristics. Either of those tendencies requires that processes execute in parallel, making concurrent programming a necessary tool. The inherent difficulty of reasoning about concurrency, however, may lead to the adverse effect of rendering the new processor architectures hard to program. In order to deal with issues such as this, we explore a threefold approach to providing ease of programmability.

The first approach employs transactional memory (TM), a promising concurrent programming paradigm. TM employs transactions in order to synchronize the access to shared data, known as data items or transactional variables. A transaction may either commit, making its updates to transactional variables visible, or abort, discarding its updates. We propose WFR-TM, an implementation that attempts to combine desirable characteristics of pessimistic and optimistic TM. In a pessimistic TM, no transaction ever aborts; however, in order to achieve that, existing TM algorithms employ locks in order to execute update transactions sequentially, decreasing the degree of achieved parallelism. Contrary to that, optimistic TM algorithms execute all transactions concurrently and commit them if they have encountered no conflict during their execution. WFR-TM provides read-only transactions that are wait-free, with the added benefit of never executing expensive synchronization operations (like CAS, LL/SC, etc). This is achieved without sacrificing the parallelism between update transactions. As such, the optimistic approach is used for the synchronization among update transactions, while they synchronize with read-only transactions pessimistically.

Transactional memory is a tool that is meant to facilitate the programmability of generic patterns of coordination among processes using a shared-memory. More specialized manners of process coordination and shared data organization may involve concurrent data structure implementations. Exemplifying that, we present a concurrent graph implementation. Graphs are versatile data structures that allow the implementation of a variety of applications, such as computer-aided design and manufacturing, video gaming, or scientific simulations. However, although data structures such as queues, stacks, and trees have been widely studied and implemented in the concurrent context, multi-process applications that rely on graphs still largely use a sequential implementation where accesses are synchronized through the use of global locks or partitioning, thus imposing serious performance bottlenecks. We introduce an innovative concurrent graph model that provides addition and removal of any edge of the graph, as well as atomic traversals of a part (or the entirety) of the graph. We further present Dense, a concurrent graph implementation that aims at mitigating the two aforementioned implementation drawbacks. Dense achieves wait-freedom by relying on light-weight helping and provides the inbuilt capability of performing a partial snapshot on a dynamically determined subset of the

graph.

We finally aim at predicted future architecture and study general techniques for implementing distributed data structures assuming they have to run on many-core architectures that offer either partially cache-coherent memory or no cache coherence at all. In the interest of code reuse and of a common paradigm, there is recent momentum towards porting software runtime environments, originally intended for shared-memory settings, onto non-cache-coherent machines. JVM, the runtime environment of the high-productivity language Java, is a notable example. Concurrent data structure implementations are important components of the libraries that environments like these incorporate. With the goal of contributing to this effort, we present different implementations of stacks, queues, and lists.