ADVANCED CONCURRENCY TECHNIQUES

FOR CONCURRENT DATA STRUCTURES

by

Shreyas Sanjeev Gokhale

APPROVED BY SUPERVISORY COMMITTEE:

Neeraj Mittal, Chair

R. Chandrasekaran

Ivor Page

S. Venkatesan

This dissertation

is dedicated to my parents,

for their constant support, guidance,

inspiration, and a strong belief in me.

ADVANCED CONCURRENCY TECHNIQUES FOR CONCURRENT DATA STRUCTURES

by

SHREYAS SANJEEV GOKHALE, BE, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 2019

ACKNOWLEDGMENTS

First and foremost, I would like to thank my PhD advisor, Dr. Neeraj Mittal, for his guidance and support throughout my PhD journey. He has constantly inspired me to dig deeper and overcome obstacles in research. I have learned a lot under his mentorship and gained knowledge which will undoubtedly help me in my future career.

I am thankful to Dr. R. Chandrasekaran who has always been kind and helpful to me.

I would also like to thank Dr. Ivor Page for his helpful suggestions during my dissertation proposal and Dr. S. Venkatesan for his advice and suggestions in general.

During this long journey, I have had the pleasure of meeting and working with many people who have helped me get through to the end. They include Joseph Beshay, K. Alex Mills, Kenneth Platz, Ketan Joshi, Swair Shah, Vishal Karande, and my fellow PhD students: Hemant Malik, Omer Ozarslan, Partha De, Sahil Dhoked, Sarat Chandra Varanasi, and Trusit Shah.

June 2019

ADVANCED CONCURRENCY TECHNIQUES

FOR CONCURRENT DATA STRUCTURES

Shreyas Sanjeev Gokhale, PhD
The University of Texas at Dallas, 2019

Supervising Professor: Neeraj Mittal, Chair

Concurrent algorithms have gained importance as multi-core machines have become more ubiquitous. Several techniques are employed to enable the construction of such algorithms. We present algorithms for the *Group Mutual Exclusion (GME)* problem which can be used as an advanced concurrency technique to increase the performance of software built on concurrent data structures.

The group mutual exclusion (GME) problem is a generalization of the classical mutual exclusion problem in which every critical section is associated with a *type* or *session*. Critical sections belonging to the same session can execute concurrently, whereas critical sections belonging to different sessions must be executed serially. The well-known read-write mutual exclusion problem is a special case of the group mutual exclusion problem. We present a new GME algorithm for an asynchronous shared-memory system under the Cache-Coherent model that, in addition to satisfying lockout freedom, bounded exit and concurrent entering properties, has $O(1)$ step-complexity when the system contains no conflicting requests *as well as* $O(1)$ space-complexity per GME object when the system contains sufficient number of GME objects. We also present a GME algorithm for the Distributed Shared Memory model that satisfies above properties and has optimal Remote-Memory-Reference complexity. To the best of our knowledge, no existing GME algorithm has $O(1)$ step-complexity for concurrent entering for either model. The Remote-Memory-Reference complexity of a request for cache-coherent model is only $O(\dot{c})$ in the amortized case, where $\dot{c}$

denotes the point contention of the request. Experiments indicate that our GME algorithm vastly outperforms well-known GME algorithms in most, if not all, cases.

We also present a lock-based concurrent algorithm for a strictly-balanced red black tree data structure. Our algorithm can be implemented on hardware directly without requiring any additional system support such as transactional memory. We also make use of several optimizations to improve the performance of our tree. Our experimental results indicate that our lock-based algorithm for a strictly balanced binary search tree outperforms other relaxed balanced trees for read-dominated workloads.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The field of concurrent computing has gained in importance due to changes seen in the processor manufacturing industry. Up until the mid-2000s, most processors consisted of a single, fast execution unit. Since around the mid-2000s, Dennard scaling [24] no longer continued to hold due to physical limitations of hardware (greater power requirements with accompanying excessive heat). As a result, major chip manufacturers shifted their focus from increasing the speed of individual processors to increasing the number of cores on a chip. To exploit these multi-core architectures, programs must be executed in a concurrent manner. The algorithms must be designed with a large number of processes and their concurrent accesses to shared data must be synchronized to prevent inconsistencies. This shared data typically takes the form of a data structure. A concurrent data structure is one in which multiple processes may need to operate on overlapping regions of the data structure at the same time. Contention between different processes must be managed in such a way that all operations complete correctly and leave the data structure in a valid state.

Concurrency often adds significant complexity to an algorithm, requiring concurrency control such as mutual exclusion to avoid problems such as race conditions. Concurrency is most often managed through locks. A process holding a lock is guaranteed exclusive access to the data structure until it releases the lock. In a coarse-grained lock based algorithm, a single lock governs large portions, even possibly the entire data structure. Fine-grained locks allow greater concurrency compared to coarse-grained locks, as they govern smaller portions of the data structure. Locking based methods make it easier to perform (potentially conflicting) updates to the data structure because they are implemented in a mutually exclusive manner and hence serialized. This also makes it easier to design, implement, and debug lock-based data structures and reason about their correctness compared to their lock-free counterparts. Furthermore, David and Guerraoui recently observed that lock-based data structures can be practically wait-free [18], the strongest progress guarantee that can be provided by a non-blocking algorithm. Wait-freedom guarantees that every

process completes its operation in a finite number of steps. Lock-based algorithms for concurrent versions of many important data structures have been developed including linked lists [33, 35], queues [53, 35], hash tables [39, 20, 47, 36], skip lists [48] and search trees [5, 46, 9, 3, 6, 16, 10]).

However, locks are blocking; while a process is holding a lock, no other process can access the portion of the data structure protected by the lock. If a process stalls while it is holding a lock, then the lock may not be released for a long time. This may cause other processes to wait on the stalled process for extended periods of time. As a result, lock-based implementations of concurrent data structures are vulnerable to problems such as deadlock, priority inversion, and convoying [35].

Non-blocking algorithms avoid the pitfalls of locks by using special (hardware-supported) read-modify-write instructions such as load-link/store-conditional (LL/SC) [31] and compare-and-swap (CAS) [35]. These algorithms can provide stronger progress guarantees since a stalled process cannot block other processes. Non-blocking algorithms for many data structures such as queues, stacks, linked lists, hash tables, search trees and tries have been developed (e.g., [31, 29, 51, 33], [23, 61, 6, 26, 19], and [58, 8, 38, 54, 55]). However, non-blocking algorithms are much harder to design, implement, and debug compared to their blocking counterparts.

Our main goal is to design efficient, scalable concurrent objects. This requires use of building blocks needed to design the objects together with concurrent data structures that make up the main component of the concurrent object. The building blocks are in the form of concurrency techniques used to implement these concurrent data structures. This work is divided into two parts: Group Mutual Exclusion as an advanced concurrency technique to further improve the performance of certain lock based concurrent data structures and a lock based algorithm for a Concurrent Red Black Tree data structure.

The main focus of this work is the Group Mutual Exclusion (GME) problem. The GME problem is a generalization of the classical mutual exclusion (ME) problem in which every critical section is associated with a *type* or *session* [43]. Critical sections belonging to the same session can execute concurrently, whereas critical sections belonging to different sessions must be executed serially. The GME problem models situations in which a resource may be accessed at the

same time by processes of the same group, but not by processes of different groups. As an example, suppose data is stored on multiple discs in a shared CD-jukebox. When a disc is loaded into the player, users that need data on that disc can access the disc concurrently, whereas users that need data on a different disc have to wait until the current disc is unloaded [43]. Another example includes a meeting room for philosophers interested in different forums or topics [44, 62]. The well-known readers/writers problem is a special case of the group mutual exclusion problem in which all read critical sections belong to the same session but every write critical section belongs to a separate session.

Note that any algorithm that solves the mutual exclusion problem also solves the group mutual exclusion problem. However, the solution is inefficient since critical sections are executed in a serial manner and thus the solution does not permit any concurrency. To rule out such inefficient solutions, a group mutual exclusion algorithm needs to satisfy concurrent entering property. Roughly speaking, the concurrent entering property states that if all processes are requesting the same session, then they *must* be able to execute their critical sections concurrently.

The GME problem has been defined for both message-passing and shared-memory systems. The focus of this work is to develop an efficient GME algorithm for shared-memory systems. Recently, GME-based locks have been used to improve the performance of lock-based concurrent skip lists for multi-core systems using the notion of unrolling by storing multiple key-value pairs in a single node [57]. Unlike in a traditional skip list, most update operations in an unrolled skip list do not need to make any structural changes to the list. This can be leveraged to allow multiple insert operations or multiple delete operations (but not both) to act on the same node simultaneously in most cases. To make structural changes to the list, an operation needs to acquire exclusive locks on the requisite nodes as before. Note that implementing this idea requires GME-based locks; read-write locks do not suffice since a lock needs to support two distinct shared modes. Experimental evaluation showed that, using GME-based locks, we can improve the performance of a concurrent (unrolled) skip list by more than 40% [57].

Next, we consider the red black tree, a self-balancing binary search tree data structure for organizing ordered data and that support search and modify (insert and delete) operations [13]. Red Black Trees provide good worst-case time complexity for search and modify operations. As a result, they are used in symbol table implementations within systems like C++, Java, Python and BSD Unix [59]. They are also used to implement completely fair schedulers in Linux kernel [42]. However, red-black trees have been remarkably resistant to parallelization using both lock-based and lock-free techniques. The tree structure causes the root and high level nodes to become the subject of high contention and thus become a bottleneck. This problem is only exacerbated by the introduction of balancing requirements.

1.1 Our Contributions

In this work, we present two algorithms for the Group Mutual Exclusion problem. The first algorithm is for the cache-coherent shared memory model and the second algorithm is for the distributed shared memory model. Both GME algorithms satisfy the group mutual exclusion, lockout freedom, bounded exit, concurrent entering, and bounded space variable properties. They also satisfy the following desirable properties. First is $O(1)$ concurrent entry step complexity. Note that, as a corollary, a process can enter its critical section within a constant number of its own steps in the absence of any other request, which is typically referred to as contention-free step complexity. To the best of our knowledge, no existing GME algorithm has $O(1)$ concurrent entry step-complexity. Second, both use only $O(m+n^2)$ space for managing m GME objects, where $O(n^2)$ space is shared among all m GME objects. In addition, each process needs only $O(\ell)$ space, where ℓ denotes the maximum number of GME objects (or locks) a process needs to hold at the same time, which is space-optimal. Third, for the cache-coherent model, the number of remote references made by a request under the cache-coherent model, which is referred to as RMR complexity, is $O(\min\{\bar{c},n\})$ in the worst case and $O(\dot{c})$ in the amortized case, where $\bar{c}$ denotes the interval contention of the request (total number of passages involving the same GME object as π that overlap with π) and

$O(\dot{c})$ denotes the point contention of the request (maximum number of passages involving the same GME object as π that are simultaneously in progress in the system at any point). Note that *passage* is defined as the code executed by a process from the time it enters or starts executing the algorithm to the time it exits for a single request. For more information on these, please refer to Section 2.1. For the distributed shared memory model, the RMR complexity is $O(n)$ which is optimal for this model.

We also present an algorithm for a *lock-based* algorithm for a concurrent (strictly-balanced) red-black tree that supports search, insert, and delete operations. Our algorithm uses single-word atomic instruction, namely compare-and-swap (CAS), in the form of a CAS-based lock (TTAS lock). The TTAS lock is shown [11] to perform better than the Java *Reentrant Lock* under low contention scenarios. The insert and delete operations require locks but the search operations are lock-free. In this work, we design a top-down framework similar to the one used in [56]. Our algorithm is similar to the one used in [54], however, it differs from it in fundamental ways with respect to the progress guarantees it provides combined with several optimizations to derive a lock-based algorithm that is able to compete with relaxed versions of binary search trees.

1.2 Dissertation Roadmap

The rest of the text is organized as follows. We first describe the preliminaries and system model necessary to understand our work in Chapter 2. In Chapter 3, we discuss prior related work on group mutual exclusion and background on red black trees. In Chapter 4, we describe the first GME algorithm, provide its pseudo-code, and discuss its proof of correctness. In Chapter 5, we analyze its performance against other GME algorithms and provide a reasoning for the same. We then describe our second GME algorithm and its pseudo-code in Chapter 6. In Chapter 7, we describe our lock-based concurrent red black tree, provide the pseudo-code, and analyze its performance.

CHAPTER 2

SYSTEM MODEL AND PROBLEM SPECIFICATION

We assume an asynchronous shared memory system where a finite set of processes running on a finite set of independent processors communicate by applying read, write and synchronization operations to shared memory locations. Each process also has its own private variables. A system execution is modeled as a sequence of process steps. In each step, a process either performs some local computation affecting only its private variables or executes one of the available instructions (read, write or RMW) on a shared variable. Processes take steps asynchronously. This means that in any execution, between two successive steps of a process, there can be an unbounded but finite number of steps performed by other processes.

2.1 Shared Memory Model Types

Typically, systems are organized with a memory hierarchy which involves a large, slow main memory and one or more layers of smaller, faster cache memories. These cache memories may be shared by one or more processors or they may be unique to each processor. Multi-core processors generally contain two to three levels of caches (L1-L3). The last level cache (L3) is typically situated farthest from the processor and is shared among multiple cores. The L2 cache may or may not be shared between multiple cores. The L1 cache is dedicated to a single core.

Most GME or Mutual exclusion lock algorithms include busy-waiting loops wherein a process spins on some flag waiting for some condition. This spinning may generate a lot of traffic on the interconnection between the processor and memory which may slow down other processes. Thus, it is important to distinguish between *local* access and *remote* access and to try to reduce the number of remote accesses as much as possible. We consider two machine architecture models: (a) Cache-Coherent (CC) model and (b) Distributed Shared Memory (DSM) model. The two models differ on where shared variables are physically stored and what is the overhead of accessing them. We define

a remote memory reference (RMR) by process p as an attempt to reference (access) a memory location that does not physically reside on ps local memory.

Cache-Coherent Model In the CC model, all shared variables are stored in a central location or global store. Each processor has a private cache. When a process accesses a shared variable, a copy of the contents of the variable is saved in the private cache of the process. Thereafter, every time the process reads that shared variable, it does so using its cached (local) copy until the cached copy is invalidated. Also, every time a process writes to a shared variable, it writes to the global store, which also invalidates all cached copies of the variable.

Distributed Shared Memory Model In the DSM model, instead of having the shared memory in a central location or a global store, each process "owns" a part of the shared memory and keeps it in its own local memory. Every shared variable is stored in the local memory of some process. Accessing a shared variable stored in the local memory of a different process causes the process to make a remote memory reference. A reference to a variable stored in a non-local memory requires traversing the processor-to-memory interconnect, which takes much longer to access than to access a locally stored variable.

In the CC model, spinning on a memory location generates at most two RMRs—one when the variable is cached and the other when the cached copy is invalidated. In the DSM model, spinning on a variable that is stored in remote memory may generate an unbounded number of RMRs. An algorithm is called *local-spinning* (under CC or DSM model) if the maximum number of RMRs made in entry and exit sections is bounded. It is desirable to design algorithms that minimize the number of remote memory references because this factor can critically affect the performance of these algorithms [50]. In general, it is difficult to achieve bounded number of RMRs in the DSM Model. In this work, we analyze the RMR complexity of our algorithm under the CC model.

We express the RMR complexity of our GME algorithm using the following measures of contention. The *interval contention* of a passage π, denoted by $\bar{c}(\pi)$, is defined as the total number

of passages involving the same GME object as π that overlap with π. The *point contention* of a passage π, denoted by $\dot{c}(\pi)$, is defined as the maximum number of passages involving the same GME object as π that are simultaneously in progress in the system at any point during π.

Note that *passage* is defined as the code executed by a process from the time it enters or starts executing the algorithm to the time it exits for a single request.

Figure 2.1 shows an execution involving 3 processes and 5 operations. The interval contention and point contention are described in the figure.

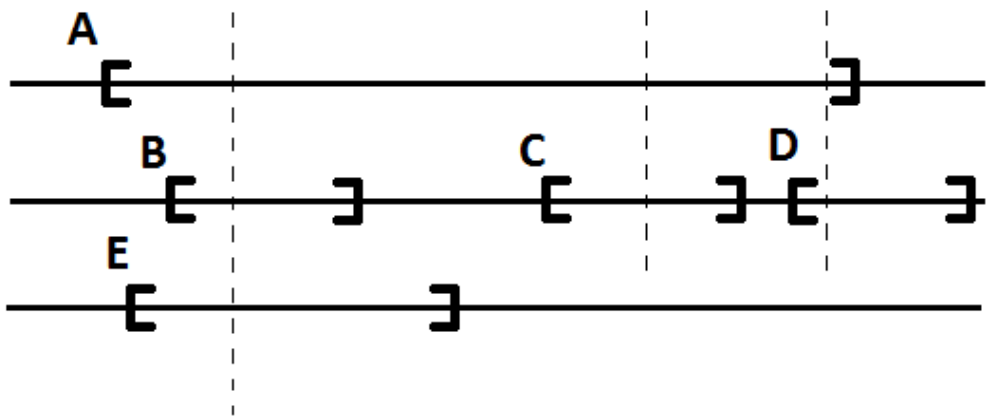


Figure 2.1: Sample execution with 3 processes and 5 operations. The point contention of A, B, and E are 3, while the point contention of C and D are 2. The interval contention of A, B, C, D, and E are 5, 3, 2, 2, and 3 respectively.

2.2 Synchronization Primitives

Processes can only communicate by performing read, write and read-modify-write (RMW) instructions on shared variables. In the first part of our work, we assume the availability of two RMW instructions, namely *compare-and-swap (CAS)* and *fetch-and-add (FAA)* for our algorithms. A compare-and-swap instruction takes a shared variable x and two values u and v as inputs. If the current value of x matches u, then it writes v to x and returns true. Otherwise, it returns false. A

Algorithm 1: Structure of a GME Algorithm

```
while true do
    NON-CRITICAL SECTION (NCS)
    ENTRY SECTION                    // try to enter critical section
    CRITICAL SECTION (CS)            // execute critical section
    EXIT SECTION                     // exit critical section
```

fetch-and-add instruction takes a shared variable x and a value v as inputs, returns the current value of x as output, and, at the same time, increments the value of x by v. These instructions are commonly available in many modern processors such as Intel 64 [2] and AMD64 [1]. In the second part of our work, we make use of exclusive locks based on *compare-and-swap (CAS)* instruction.

2.3 GME Problem Specification

In the GME problem, each process repeatedly executes four sections of code, namely *non-critical section (NCS)*, *entry section*, *critical section (CS)*, and *exit section*, as shown in Algorithm 1. Each critical section is associated with a *type* or a *session*. Critical sections belonging to the *same session* can execute *concurrently*, whereas critical sections belonging to *different sessions* must be executed *serially*. We refer to the code executed by a process from the beginning of its entry section until the end of its exit section as an *passage*. Note that the session associated with a critical section may be different in different passages (and is selected based on the needs of the underlying application). We say that a process is *active* if it is in one of its *passages*.

We assume that every process is *live* meaning that, if it is not executing its non-critical section, then it will eventually execute its next step.

2.3.1 Properties

Solving the GME problem involves designing code for entry and exit sections in order to ensure the following four properties are satisfied in each passage:

(P1) Group mutual exclusion If two processes are in their critical sections at the same time, then they have requested the same session.

(P2) Lockout freedom If a process is trying to enter its critical section, then it is able to do so eventually (entry section is finite).

(P3) Bounded exit If a process is trying to leave its critical section, then it is able to do so eventually within a bounded number of its own steps (exit section is bounded).

(P4) Concurrent entering If a process is trying to enter its critical section and no process in the system is requesting a different session, then the (former) process is able to enter its critical session eventually within a bounded number of its own steps (entry section is bounded in the absence of a request for a different session).

2.3.2 Complexity Measures

We say that two requests *conflict* if they involve the same GME object but belong to different sessions. We say that a request is *outstanding* until its process has finished executing the exit section. We use the following metrics to evaluate the performance of our GME algorithm:

Context switch complexity It is defined as the maximum number of sessions that can be established while a process is waiting to enter its critical section. It is also referred to as session switch complexity elsewhere [45, 28].

Concurrent entry step complexity It is defined as the maximum number of steps a process has to execute in its entry and exit sections provided no other process in the system has an outstanding conflicting request during that period.

Remote memory reference (RMR) complexity It is defined as the maximum number of remote memory references required by a process in its entry and exit sections.

In addition, we also consider the memory footprint of the GME algorithm when the system contains multiple GME objects.

Multi-object space complexity It is defined as the maximum amount of space needed to instantiate and maintain a certain number of GME objects.

2.4 Red Black Tree Preliminaries

We assume that a binary search tree (BST) implements a dictionary abstract data type and supports *search*, *insert* and *delete* operations [19]. For convenience, we refer to the insert and delete operations as *modify* operations. A search operation explores the tree for a given key and returns true if the key is present in the tree and false otherwise. An insert operation adds a given key to the tree if the key is not already present in the tree. Duplicate keys are not allowed in our model. A delete operation removes a key from the tree if the key is indeed present in the tree. In both cases, a modify operation returns true if it changed the set of keys present in the tree (added or removed a key) and false otherwise.

Given a tree, we refer to a sequence of nodes in the tree such that every node in the sequence (except for the last node) is the *parent* of the next node in the sequence as a *path*. Further, we refer to the top-most and bottom-most nodes in the path as its *head* and *tail* nodes, respectively.

A binary search tree satisfies the following key-based properties:

(1) the left subtree of a node contains only nodes with keys less than or equal to the node's key

(2) the right subtree of a node contains only nodes with keys greater than the node's key

(3) the left and right subtrees of a node are also binary search trees.

To locate a key in a binary search tree, the tree is traversed starting from its root node. At each node, the target key (the key being searched) is compared with the stored key (the key present at the node) and depending on the result either the node's left child (the target key is smaller than the

stored key) or its right child is followed (otherwise). This path in the tree from the root node to a leaf node as induced by a key is referred to as its *access-path*.

A red-black tree (RBT) is a type of self-balancing binary search tree in which every node is either colored *red* or *black*. A red-black tree satisfies the following color-based properties:

(1) each node in the tree is either red or black

(2) a red node cannot have a red child node

(3) every path from a given node to any of its descendant leaf node contains the same number of black nodes.

The above color-based properties ensure that the height of a red-black tree is a logarithmic function of the number of keys present in the tree. We use an *external* red-black tree in our algorithm. Furthermore, every internal node has exactly two children.

2.5 Correctness Conditions

To demonstrate the correctness of our algorithms, we use linearizability [37] as the safety property. Each operation consists of an invocation and response event. Essentially, linearizability requires that an operation should appear to take effect instantaneously at some point between these events. This point is called the linearization point. For our lock-based red black tree algorithm we use deadlock-freedom [35] for the liveness property. Deadlock-freedom requires that some process with a pending operation be able to complete its operation eventually. For our GME algorithms, we use the starvation-freedom for the liveness property. Starvation-freedom requires that every operation (session request) executed by every process will eventually complete.

CHAPTER 3

BACKGROUND AND RELATED WORK

In this chapter, we first give an overview of earlier work done in the area of Group Mutual Exclusion. Next, we consider the related work for the Concurrent Red Black Tree data structure.

3.1 Related Work for Group Mutual Exclusion

Since the GME problem was first introduced by Joung around two decades ago [43], several algorithms have been proposed to solve the problem for shared-memory systems [43, 45, 28, 62, 41, 17, 7, 30]. These algorithms provide different trade-offs between fairness, concurrency, step complexity and space complexity.

3.1.1 Previous Work Drawbacks

To the best of our knowledge, all of the prior work suffers from at least one and possibly both of the following drawbacks:

Drawback 1 (high step complexity in the absence of any conflicting request): In a system using fine-gained locking, most of the lock acquisitions are likely to be uncontended in practice (i.e., at most one process is trying to acquire a given lock). Note that this is the primary motivation behind providing a fast-path mechanism for acquiring a lock [35]. Moreover, in concurrent unrolled skip lists using GME-based locks [57], most of the lock acquisitions involve only two shared sessions. In many cases, all requests are likely to be for the same session. This necessiates the need for a GME algorithm that has low step-complexity when all requests for acquiring a given lock are for the same session, which we refer to as *concurrent entry step complexity*. (Note that this includes the case where there is only one request for lock acquisition.)

To the best of our knowledge, except for two, all other existing GME algorithms have concurrent entry step complexity of $\Omega(n)$, where n denotes the number of processes in the system. The

GME algorithm by Bhatt and Huang [7] has concurrent entry step-complexity of $O(\min\{\log n, \dot{c}\})$, where $\dot{c}$ denotes the point contention of the request. Also, one of the GME algorithms by Danik and Hadzilcos [17, Algorithm 3] has concurrent entry step complexity of $O(\log s \cdot \min\{\log n, \dot{c}\})$, where s denotes the number of different types of sessions.

Drawback 2 (high space complexity with a large number of GME objects): All the existing work in this area has (implicitly) focused on a *single* GME object. However, many systems use fine-grained locking to achieve increased scalability in multi-core/multi-processor systems. For example, each node in a concurrent data structure is protected by a separate lock [35, 57].

All the existing GME algorithms that guarantee starvation freedom have a space-complexity of at least $\Theta(n)$ for a single GME object. Note that this is expected because mutual exclusion is a special case of group mutual exclusion and any starvation-free mutual exclusion algorithm requires $\Omega(n)$ space even when powerful atomic instructions such as compare-and-swap are used [22]. Some of these GME algorithms (*e.g.*, [45, 17, 30]) can be modified relatively easily to share the bulk of this space among all GME objects and, as a result, the additional space usage for each new GME object is only $O(1)$. However, it is not clear how the other GME algorithms (*e.g.*, [43, 28, 62, 41, 17, 7, 30]) can be modified to achieve the same space savings. For these GME algorithms, to our understanding, the additional space usage for each new GME object is at least $\Theta(n)$. We refer to the former set of GME algorithms as *space-efficient* and the latter set of GME algorithms as *space-inefficient*.

Consider the example of a concurrent data structure using GME-based locks to improve performance [57]. If n is relatively large, then the size of a node equipped with a lock based on a GME algorithm that is space-inefficient may be several factors more than its size otherwise. This will *significantly* increase the memory footprint of the concurrent data structure, which, in turn, will adversely affect its performance and may even negate the benefit of increased concurrency resulting from using a GME-based lock.

Table 3.1: Fairness and concurrency properties satisfied by different algorithms. Note that all algorithms satisfy P1.

Algorithm	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**
Joung [43]	✓	✓	✓	✓	✗	✗	✗	✗
Keane & Moir [45]	✓	✗	✗	✗	✗	✗	✗	✗
Hadzilacos [28]	✓	✓	✓	✗	✓	✓	✗	✗
Takamura & Igarashi [62, Algorithm 1]	✗	✓	✗	✗	✗	✗	✗	✗
Takamura & Igarashi [62, Algorithm 2]	✓	✗	✗	✗	✗	✗	✗	✗
Takamura & Igarashi [62, Algorithm 3]	✓	✗	✗	✗	✗	✗	✗	✗
Jayanti *et al.* [41, Algorithm 1]	✓	✓	✓	✗	✓	✓	✗	✗
Jayanti *et al.* [41, Algorithm 2]	✓	✓	✓	✓	✓	✗	✓	✗
Jayanti *et al.* [41, Algorithm 3]	✓	✓	✓	✓	✓	✗	✓	✗
Danek & Hadzilacos [17, Algorithm 1]	✓	✓	✓	✓	✓	✓	✓	✗
Danek & Hadzilacos [17, Algorithm 2]	✓	✓	✓	✗	✗	✓	✗	✗
Danek & Hadzilacos [17, Algorithm 3]	✓	✓	✓	✗	✓	✓	✗	✗
Bhatt & Huang [7]	✓	✓	✓	✗	✗	✓	✗	✓
He *et al.* [30, Algorithm 1]	✓	✓	✓	✗	✓	✗	✗	✗
He *et al.* [30, Algorithm 2]	✓	✓	✓	✗	✓	✗	✗	✗
Our Algorithm [This Work]	✓	✓	✓	✗	✗	✗	✗	✗

Table 3.2: Complexity measures for ME algorithms used by some GME algorithms.

Algorithm	**Space Complexity**	**Space Shareable Across Multiple Objects**	**Solitary Request Step Complexity**	**RMR Complexity**	**RMW Instructions**
Yang & Anderson's Algorithm 1 [65]	$O(n)$	✗	$O(\log n)$	$O(\log n)$	-
Mellor-Crummey & Scott's Algorithm [50]	$O(1)$	✓	$O(1)$	$O(n)$	FAS

n: number of processes

Table 3.3: Complexity measures of GME algorithms excluding those in [41, 17] that use an abortable ME algorithm as a subroutine.

Algorithm	Multi-Object Space Complexity	Solitary Request Step Complexity	Concurrent Entering Step Complexity	RMR Complexity	Bounded Shared Variables	RMW Instructions
Joung [43]	$O(mn)$	$\Omega(n)$	$\Omega(n)$	∞	✓	-
Keane & Moir [45] (with Yang & Anderson's Algorithm 1)	$O(mn)$	$O(\log n)$	–	$O(\log n + \dot{c})$	✓	-
Keane & Moir [45] (with Mellor-Crummey and Scott's Algorithm)	$O(m+n)$	$O(1)$	–	$O(n)$	✓	FAS
Hadzilacos [28]	$O(mn^2)$	$\Omega(n)$	$\Omega(n)$	$O(n+\dot{c}^2)$	✓	-
Takamura & Igarashi [62, Algorithm 1]	$O(m+n)$	$\Omega(n)$	–	∞	✓	-
Takamura & Igarashi [62, Algorithm 2]	$O(m+n)$	$\Omega(n)$	–	$O(n)$	✗	-
Takamura & Igarashi [62, Algorithm 3]	$O(m+n)$	$\Omega(n)$	–	$O(n)$	✗	-
Jayanti *et al.* [41, Algorithm 1]	$O(mn)$	$\Omega(n)$	$\Omega(n)$	$O(n+\dot{c}^2)$	✓	-
He *et al.* [30, Algorithm 1]	$O(m+n)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	-
He *et al.* [30, Algorithm 2]	$O(m+n)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✓	-
Bhatt & Huang [7]	$O(mn)$	$O(1)$	$O(\min\{\log n, \dot{c}\})$	$O(\min\{\log n, \dot{c}\})$	✗	LL/SC
Our Algorithm [This Work]	$O(m+n^2)$	$O(1)$	$O(1)$	$O(\dot{c})^*$	✓	CAS and FAA

–: the algorithm does not satisfy P4

n: number of processes

m: number of GME objects

s: number of different types of sessions

$\dot{c}$: point contention of the request

$*$: amortized case

Table 3.4: Complexity measures of the GME algorithms in [41, 17] using the three abortable mutex algorithms.

Abortable ME Algorithm	Algorithm	Space Complexity	Solitary Request Step Complexity	Concurrent Entering Step Complexity	RMR Complexity	Bounded Shared Variables	RMW Instructions
n-bit FCFS	Jayanti *et al.* [41, Algorithm 2]	$O(mn^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✓	-
	Jayanti *et al.* [41, Algorithm 3]	$O(mn^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	-
	Danek & Hadzilacos [17, Algorithm 1]	$O(mn^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✓	-
	Danek & Hadzilacos [17, Algorithm 2]	$O(mn^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✓	CAS and FAA
	Danek & Hadzilacos [17, Algorithm 3]	$O(mn^2s)$	$O(n\log s)$	$O(n\log s)$	$O(n\log s)$	✗	CAS and FAA
modified Bakery algorithm	Jayanti *et al.* [41, Algorithm 2]	$O(mn^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	-
	Jayanti *et al.* [41, Algorithm 3]	$O(mn)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	-
	Danek & Hadzilacos [17, Algorithm 1]	$O(m+n^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	-
	Danek & Hadzilacos [17, Algorithm 2]	$O(m+n^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	CAS and FAA
	Danek & Hadzilacos [17, Algorithm 3]	$O(mns)$	$O(n\log s)$	$O(n\log s)$	$O(n\log s)$	✗	CAS and FAA
Jayanti's algorithm	Jayanti *et al.* [41, Algorithm 2]	$O(mn^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	-
	Jayanti *et al.* [41, Algorithm 3]	$O(mn)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	-
	Danek & Hadzilacos [17, Algorithm 1]	$O(mn+n^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	-
	Danek & Hadzilacos [17, Algorithm 2]	$O(mn+n^2)$	$\Omega(n)$	$\Omega(n)$	$O(n)$	✗	CAS and FAA
	Danek & Hadzilacos [17, Algorithm 3]	$O(mns)$	$O(\log s)$	$O\left(\begin{matrix}\log s \times \\ \min\{\log n, \dot{c}\}\end{matrix}\right)$	$O\left(\begin{matrix}\log s \times \\ \min\{\log n, \dot{c}\}\end{matrix}\right)$	✗	CAS and FAA

n: number of processes
m: number of GME objects
s: number of different types of sessions
$\dot{c}$: point contention of the request

3.1.2 Subroutines

Most of the earlier algorithms use only read and write instructions whereas many of the later algorithms use atomic instructions as well. As mentioned earlier, different algorithms provide different fairness, concurrency and performance guarantees.

Many GME algorithms use a *traditional* or an *abortable* mutual exclusion (ME) algorithm as a subroutine. The GME algorithm proposed by Keane and Moir in [45] uses a traditional ME algorithm as an exclusive lock to protect access to entry and exit sections of the algorithm. As such, this algorithm does not satisfy bounded exit and concurrent entering properties. The GME algorithms presented in [17, 7] use an abortable ME algorithm as a subroutine. The main idea is that a process can enter its critical section using multiple pathways: (i) as a "leader" by establishing a new session, or (ii) as a "follower" by joining an existing session. The first case occurs if the process is able to acquire the exclusive lock. The second case occurs if the process learns that a session "compatible" with its own request is already in progress in which case it aborts the ME algorithm and joins that session. Both pathways are explored concurrently and, as soon as one of them allows the process enter its critical section, the other one is abandoned.

3.1.3 Fairness and Concurrency Guarantees

In many (group) mutual exclusion algorithms, the entry section consists of two distinct subsections: a *doorway* and a *waiting-room*. A doorway is the wait-free portion of the entry section that a process can complete within a bounded number of its own steps. A waiting-room of the entry section is the portion where a process is blocked until it is its turn to execute its critical section.

We say that two active processes are *fellow* processes if they are requesting the same session (of the same GME object) and *conflicting* processes if they are requesting different sessions (of the same GME object).

We say that an active process p *doorway-preceeds* another active process q if p completes the doorway before q enters the doorway. Besides the four properties listed in Section 2.3.1, a GME

algorithm may satisfy one or more of the properties listed below. These properties, which were defined in [28, 41, 7], describe additional guarantees that a GME algorithm may provide.

(P5) Strong Concurrent Entering If a process p has completed its doorway, and p doorway-precedes every active conflicting process, then p enters its critical section within a bounded number of its own steps.

(P6) First-Come-First-Served (FCFS) If p and q are two conflicting processes such that p doorway-preceeds q, then p enters its critical section before q.

(P7) Relaxed FCFS If p and q are two conflicting processes such that p doorway-preceeds q but q enter its critical section before p, then there exists another process r whose current attempt overlaps with that of q such that q and r are fellow processes p does not doorway-preceed r.

(P8) First-In-First-Enabled (FIFE) If p and q are two fellow processes such that p doorway-preceeds q and q enters its critical section before p, then p can enter its critical section within a bounded number of its own steps.

(P9) Pulling Suppose p and q are two fellow processes such that p is currently in its critical section and doorway-preceeds all conflicting processes. If q is currently in the waiting room, then q can enter its critical section within a bounded number of its own steps.

3.2 Related Work for Concurrent Red Black Tree

Several lock-based algorithms for concurrent search trees have been proposed in [6, 9, 16].

Universal constructions that can be used to derive concurrent lock-free and wait-free data structures from a sequential version have been proposed in [31, 32, 12, 21]. Due to the general nature of the constructions, when applied to a binary search tree, the resultant data structures are quite inefficient. They involve either: (a) applying operations to the data structure in a serial manner, or (b) copying the entire data structure (or parts of it that will change and any parts that directly

or indirectly point to them), applying the operation to the copy and then updating the relevant part of the shared data structure to point to the copy. The first approach precludes any concurrency. The second approach, when applied to a tree, also precludes any concurrency since the root node of the tree indirectly points to every node in the tree. A general wait-free construction for a tree-based data structure in which all operations work in a top-down manner has been proposed in [64]. Operations are injected into the tree at the root node, and work their way down, toward a leaf node, by operating on a small contiguous portion or window of the tree. When compared with universal constructions, the wait-free algorithm obtained using Tsay and Li's tree-based framework, hereafter referred to as *TL-framework*, yields more efficient modify operations (as modify operations working on different parts of the tree can execute concurrently once their paths diverge) but less efficient search operations (as search operations also have to copy nodes and execute atomic instructions).

Natarajan et al. presented an efficient wait-free algorithm for a concurrent red-black tree in [56] based on *modified* TL-framework. The TL-framework was modified to remove its practical limitations such as: (i) using an atomic instruction that is typically not implemented in hardware, and (ii) storing two addresses in a single word. Furthermore, operations in Natarajan *et al.*'s algorithm (search as well as modify) are much more efficient than those obtained directly from the framework (even the modified framework). Concurrent algorithms for red-black trees using transactional memory support have been proposed in [25, 15].

A lock-based, relaxed-balanced AVL tree was proposed in [9]. This is a variation of internal trees, referred to as *partially-external trees*. An internal tree maintains values in inner nodes whereas an external tree maintains values only at the leaves and the key of an inner node is used for routing purposes only. In *partially-external trees*, a node with two children is marked as logically removed via a designated flag, and it is not physically removed until its number of children reduces to one due to another removal or due to rotations. An insert can revive such a node by flipping this flag to false. Internal trees may suffer more contention in smaller trees since threads contend

on nodes more frequently (since there could be up to two times fewer nodes in internal trees than in external trees). However, delete operations are simpler in external trees than internal trees. Finally, a non-blocking algorithm for a relaxed variant of red-black trees (namely, chromatic tree) has been proposed in [10]. In this paper, Brown *et al.*, proposed general primitives for developing non-blocking algorithms for concurrent search trees, namely LLX, SCX, and VLX. Using these primitives, they have developed efficient non-blocking algorithms for relaxed external red-black trees.

In relaxed balanced trees, the mutating operations are decoupled from the rebalancing operations i.e. the mutating operation is first completed and only sometime afterwards, the rebalance process is started. Although relaxed balanced trees are guaranteed to be strictly balanced when there are no ongoing mutating operations [9, 10], the rebalance process may be arbitrarily delayed. In strictly balanced trees, the balancing takes place after every mutating operation. Balanced trees provide logarithmic worst-case time complexity guarantees. This becomes crucial as the tree grows in size or when the values for the operations are picked non-uniformly. Thus, it may be beneficial to focus on strictly balanced trees if their performance can be shown to be on-par with relaxed variants.

CHAPTER 4

GME ALGORITHM FOR CACHE-COHERENT (CC) MODEL

In this chapter, we present our GME algorithm for the Cache-coherent model. We first discuss Herlihy's algorithm which inspired our work. We then describe our GME algorithm in an incremental manner. First, we describe a basic GME algorithm that is only deadlock-free (some session request is eventually satisfied but a given request may be starved), and uses unbounded space. Next, we enhance the basic algorithm to achieve starvation freedom (every session request is eventually satisfied) using a helping mechanism. Finally, we enhance the algorithm to make it space-efficient by reusing nodes using a memory reclamation algorithm. Note that all our algorithms are safe in the sense that they satisfy the group mutual exclusion property. For ease of exposition, we describe the first two variants followed by Section 4.2 along with a correctness proof and complexity analysis. We then describe the third (and the final) variant in the section thereafter, Section 4.3.

4.1 The Main Idea

Our GME algorithm is inspired by Herlihy's universal construction for deriving a wait-free linearizable implementation of a concurrent object from its sequential specification using consensus objects [31, 32]. Roughly speaking, the universal construction works as follows. The state of the concurrent object is represented using (i) its initial state and (ii) the sequence of operations that have applied to the object so far. The two attributes of the object are maintained using a singly linked list in which the first node represents the initial state and the remaining nodes represent the operations. To perform an operation, a process first creates a new node and initializes it with all the relevant details of the operation, namely its type and its input arguments. It then tries to append the node at the end of the list. To manage conflicts in case multiple processes are trying to append their own node to the list, a consensus object is used to determine which of several nodes is chosen to be appended to the list. Specifically, every node stores a consensus object and the consensus object

of the current last node is used to decide its successor (*i.e.*, the next operation to be applied to the object). A process whose node is not selected simply tries again. A helping mechanism is used to guarantee that every process trying to perform an operation eventually succeeds in appending its node to the list.

We modify the aforementioned universal construction to derive a GME algorithm that satisfies several desirable properties. Intuitively, an operation in the universal construction corresponds to a critical section request in our GME algorithm. *Appending a new node* to the list thus corresponds to *establishing a new session*. However, unlike in the universal construction, a single session in our GME algorithm can be used to satisfy multiple critical section requests. This basically means that every critical section request does not cause a new node to be appended to the list. This requires some careful bookkeeping so that no "useless" sessions are established. Further, a simple consensus algorithm, implemented using CAS instruction, is used to determine the next session to be established.

For ease of exposition, we describe the first two variants in the next section, Section 4.2 along with a correctness proof and complexity analysis. We then describe the third (and the final) variant in the section thereafter, Section 4.3.

4.2 A Starvation-Free Algorithm

In this section, assume that nodes are *never* reused. A pseudocode of our GME algorithm is given in Algorithms 2-7. In the pseudocode, *me* refers to the identifier of the process (*e.g.*, *me* for process p_i will evaluate to i).

4.2.1 Data Structures Used

List node: Central to our GME algorithm is a (list) node (Figure 4.1); it is used to maintain information about a session. As opposed to the linked list in the wait-free construction, which is a singly linked list, we maintain a doubly linked list. A node stores the following information

Algorithm 2: Data types and variables used.

```
   // Node of a list
6  struct Node {
       integer session;                          // session associated with the node
       integer instance;                   // instance identifier of the GME object
       integer number;                             // the next process to be helped
       {bool,bool,bool,bool} state;                // four flags representing state
       integer size;               // number of processes currently in the session
       NodePtr prev, next;               // address of the previous and next nodes
       integer owner;                            // the last process to own the node
   };

7  shared variables
8      head: array [1...m] of NodePtr;     // to store references to head nodes of
        lists
9      announce: array [1...n] of NodePtr, initially [null,...,null];     // to announce CS
        requests

10 private variables
11     snapshot: array [1...n] of NodePtr; // to store snapshots of the head nodes
       // snapshot[i] is a private variable of process p_i

   initialization
12 begin
       // initialize shared variables
13     foreach i ∈ [1...m] do
14         head[i] := new Node;                                  // create a new node
15         head[i] → state := LEADERLESS ;                   // session has no leader
16         head[i] → size := 0;                           // session has no processes
17         head[i] → next := null;                           // node has no successor
           // all other fields can be initialized arbitrarily
18     foreach i ∈ [1...n] do
19         announce[i] := null ;                // process has no outstanding request
```

(line 6): (a) the session represented by the node, (b) the instance identifier of the GME object to which the session belongs, (c) the state of the session, (d) the size of the session, (e) the address of the previous and next nodes in the (doubly linked) list, and (f) the owner of the node.

Algorithm 3: Functions operating on session state.

```
// returns true if the session is closed and false otherwise
bool IsClosed(integer state)    { return (state & LEADERLESS) and (state & CONFLICT); }

// returns true if the session is adjourned and false otherwise
bool IsAdjourned(integer state)    { return (state & VACANT); }

// returns true if the node is retired and false otherwise
bool IsRetired(integer state)    { return (state & RETIRED); }

// sets a given guard flag (LEADERLESS or CONFLICT) in the session
   state
SetGuardFlag(NodePtr node, bool flag )
begin
    while true do
        integer state := node → state ;                    // read the current state
        if (state & flag) then return;                     // flag already set
        if CAS(node → state, state, state | flag) then return; // successfully set the
          flag

// sets the vacant flag in the state if possible
SetVacantFlag(NodePtr node)
begin
    integer state := node → state;                         // read the current state
    if not(IsClosed(state)) then return;                   // session is still open
    if (node → size ≠ 0) then return;                      // session still has participants
    CAS(node → state, state, state | VACANT);

// mark the node as retired
MarkAsRetired(NodePtr node)    { node → state := LEADERLESS | CONFLICT | VACANT
  | RETIRED; }
```

A session (or node) has four possible states: (i) *open:* it means that the session is currently in progress and new processes can join in, (ii) *closed:* it means the session is currently in progress but no new processes can join in, (iii) *adjourned:* it means that the session is no longer in progress and has no participating processes, and (iv) *retired:* it means that the node is no longer needed to either establish or maintain an already established session. When a session is first established, it is

Algorithm 4: Functions operating on list head.

```
// reads the current head pointer of the list
READHEAD(integer instance)
begin
    snapshot[me] := head[instance];

// returns true if the head of the list has not moved and false
   otherwise
bool TESTHEAD(integer instance)
begin
    if (head[instance] ≠ snapshot[me]) then return false;        // head has advanced
    else return true;

// advances the head of a list to the given node if the head has not
   moved
ADVANCEHEAD(integer instance, NodePtr successor)
begin
    CAS(head[instance], snapshot[me], successor);
```

in open state. It stays open as long as one of the following conditions still holds: (1) there is no conflicting request in the system, or (2) the request that established the session is still outstanding, *i.e.*, executing its critical section. Once both the conditions become false, the session moves to closed state. Note that, in closed state, the session may still have participants executing their critical sections. Once all such participants have left the session, the session moves to adjourned state. Finally, the node associated with a request is retired once either the session established by the node has adjourned and a new session has been established or the node is no longer needed to establish a session.

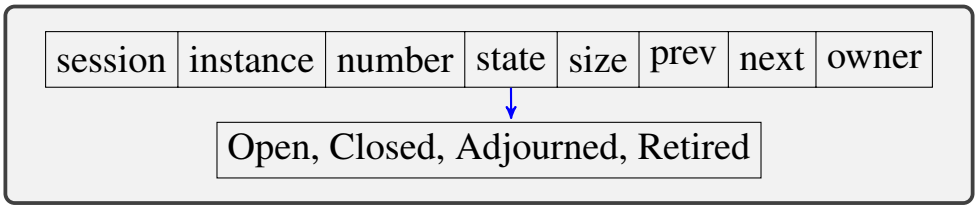


Figure 4.1: Structure of the node with its contents

Algorithm 5: Entry section.

```
// code for entry section
ENTER(integer myinstance, integer mysession)
begin
    // initialize a node and announce the request to other processes
    GETNEWNODE(myinstance, mysession);
    NodePtr mynode := announce[me];
    while true do
        READHEAD(myinstance);                    // read the head pointer of the list
        NodePtr current := snapshot[me];         // find the last node in the list
        if (announce[me] = current) then
            // join the session as a leader and retire the predecessor
               node
            RETIRENODE(mynode → prev);
            return;
        if (current → session = mysession) then   // my request is compatible with
         the current session
            if not(ISCLOSED(current → state)) then          // the session is open
                // attempt to join the session as a follower
                FAA(current → size, 1);                 // increment the session size
                if not(ISCLOSED(current → state)) then      // the session is still
                 open
                    // join the session as a follower and retire own node
                    RETIRENODE(mynode);
                    return;
                else                                  // the session is no longer open
                    FAA(current → size, -1);     // abort the attempt and decrement
                     the session size
                    SETVACANTFLAG(current);   // set VACANT flag if applicable
        else                  // my request conflicts with the current session
            SETGUARDFLAG(current, CONFLICT);              // set CONFLICT flag
            SETVACANTFLAG(current);          // set VACANT flag if applicable
        while not(ISADJOURNED(current → state)) do ;                // do nothing
        ;                                                                  // spin
        if TESTHEAD(myinstance) then APPENDNEXTNODE(myinstance); // establish
         a new session
```

Algorithm 6: Exit section.

```
// code for exit section
EXIT(integer myinstance)
begin
    READHEAD(myinstance);                    // find the head node of the list
    NodePtr current := snapshot[me];
    if (current → owner = me) then           // joined the session as a leader
        SETGUARDFLAG(current, LEADERLESS);       // set the LEADERLESS flag
    FAA(current → size, -1);                     // decrement the session size
    SETVACANTFLAG(current);                  // set VACANT flag if applicable
```

We use four flags to represent session state: (1) `LEADERLESS` flag to indicate that the session leader has left its critical section, (2) `CONFLICT` flag to indicate that some process has made a conflicting request, (3) `VACANT` flag to indicate that the session is empty or vacant, and (4) `RETIRED` flag to indicate that the node has been retired. For convenience, we refer to the first two flags as *guard* flags, the third flag as *vacant* flag and the fourth flag as *retired* flag.

The vacant flag is set only after *both* the guard flags have been set. Thus a session is closed if both its guard flags are set. It is adjourned if its vacant flag is also set. Finally, a node is considered retired if its retired flag is set. For convenience, when the retired flag is set, we set the remaining three flags as well to simplify the algorithm. Thus if the vacant flag is set, then both the guard flags are also set; if the retired flag is set, then the vacant flag as well as both the guard flags are also set. All the four flags are stored in a single word hence the value of session state can be easily read and updated atomically.

The size of a session refers to the number of processes that have joined or trying to join the session, *i.e.*, still executing their critical sections.

Shared variables: Each GME object has a separate linked list associated with it. Each list has a *head*, which points to the last node in the list. Initially, the head of each list points to a "dummy"

Algorithm 7: Functions operating on a list node.

```
// get a new node, initialize it and announce it to other processes
GETNEWNODE(integer instance, integer session)
begin
    NodePtr node := get a new node;                    // invoke dynamic memory manager
    node → owner := me;                                // set the owner as myself
    node → instance := instance;                       // initialize node's instance
    node → session := session;                         // initialize node's session
    node → size := 1;                                  // initialize session size
    node → next := null ;                              // node has no successor
    node → prev := null ;                              // node has no predecessor
    node → state := 0;                 // session is open with no condition flag set
    node → number := 0;              // set the sequence number to a sentinel value
    announce[me] := node;              // make the node visible to other processes

// get the next node to be appended to the list
NodePtr SELECTNEXTNODE(integer instance)
begin
    NodePtr mine := announce[me];                                        // my node
    NodePtr helpee := announce[snapshot[me] → number];                 // helpee's node
    // ascertain that the helpee's node is usable
    if (helpee = null) then return mine;      // process has no outstanding request
    if (helpee → instance ≠ instance) then  return mine;        // request is for a
     different GME object
    if ISRETIRED(helpee) then  return mine;                // node has been retired
    if not(TESTHEAD(instance)) then  return mine;                // head has moved
    return helpee;                          // helpee's node passed all the tests
```

node representing an adjourned session. For ease of exposition, we assume that pointers to all head nodes are stored in an array with one entry for each GME object, denoted by *head* (line 8).

To enable helping, each process *announces* its request by storing address of the node associated with its request in an array with one entry for each process, denoted by *announce* (line 9).

Private variables: In addition, each process uses a private variable to maintain a snapshot of the pointer to the head node of the list associated with the current request (line 11). Note that a private variable is modeled as an array in our algorithm; the ith entry of each array is private to process p_i.

Algorithm 8: More Functions operating on a list node.

```
// append a new node to the list
APPENDNEXTNODE(integer instance)
begin
    NodePtr current := snapshot[me];                    // get the last node in the list
    NodePtr successor := SELECTNEXTNODE(instance);  // choose a node to append
    CAS(current → next, null, successor);    // set the next field of the current
     last node
    NodePtr successor := current → next ;                    // read the next field
    if not(TESTHEAD(instance)) then  return;                    // head has moved
    successor → prev := current;     // set the previous field of the successor
    successor → number := (current → number + 1)  mod n + 1;   // set the sequence
     number used in helping
    ADVANCEHEAD(instance, successor);                        // advance the head

// retire the node
RETIRENODE(NodePtr node)
begin
    announce[me] := null;                            // help is no longer needed
    MARKASRETIRED(node);                             // mark the node as retired
```

Managing session state: Algorithm 3 shows the pseudocode for accessing and manipulating session state. The methods for reading session state ISCLOSED (line 20), ISADJOURNED (line 21) and ISRETIRED (line 22) follow from the discussion earlier and are self-explanatory. The method SETGUARDFLAG repeatedly attempts to set the given guard flag in the session state, if not already set, using a CAS instruction until it succeeds (lines 25-28). The method SETVACANTFLAG attempts to set the vacant flag in the session state using a CAS instruction provided the session has closed and has no participants (lines 31-34). The method MARKASRETIRED sets all the four flags in the session state (line 35).

The following lemma limits the number of times the loop in SETGUARDFLAG method is executed:

Lemma 1. *The while-do loop in* SETGUARDFLAG *method (lines 25-28) is executed only* $O(1)$ *times per invocation of the method.*

Algorithm 9: Additional Data types, variables, & Initialization

```
// add the following field to node structure
    bool condition;   // a boolean flag to indicate whether the node is safe or unsafe

additional shared variables
    hp: array [1...n][1...2] of NodePtr;                    // to store hazard pointers

additional private variables
    pool: array [1...n][1...2][1...3n] of NodePtr;          // to store pools of nodes
    which: array [1...n] of integer;    // to indicate which of the two pools is active
    marker array [1...n] of integer;    // pointer to the first safe node in the active pool

initialization of additional variables
begin
    // initialize shared variables
    foreach i ∈ [1...n], j ∈ [1...2] do  hp[i][j] := null;
    // initialize private variables
    foreach i ∈ [1...n] do
        foreach j ∈ [1...2], k ∈ [1...3n] do
            pool[i][j][k] := new Node;                      // create a new node
            pool[i][j][k] → owner := me;                    // set the owner as myself
        which[i] := 1;                                      // designate pool[i][1] as active
        marker[i] := 1;                                     // designate pool[i][1][1] as the first safe node
```

Proof. A new iteration of the while-do loop is executed only if the CAS instruction performed on the session state fails. The failure occurs only if one of the two guard flags in the session state has been set by another CAS instruction. This can only happen at most two times. □

Lemma 2. *A session can adjourn only after it has closed.*

Proof. For a session to adjourn, both the guard flags (`LEADERLESS` and `CONFLICT`) must be set in the session state. This implies that the session must be closed before it can be adjourned. □

Algorithm 10: Reusing retired nodes.

```
   // changes to READHEAD method - replace line 38 with lines 130-133
130 repeat
131 |   snapshot[me] := head[instance];      // read the current head pointer of the
    |    list
132 |   hp[me][1] := snapshot[me];                  // declare it as a hazard pointer
133 until (snapshot[me] = head[instance]);

   // changes to GETNEWNODE method - replace lines 81-82 with
      lines 134-135
134 NodePtr node := pool[me][which[me]][marker[me]];      // get a safe node from the
    active pool
135 node → condition := UNSAFE;                                  // mark it as unsafe

   // changes to SELECTNEXTNODE method - insert lines 136-137 after
      line 94
136 hp[me][2] := helpee;    // declare reference to the helpee's node as a hazard
    pointer
137 if (announce[snapshot[me] → number] ≠ helpee) then return mine; // request already
    fulfilled

   // changes to APPENDNEXTNODE method - insert line 138 after line 105
138 hp[me][2] := successor;        // declare reference to the successor node as a
    hazard pointer

   // changes to RETIRENODE method - insert lines 139-141 before line 113
139 node → owner := me;                          // claim the ownership of the node
140 pool[me][which[me]][marker[me]] := node;       // replace in case reclaiming the
    predecessor node
141 marker[me] := marker[me] + 1;        // advance the pointer for the safe nodes
```

Managing list head: Algorithm 4 shows the pseudocode for accessing and manipulating list head. The method READHEAD reads the pointer to the current head of the list and stores it in its private variable (line 38). The method TESTHEAD checks whether the head of the list is still the same since it was declared to be a hazard pointer (lines 41-42). The method ADVANCEHEAD advances the head of the list to its successor (line 45).

Algorithm 11: Cleanup algorithm.

```
// used to identify safe nodes in the passive pool; executing the
   method once corresponds to one epoch
CLEANUP( )
begin
    integer other := 3 − which[me];
    foreach i ∈ [1, 3n] do        // mark the condition of all the nodes in the
     passive pool as unknown
        pool[me][other][i] → condition := UNKNOWN;
    foreach i ∈ [1, n], j ∈ [1, 2] do                // scan all the hazard pointers
        NodePtr node := hp[i][j];
        if (node → condition = UNKNOWN) then        // node is in a passive pool
            if (node → owner = me) then                          // I own the node
                if (node → condition = UNKNOWN) then        // node must be in my
                 passive pool
                    node → condition := UNSAFE;        // mark the node as unsafe

    let 𝒮 denote the subset of all nodes in the passive pool whose condition is set to
     UNKNOWN;
    collect all nodes in 𝒮 toward the end of the passive pool using a method similar to the
     partition procedure used in quick sort, which has linear running time, and also change
     their condition to SAFE;
    // start a new epoch
    marker[me] := index of the first safe node in the passive pool;
    which[me] := 3 − which[me];          // switch the designations of the pools
```

4.2.2 Achieving Deadlock-Freedom

Entering critical section: Whenever a process generates a critical section request, it obtains a new node and initializes it appropriately (lines 83-89). Specifically, all flags in the session state are cleared, the number of processes in the session is set to *one*, and the address of the previous and next nodes are set to *null*. The process then repeatedly performs the following steps until it is able to enter its critical section (lines 50-70 in ENTER method):

(1) It locates the current head of the linked list associated with the GME object (line 51).

(2) If the head node (of the list) matches its own node (may happen because of helping described in Section 4.2.3), it retires its predecessor node (line 54) and enters its critical section (line 55). Otherwise, if (i) the session is compatible with its own request, and (ii) the session is open (lines 56-57), it attempts to join the session by incrementing the session size using an FAA instruction (line 58). It then ascertains that the session is still in open state (line 59). If so, it retires its own node (line 60) and enters its critical section (line 61). If not, it aborts the attempt, decrements the session size using an FAA instruction (line 63) and attempts to adjourn the session if possible (line 64). Finally, if the session is not compatible with its own request, it sets the `CONFLICT` flag in the session state (line 66) and attempts to adjourn the session if applicable (line 67).

(3) If it is unable to join the session in the previous step for any reason (*e.g.*, the session was not compatible with its own request or was not open or was closed before it could join), it busy waits for the session state to change to adjourned (lines 68-69).

(4) If the head of the list has not yet moved, then it attempts to establish a new session by appending a new node to the list (line 70).

(5) To append a new node to the list (lines 102-109), it first obtains a node to be used for appending (for now assume its own node) (line 103) and attempts to set the next pointer of the current head to that node using a CAS instruction (line 104). Note that, irrespective of the outcome (of the CAS instruction), a new node is guaranteed to be appended to the list. It then sets the previous pointer and the sequence number of the newly appended node (line 107appendnode:counter). Finally, it attempts to advance the head of the list to the newly appended node using a CAS instruction (line 109). Figure 4.2 shows a snapshot of this process in execution.

The following lemmas characterize the working of the entry section:

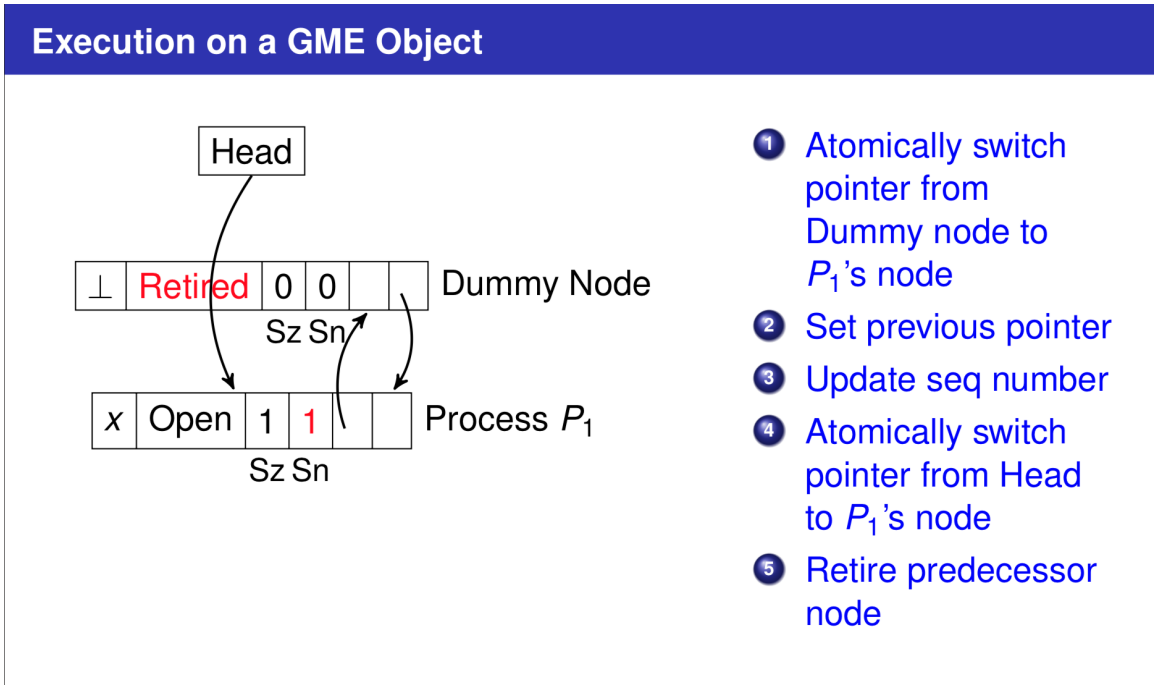


Figure 4.2: Snapshot of a process in execution. Initially, head points to a dummy node. P_1 enters with session request x. Since there are no other processes in the system, P_1 establishes its session in the following sequence of events: it atomically switches the next pointer of the dummny node to point to its node, sets the previous pointer to point to the dummy node, updates the sequence number field in its node (1 + dummy node's number), atomically switches the head pointer to point to its node, and finally, retires the dummy node. Note: For brevity, the node structure only shows the following fields: session, state, size (Sz), sequence number (Sn), prev, and next pointers

Lemma 3. *A process starts executing its critical section as a follower only if the session it joins is compatible with its request and the session is still open after it incremented the session size.*

Proof. After incrementing the session size, a process joins the session (and starts executing its critical section) only after ascertaining that the session is still open. □

Lemma 4. *No new node can be appended to a list until the session associated with the current head of the list has adjourned.*

Proof. Only a process that is unable to join a session tries to append a new node to the list, but only after it has detected that the session has adjourned. □

Leaving critical section: We say that a process enters its critical section as a *leader* if its node is used to establish a new session. Otherwise, we say that it enters as a *follower*. On leaving the critical section, a process performs the following steps (lines 74-78 in EXIT method):

(1) If it owns the head node, then it sets the `LEADERLESS` flag in the session state.

(2) It then decrements the session size using an FAA instruction (line 77).

(3) It finally attempts to adjourn the session if applicable (line 78).

Lemma 5. *If a process has successfully joined a session, then the session cannot adjourn until after it starts executing its exit section.*

Proof. If a process enters its critical section as a leader, then the session size is incremented even before its node is appended to the list. If a process enters its critical section as a follower, then, from Lemma 3, the session was open after the process incrementing the session size.

Clearly, when the session closes, the value of the session size is greater than or equal to the number of processes in the session that are executing their critical sections. And, no process sets the vacant flag in the session state until the session size reaches zero. □

The algorithm is not starvation free since there is no guarantee that a session compatible with the request of the process is ever established.

4.2.3 Achieving Starvation-Freedom

To achieve starvation-freedom, when selecting a node to append to the list, we use the *helping* mechanism used in many wait-free algorithms. This requires making changes to GETNEWNODE, SELECTNEXTNODE, APPENDNEXTNODE and RETIRENODE methods.

After obtaining a new node and initializing it (lines 81-89 in GETNEWNODE method), the process *announces* its request to other processes by storing the node's address in a shared array, which has one entry for each process, denoted by *announce* (line 90).

When selecting a node to establish a new session (SELECTNEXTNODE method), instead of always choosing its own node (line 93), it selects another process to help and chooses its node if the helpee process has an outstanding request (line 95) for the same GME object (line 96) and the node has not been retired yet (line 97).

We use a simple *round-robin* scheme to determine which process to help by storing a sequence number in every node. Every time a new node is appended to the list (lines 102-109 in APPEND-NEXTNODE), the sequence number of the (appended) node is set to one more than that of its predecessor using modulo n arithmetic (line 108).

Finally, in RETIRENODE, the process also revokes its announcement by clearing its entry in *announce* array (line 112).

Lemma 6. *At the time a node is appended to the list, the request associated with the node is (a) for the GME object that owns the list and (b) still outstanding.*

Lemma 7. *After a process has announced its request, at most $n+1$ new sessions can be established until its request is fulfilled.*

Proof. Every time a new node is appended to the list and its head pointer updated, the sequence number in the new head node of the list is incremented by one using module n arithmetic. Let the sequence number of the head node when a process, say p_i with $i \in [1 \ldots n]$, announces its request be x. Among the next n values, given by $\{(x+1) \mod n+1\}$, $\{(x+2) \mod n+1\}$, ..., $\{(x+n) \mod n+1\}$, at least one value matches i. Clearly, when the sequence number of the head node reaches i, every process that tries to append a new node to the list chooses the node for p_i as the one to append unless it is already retired. □

Figure 4.3 shows a snapshot of the helping mechanism in execution. The next step for P_3 is to atomically switch pointers to P_2's node and update the sequence number.

Note that the algorithm is still space-inefficient since a new node is allocated for every request.

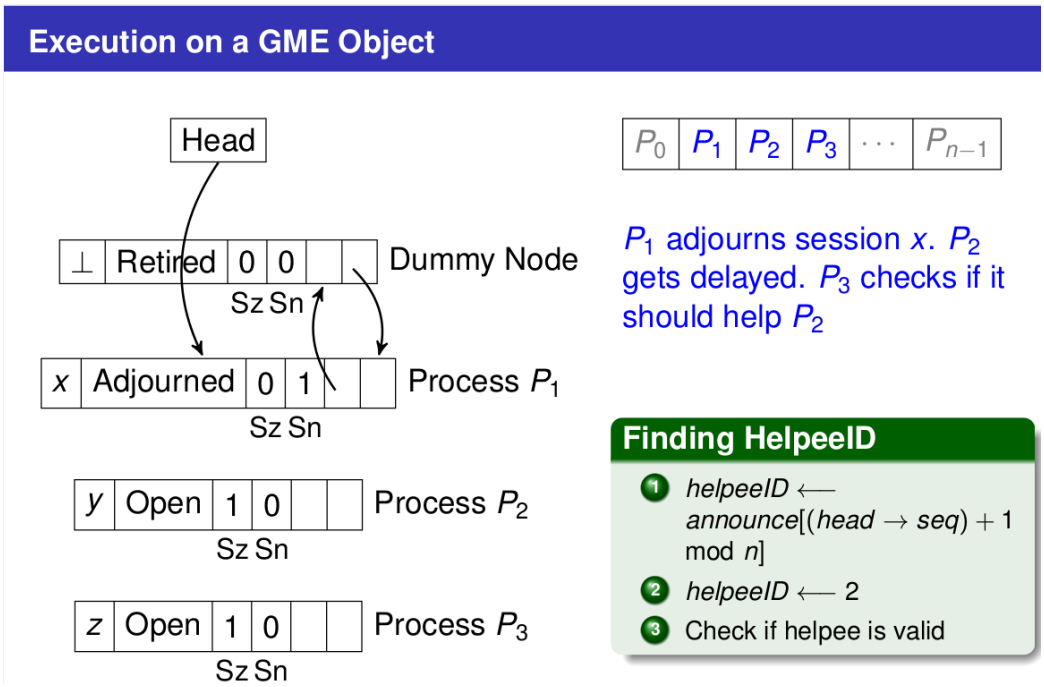


Figure 4.3: Snapshot of a process in execution during helping. Announce array is shown in top right corner. P_1, P_2, and P_3 have announced their nodes to other processes. Once P_1 adjourns the session, P_2 and P_3 compete to establish their session. If P_2 gets delayed, P_3 checks if it should help P_2. Note: For brevity, the node structure only shows the following fields: session, state, size (Sz), sequence number (Sn), prev, and next pointers

4.2.4 Correctness Proof

In this section, unless explicity mentioned, we focus on a single GME object. Our correctness proof easily carries over to multiple GME objects. We first prove the group mutual exclusion property.

Theorem 1 (group mutual exclusion)**.** *The GME algorithm satisfies the group mutual exclusion property.*

Proof. Lemma 3 implies that only those processes whose request is compatible with the session can join the session and execute their critical sections within the session. Lemma 5 implies that, as long as a process is executing its critical section within a session, the session cannot adjourn.

Finally, Lemma 4 implies that no new session can be established until the current session has adjourned. □

We next prove the bounded exit property.

Theorem 2 (bounded exit)**.** *The GME algorithm satisfies the bounded exit property.*

Proof. The body of exit section (EXIT method) includes up to one invocation of READHEAD, SETGUARDFLAG and SETVACANTFLAG methods. Only the second method contains a loop; Lemma 1 implies that the loop is only executed $O(1)$ times. □

We now prove the concurrent entering property. To that end, we start by establishing some properties of our GME algorithm.

Lemma 8. *An open session can close only if the system contains a conflicting request.*

Proof. For a session to close, `CONFLICT` flag in the session state must be set. The flag can only be set by a process whose request conflicts with the current session. □

As part of joining a session as a follower, a process first increments the session size and then rechecks if the session is still open. If not, it decrements the session size immediately without executing its critical section. We refer to such an increment as *spurious*. Note that spurious increments may prevent a session from moving to adjourned state.

Lemma 9. *A process spuriously increments the size of a session at most once.*

Proof. After performing a spurious increment followed by a matching decrement, a process busy waits until the session has adjourned. □

Lemma 10. *A process spuriously increments the size of a session only if some other process in the system has a request that conflicts with its own request.*

Proof. Note that if the increment of session size turns out to be spurious then it implies that the session closed *after* the increment step but *before* the decrement step. Lemma 8 implies that the system has a conflicting request at the point the session closed. □

We consider an iteration of a while-do loop to start just after the boolean condition is evaluated and end just after the boolean condition is evaluated next or the loop is quit, whichever case applies.

Lemma 11. *Consider an execution of one iteration of the outer while-do loop at lines 50-70 in the entry section. At the end of the iteration, either the process joins the current session or a new session is established.*

We say that a system state is *homogeneous* if no two requests, current or future, are for different sessions. Note that homogeneity is a *stable* property; once the system enters a homogeneous state, it stays in a homogeneous state.

Lemma 12. *Once the system reaches a homogeneous state, at most one new session can be established thereafter.*

Lemma 13. *Assume that the system is in a homogeneous state when a process starts executing an iteration of the outer while-do loop at lines 50-70. Then the process executes the body of the while-do loop at most twice.*

Proof. Follows from Lemma 6, Lemma 11, and Lemma 12. □

For the next lemma, we first define some notation. Given a node U, let $s(U)$ denote the session hosted by U. Also, given a request ρ, let $s(\rho)$ denote the session that ρ wants to join.

Lemma 14. *Assume that the system is in a homogeneous state at the beginning of an iteration of the inner while-do loop at line 68. Then the process executes the body of the while-do loop at most once.*

Proof. Let p denote the process executing the loop mentioned in the lemma statement, t_0 the time at which it starts executing the current iteration and ρ the outstanding request of p at time t_0. Also, let H denote the head of the list when p starts executing the iteration, and U the head of the list read by p using the READHEAD method most recently before time t_0. Note that, by assumption, the system is in a homogeneous state at time t_0, which, in turn, implies that there is no outstanding request at time t_0 that conflicts with ρ. There are two cases to consider:

Case 1 ($U \neq H$): This implies that U is an *old* head node of the list and $s(U)$ is already adjourned at time t_0.

Case 2 ($U = H$): We claim that $s(H) \neq s(\rho)$. Otherwise, $s(H)$ is already closed at time t_0. Lemma 8 implies that there exists an outstanding request σ at time t_0 such that $s(\sigma) \neq s(H)$. This is turn implies that ρ and σ are both outstanding requests at time t_0 with $s(\rho) \neq s(\sigma)$—a contradiction. Thus, for the rest of this case, assume that $s(H) \neq s(\rho)$.

Now, let q denote the last process to leave $s(H)$; q exists because there are no current or future requests for $s(H)$ at time t_0. Note that both p and q invoke the SETVACANTFLAG method—p after setting the `CONFLICT` flag in the state field of H and q after decrementing the size field of H. Let $r \in \{p, q\}$ denote the process that invoked the method *later*. Note that, when r invokes the SETVACANTFLAG method, the following must hold: (a) both the `LEADERLESS` and `CONFLICT` flags are already set in the state field of H, and (b) the size field of H is zero and stays zero thereafter (*i.e.*, it does not undergo any spurious increments). The latter holds because, otherwise, it would imply that exists a pending request σ at time t_0 such that $s(\sigma) = s(H) \neq s(\rho)$. In other words, ρ and σ are both outstanding requests at time t_0 with $s(\rho) \neq s(\sigma)$—a contradiction. Clearly, when the SETVACANTFLAG method invoked by r returns, the session hosted by H is guaranteed to be adjourned.

It now remains to argue that the method returns before time t_0. If $r = p$, then it follows trivially from the code. If $r = q$, then, by definition, the system cannot be in homogeneous state until q has finished executing its exit section.

In both cases, $s(U)$ is guaranteed to be in adjourned state at time t_0 and thus p is guaranteed to quit the loop after completing the current iteration. □

We are now ready to prove the concurrent entering property.

Theorem 3 (concurrent entering)**.** *The GME algorithm satisfies the concurrent entering property.*

Proof. Lemma 1 and Lemma 14 imply that, once the system is in a homogeneous state, a process finishes executing an iteration of the outer while-do loop of its entry section within a bounded number of its own steps. The property then follows from Lemma 13. □

For the lockout freedom property, we need the following additional lemmas.

Lemma 15. *If the system contains a conflicting request while a session is in progress, then the session eventually closes.*

Proof. All processes with a conflicting request eventually invoke SETGUARDFLAG method to set `CONFLICT` flag in the session state, which terminates only after the flag has been set. Further, when the leader of the session leaves its critical section, it invokes SETGUARDFLAG method to set `LEADERLESS` flag in the session state, which terminates only after the flag has been set. □

Lemma 16. *Once a session is closed, its size can be incremented spuriously at most n times.*

Proof. Each process is responsible for at most one spurious increment to the session size. □

Lemma 17. *Once a session is closed, eventually the session size becomes zero and stays zero thereafter.*

Proof. After a session closes, no new process can join the session. Every process that is in the session at the point the session closes eventually leaves the session. The result then follows from Lemma 16. □

Lemma 18. *A closed session is eventually adjourned.*

Proof. Whenever a process either sets one of the guard flags in the session state or decrements the session size, it attempts to set the vacant flag afterward. The result then follows from Lemmas 1-17. □

Lemma 19. *Once a session is adjourned, a new session is eventually established.*

Proof. A session closes (and hence adjourns) only if there is a conflicting request in the system. Clearly, this implies that, after a session is adjourned, at least one process in the system tries to append a new node to the list (and establish a new session). □

Finally, we have

Theorem 4 (lockout freedom)**.** *The GME algorithm satisfies the lockout freedom property.*

Proof. As long as a process has an outstanding request, Lemma 15, Lemma 18, and Lemma 19 imply that eventually either the process is able to join the session or the current session is adjourned and a new session is established. The lockout freedom then follows from Lemma 7. □

4.2.5 Complexity Analysis

In this section, as in the previous section, unless explicity mentioned, we focus on a single GME object. Our complexity analysis easily carries over to multiple GME objects.

Theorem 5 (worst case context switch complexity)**.** *The context switch complexity of a passage* π *is at most* $\min\{\bar{c}(\pi), n\} + 1$ *in the worst case.*

Proof. Lemma 7 implies that at most $n+1$ new sessions can be established after a process has announced its request and before it is able to enter its critical section. Moreover, if a new session is established while a process is waiting to enter its critical section, then, clearly, the leader of that session has a request whose passage overlaps with that of the given process. □

The main result in [27] implies that

Theorem 6 (amortized case context switch complexity)**.** *The context switch complexity of a passage π is at most $\dot{c}(\pi)+1$ in the amortized case.*

Lemma 20. *Let s denotes the number of sessions that overlap with the entry section of a process. Then the process performs only $O(s)$ remote references in its entry section.*

Proof. Lemma 11 implies that a process performs at most s iterations of the outer while-do loop at lines 50-70 in its entry section. In every iteration, a process performs at most $O(1)$ instructions outside of the inner while-do loop at lines 68-69. While spinning in the inner-while loop, it reads the contents of the session state (of the node pointed to by the head pointer) repeatedly. The session state consists of four flags which, once set, are never reset (assuming no memory reclamation). Thus reading the session state repeatedly in the loop is also responsible for only $O(1)$ remote references per list node. □

Lemma 21. *A process performs only $O(1)$ remote references in its exit section.*

Proof. The only loop in the exit section is in SETGUARDFLAG method, which is invoked only once. The result then follows from Lemma 1. □

Theorem 7 (RMR complexity)**.** *The RMR complexity of entry and exit sections of a passage π is $O(\min\{\bar{c}(\pi), n\})$ in the worst-case and $O(\dot{c}(\pi))$ in the amortized case.*

Proof. The number of sessions that overlap with the entry section of a process is upper-bounded by one plus the context-switch complexity of the corresponding passage. The result then follows from Theorem 5, Theorem 6, Lemma 20, and Lemma 21. □

Theorem 8 (concurrent entering step complexity)**.** *The maximum number of steps a process has to execute in its entry and exit sections provided all current and future requests are for the same session is $O(1)$.*

Proof. Assume that the system is in a homogeneous state. We first analyze the entry section of the process. Lemma 14 implies that, during one iteration of the outer while-do loop, the process executes only $O(1)$ iterations of the inner while-do loop. Further, Lemma 13 implies that, the process executes only $O(1)$ iterations of the outer while-do loop. Thus, the process executes only $O(1)$ steps in its entry section after the system has entered a homogeneous state. Clearly, a process executes only $O(1)$ in its exit section. □

4.3 Achieving Space Efficiency

To achieve space efficiency, we describe a way to reuse/recycle nodes in a safe manner while adding only $O(1)$ steps to each passage.

Consider a node that is used to establish a new session. Note that the session may stay "active" long after the owner of the node (also the leader of the session) has left its critical section. To address this issue, the leader of a session, on leaving its critical section, relinquishes the ownership of its node and instead claims the ownership of its predecessor node. This is similar to the approach used in the well-known queue-based mutual exclusion algorithm presented in [14, 49]. On the other hand, if a process joins a session as a follower, it retains the ownership of its node. In both cases, the node is considered to be retired and is not used to establish a new session.

Claiming the ownership of the predecessor node: Claiming the ownership of the predecessor node is relatively straightforward in [14, 49] because, unlike in our algorithm, only one process is holding a reference to the predecessor node when it is reclaimed. In our algorithm, multiple processes may be holding a reference to the predecessor node because of the helping mechanism used to achieve starvation-freedom. Note that a node may be appended to the list by any process in the system and not necessarily by its owner.

Note that, when a node is appended to the list, we store a pointer at the node to its predecessor. When the owner of the appended node (also the leader of the session associated with the node)

leaves its critical section, it can use this pointer to access the predecessor node and claim it as its own node.

Reusing a retired node: To determine when it is safe to reuse a node, we use a variant of the well known memory reclamation technique based on *hazard pointers* first presented in [52]. The technique works as follows. Each process maintains information about the set of objects it is dereferencing currently or will dereference in the future, and hence it is "hazardous" to reclaim their memory. A process can reuse or recycle an object only if no process has declared it as a hazard pointer. To declare a hazard pointer, a process performs the following sequence of steps repeatedly until it succeeds: it first reads the address of the node it wishes to dereference, it then writes the address to a shared location (visible to other processes) and finally ascertains that the node still needs to be dereferenced. If the validation succeeds in the last step, then it implies that the address of the node has been successfully declared as a hazard pointer.

The above algorithm increases the step complexity of an operation by $O(1)$ in the amortized case but $O(n)$ in the worst-case (assuming that each process holds only $O(1)$ hazard pointers). One disadvantage of the algorithm is that, when used to manage memory in a concurrent algorithm with wait-free operations, it weakens the progress guarantee of an operation from wait-freedom to lock-freedom. Aghazadeh *et al.* improve upon the above memory reclamation algorithm in two ways in [4]. First, their algorithm increases the step complexity of an operation by only $O(1)$ in the *worst-case*. Second, it does not degrade the progress guarantee of the underlying concurrent algorithm; wait-free operations remain wait-free.

In our case, the mechanism used in the previous section to achieve starvation-freedom is not impacted by the memory reclamation algorithm based on hazard pointers. Thus we only focus on the lock-free version of Aghazadeh *et al.*'s algorithm that guarantees the first property only, which works as follows. Each process maintains a pool of $\Theta(n)$ objects; the ownership of an object is fixed and does not change at run time. To identify which objects in its pool can be reused, a process

scans the hazard pointers of processes in a *lazy* manner; specifically, during each operation, it scans the hazard pointer(s) of only one process. An object can be reused if the following two conditions hold: (a) it was retired before the last n operations and (b) no process was found to hold a reference to it in its list of hazard pointers during the last n operations.

Note that Aghazadeh *et al.*'s algorithm cannot be directly used to manage memory of nodes in our case because, in our GME algorithm, the ownership of a node may change over time, and some nodes, namely the head nodes of lists, are not owned by any process but by their respective objects. We adapt the lock-free version of their algorithm to work in our case, while adding only $O(1)$ cost to the complexity of each passage, as follows.

Our lazy memory reclamation approach: Algorithm 10 shows the changes/additions we made to the pseudocode in Algorithms 2-8 to reclaim memory of retired nodes and, thus, achieve space efficiency.

We say that a retired node has become *safe* if no process was found to hold a reference to it as a hazard pointer (and thus can be reused to establish a new session); otherwise, we say that it is *unsafe*. A node now contains an additional field, namely *condition*, to indicate the status of the node with respect to memory reclamation—`SAFE`, `UNSAFE` or `UNKNOWN` (line 114).

To enable memory reclamation, each process maintains a small number of (specifically, two) hazard pointers in an array with one entry for each process, denoted by *hp* (line 116). Hazard pointers of a process contain the addresses of the following nodes associated with its current request: (i) the last known head of the list (line 132), and (ii) its successor—potential (line 136) or actual (line 138). Each process also maintains the following private variables: (a) two disjoint pools of nodes, each consisting of $3n$ nodes (line 118), (b) which of the two pools is active, *i.e.*, currently used to service requests (line 119), and (c) the index of the first safe node in the active pool (all nodes that are safe to resue are guaranteed to be stored toward the end in the pool) (line 120).

The method READHEAD now works as follows: it repeatedly reads the pointer to the current head of the list, declares it as a hazard pointer and then validates the reference, until the validation succeeds (lines 130-133).

The method SELECTNEXTNODE now includes statements to declare the reference to the helpee node as a hazard pointer, followed by its validation lines 136-137.

The method APPENDNEXTNODE now includes a statement to declare the reference to the successor node as a hazard pointer line 138, which is validated at line 106.

The execution of a process is divided into *epochs*. Each epoch consists of exactly n passages. During an epoch, one of the pools is designated as *active*, while the other is designated as *passive*. Intuitively, during an epoch, the active pool is used to service critical section requests (line 134), whereas the passive pool is processed incrementally (in lazy manner) to identify at least n safe nodes to service requests in the next epoch (lines 144-156). The designation is switched at beginning of each epoch. To identify the subset of nodes in its passive pool that are reusable, a process first sets the condition field of all the nodes in the passive pool to `UNKNOWN` (lines 144-146). It then scans the hazard pointers of all processes (lines 147-148) and changes the condition field of any node whose condition field is currently set to `UNKNOWN`, that is owned by it and reference to which has been declared as a hazard pointer to `UNSAFE` (lines 149-152). It next collects all nodes in the passive pool whose condition field is still set to `UNKNOWN` towards the end of the pool and also changes their condition field to `SAFE` (lines 153-154). Finally, it switches the designation of the two pools (lines 155-156).

To complete the memory reclamation algorithm, we make changes to two more methods. In the GETNEWNODE method, a node is obtained from the active pool and its condition field is set to `UNSAFE` (lines 81-82). Finally, in the RETIRENODE method, when a process releases the ownership of a node and acquires the ownership of another node (the predecessor of the current head) (line 139), it replaces the former node with the latter node in its active pool.

Our memory reclamation algorithm satisfies the following properties. First, a retired node is deemed to be safe to reuse only after none of the processes has declared it as a hazard pointer *after*

the node was retired. Second, a node belongs to *at most one* pool. Third, the condition of a node is set to `UNKNOWN` *only if* the node belongs to a passive pool. Fourth, if the node belongs to a pool, then its *current owner* information is available in the node's owner field.

The first property helps to guarantee that, once a process has validated a reference after declaring it as a hazard pointer, the node associated with the reference cannot be reused as long as it is declared to be hazard pointer and, thus, the starvation-free GME algorithm does not interfere with the memory reclamation algorithm. The last three properties help to guarantee that a process modifies the condition field of a node at line 152 only if the node belongs to its own passive pool. The relevant section of the pseudocode is from lines 149-152. Consider a process p executing the CLEANUP method as part of some epoch. The first if-statement checks that the condition field of the node is set to `UNKNOWN`. This implies that the node is in the passive pool of some process, say q. Note that q may be different from p. The second if-statement checks that the owner field of the node is set to p. But both if-statements may also evaluate to true if the node has migrated from the passive pool of q to the active pool of p since the first if-statement was evaluated (recall that the CLEANUP method is executed incrementally). In this case, however, the condition field of the node is guaranteed to be set to `UNSAFE` because the node will stay in the active pool until the end of the epoch. Thus, the third if-statement ensures that the node is indeed in the passive pool of p.

Step complexity analysis: Note that the CLEANUP method can be executed in $O(n)$ steps because a pool contains $3n$ nodes and each process only holds two hazard pointers. By setting the size of each pool to $3n$, we can ascertain that, by the end of an epoch, a process is able to identify at least n reusable nodes in its passive pool. Clearly, a passive pool can be processed in an incremental manner such that only $O(1)$ steps are added to each passage of a process contained in its epoch.

The only change to a method that may increase the step complexity (asymptotically) is the one made to the READHEAD method since it now contains a loop; all other changes only add

$O(1)$ steps to their respective methods. We first bound the total number of times the loop in the READHEAD method is executed over *all* invocations in the entry section of a process.

The following lemma limits the number of times the loop in READHEAD method is executed:

Lemma 22. *The number of times the repeat-until loop in* READHEAD *method (lines 130-133) is executed is bounded by one plus the number of nodes appended to the list during the loop execution.*

Proof. A new iteration of the repeat-until loop is executed only if the process finds that the pointer to the head node of the list has changed. □

Lemma 23. *Let s denote the total number of sessions that overlap with the entry section of a process. Further, let ℓ_i denote the number of iterations of the repeat-until loop at lines 130-133 executed by a process in the ith invocation of the* READHEAD *method in its entry section. Then, we have*

$$\left(\sum_i \ell_i\right) \quad \leq \quad 2s$$

Proof. The result follows from Lemma 11 and Lemma 22. □

Lemma 24. *In any invocation of the* READHEAD *method at line 73 in the exit section of a process, the repeat-until loop at lines 130-133 is executed only once.*

Proof. Note that the head of the list cannot advance until after all processes that joined the session have also left the session. □

All lemma and theorem statements in Section 4.2.4 and Section 4.2.5 still hold. The only proof that needs to be modified is for Lemma 20; it particular, it needs to incorporate the result of Lemma 23 (but the statement still holds).

Space complexity analysis: Finally, we analyze the space complexity of our GME algorithm considering that the system may contain multiple GME objects and a process may hold locks on multiple GME objects at the same time. Note that our GME algorithm still works without any modification even if a process needs to hold lock on multiple GME objects at the same time.

Theorem 9 (multi-object space complexity)**.** *The space complexity of our GME algorithm is $O(m+n^2+n\ell)$ space, where n denotes the number of processes, m denotes the number of GME objects and ℓ denotes the maximum number of locks a process needs to hold at the same time.*

Proof. Our algorithm uses only $O(m+n^2)$ space for managing m GME objects, where $O(n^2)$ space is shared among all m GME objects. In addition, each process needs only $O(\ell)$ space, where ℓ denotes the maximum number of GME objects (or locks) a process needs to hold at the same time. □

Theorem 10 (bounded space variables)**.** *Our GME algorithm only uses bounded space variables.*

CHAPTER 5

EXPERIMENTAL EVALUATION

In this section, we present our experimental results of evaluating different GME algorithms.

5.1 Different Group Mutual Exclusion Algorithms

We compare the performance of the following implementations of GME algorithms:

(a) the GME algorithm proposed by Bhatt and Huang [7], which is based on f-array data structure [40], denoted by BH-GME,

(b) the GME algorithm proposed by He *et al.* [30], which is a generalization of the classical Lamport's Bakery algorithm, denoted by GLB-GME, and

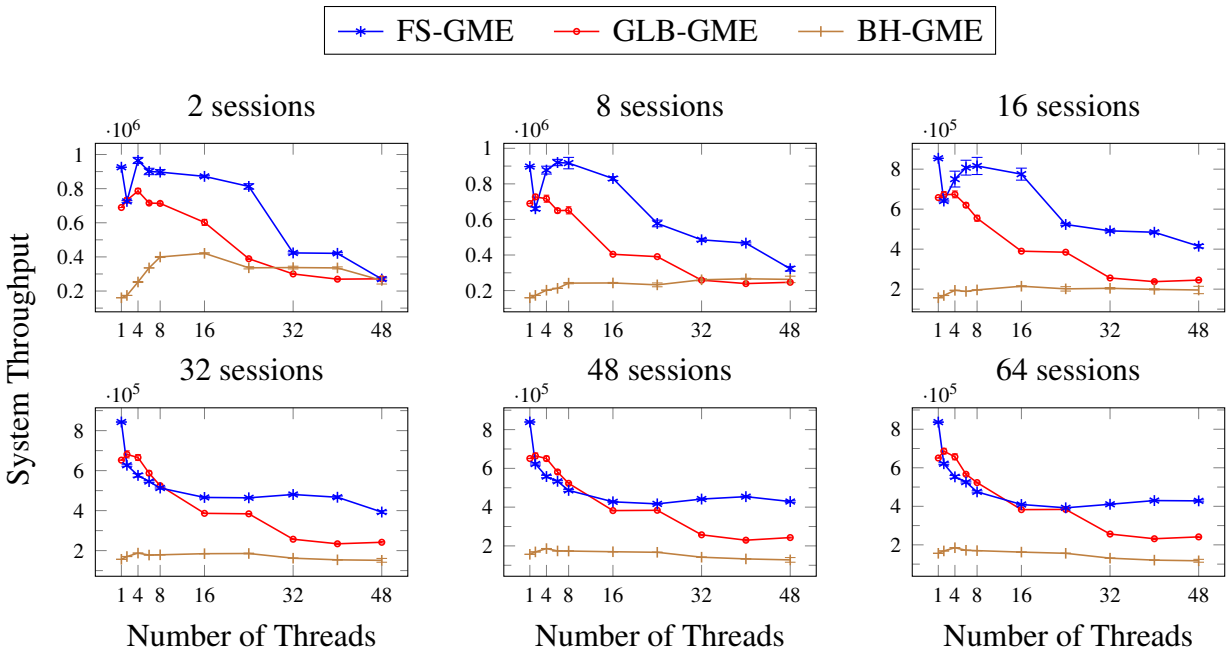
(c) the GME algorithm presented in this work, denoted by FS-GME.

We chose GLB-GME and BH-GME for comparison due to the following reasons. First, to our knowledge, BH-GME has the best RMR complexity among all existing GME algorithms, and GLB-GME is the most recently proposed GME algorithm. Second, both algorithms satisfy the First-Come-First-Serve (FCFS) property—relaxed in the case of BH-GME and strict in the case of GLB-GME. Additionally, BH-GME also satisfies the pulling property. Third, BH-GME uses load-linked and store-conditional (LL/SC) RMW instructions whereas GLB-GME does not use any RMW instruction.

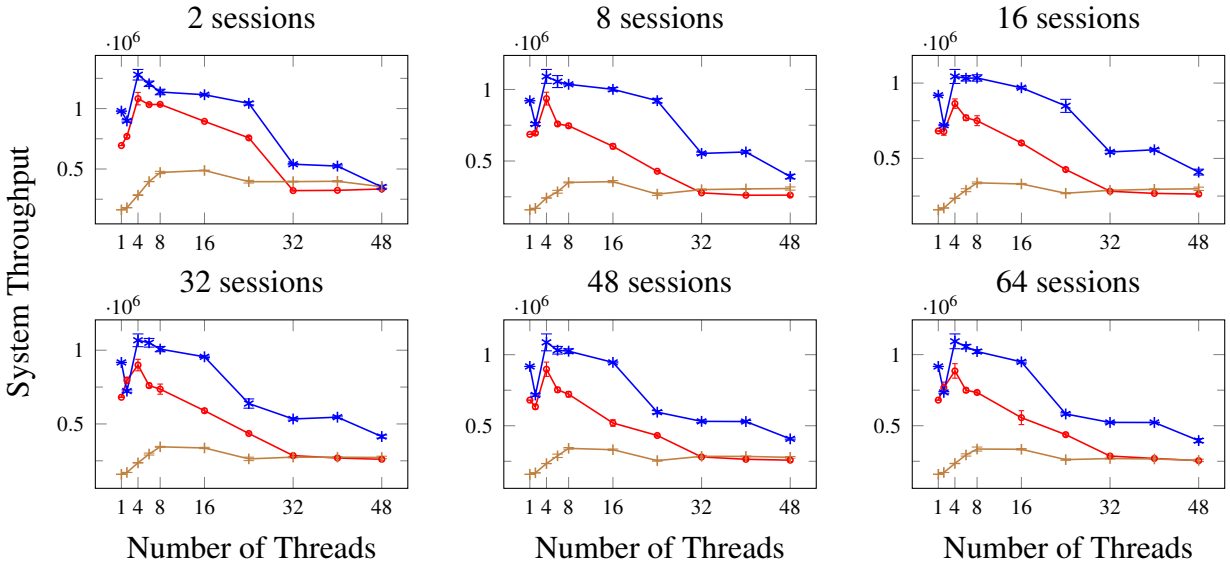
To our knowledge, no current implementations of GLB-GME and BH-GME exist (confirmed with the authors) so we implemented them ourselves. All implementations were written in C/C++.

5.2 Experimental Setup

System used: We conducted our experiments on a dual socket Intel Xeon E5-2690 v3 processor consisting of 12 2.6 GHz cores per socket with hyper-threading enabled and 64GB RAM. We used g++ compiler with optimization flags set to -O3.



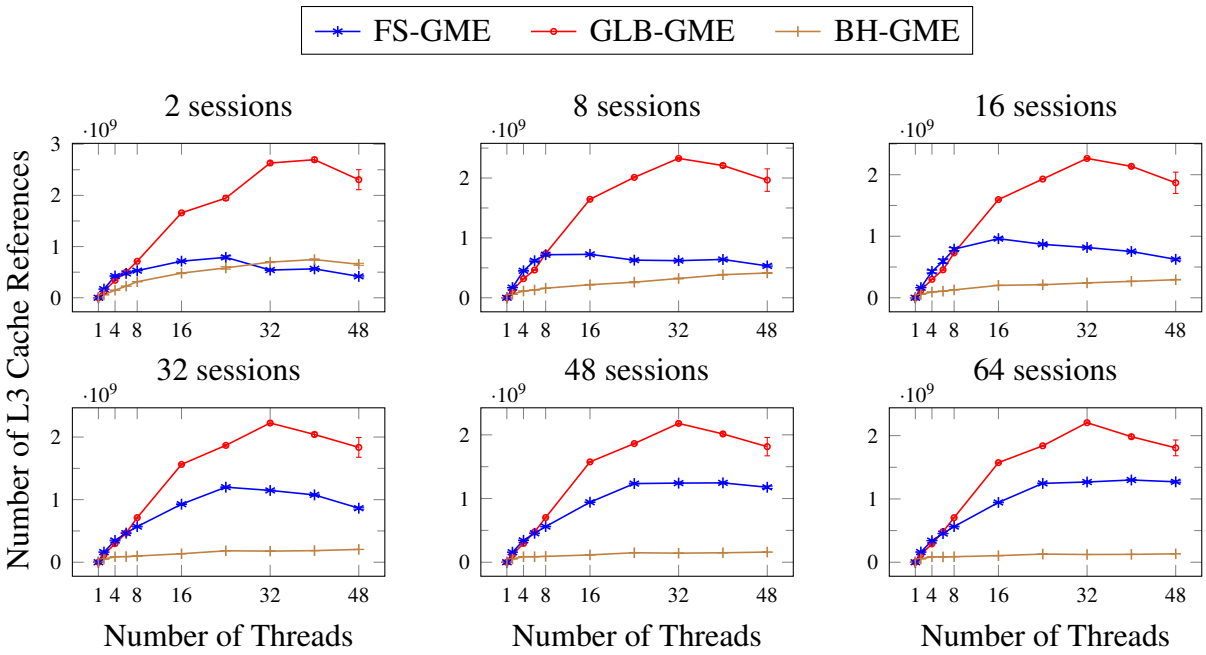
(a) Uniform session distribution



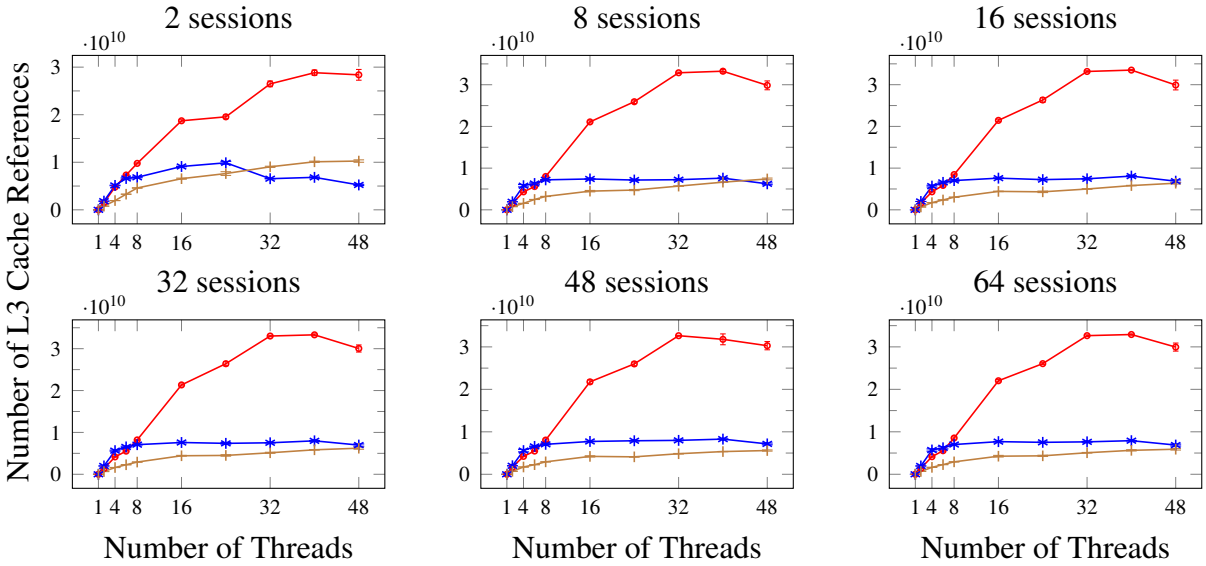
(b) Non-uniform session distribution

Figure 5.1: Comparison of system throughput of different algorithms. Higher the throughput, better the performance of the algorithm.

Experimental parameters: To comparatively evaluate different implementations, we considered the following parameters:



(a) Uniform session distribution

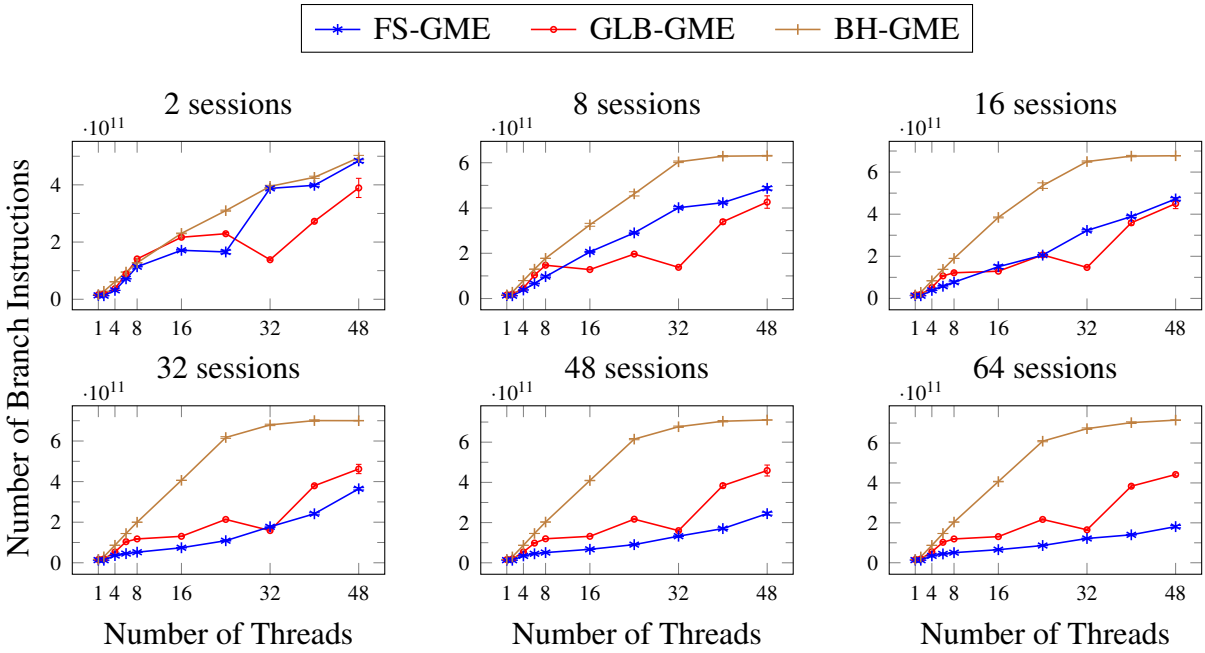


(b) Non-uniform session distribution

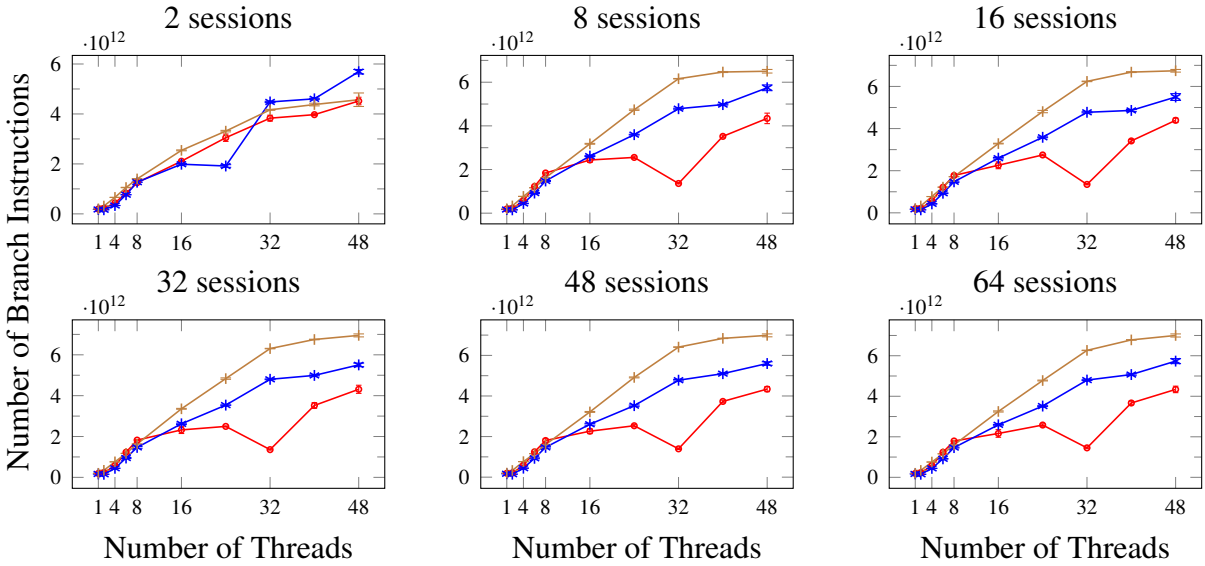
Figure 5.2: Comparison of L3 cache references of different algorithms.

1. **Number of Different Sessions:** We considered six different values of 2, 8, 16, 32, 48 and 64.

2. **Distribution of Sessions:** We considered two different session distributions: (a) *uniform:* all session types are requested with the same probability. (b) *non-uniform:* different session types



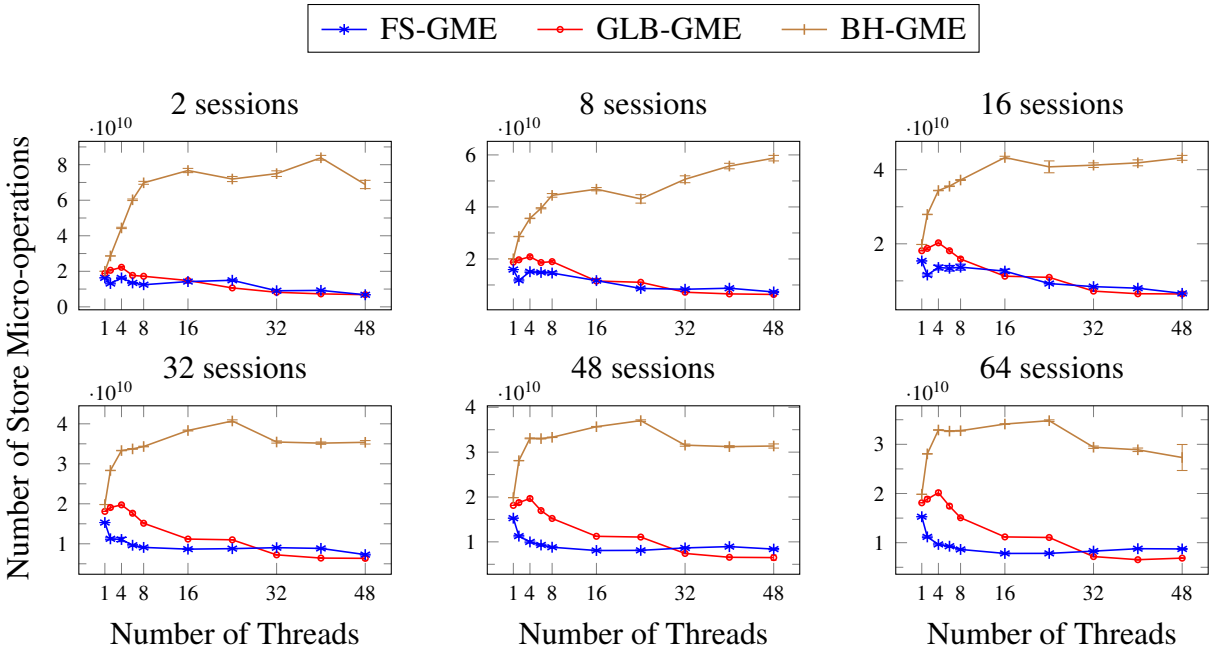
(a) Uniform session distribution



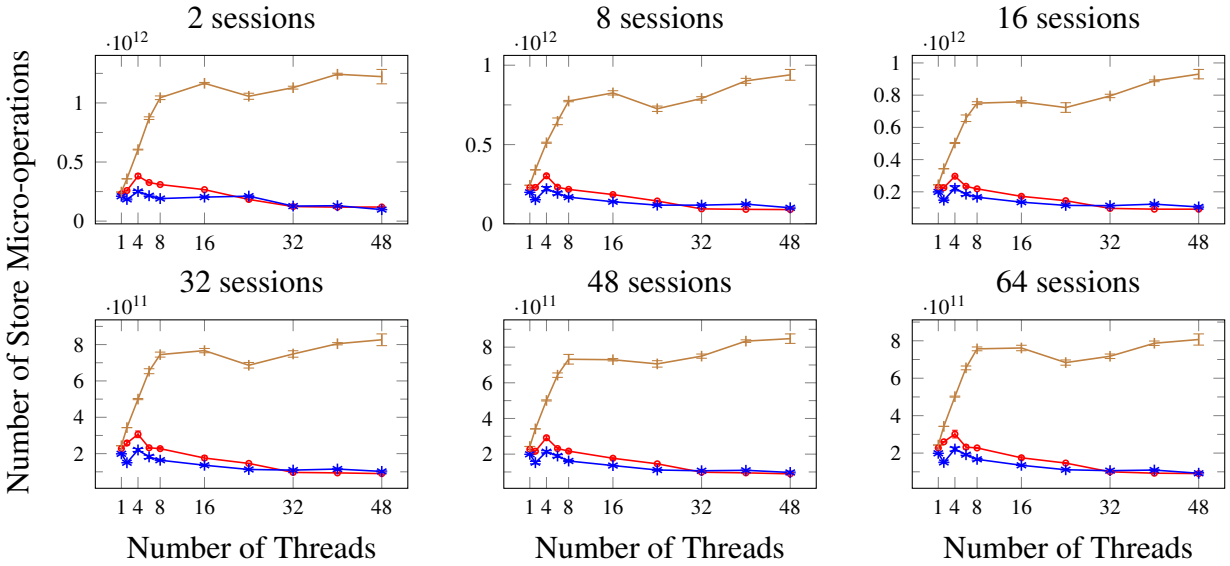
(b) Non-uniform session distribution

Figure 5.3: Comparison of branch instructions of different algorithms.

are requested with different probabilities. In our experiments, we assumed that two session types are requested 90% of the time and the remaining 10% of the time (90/10 distribution) [57].



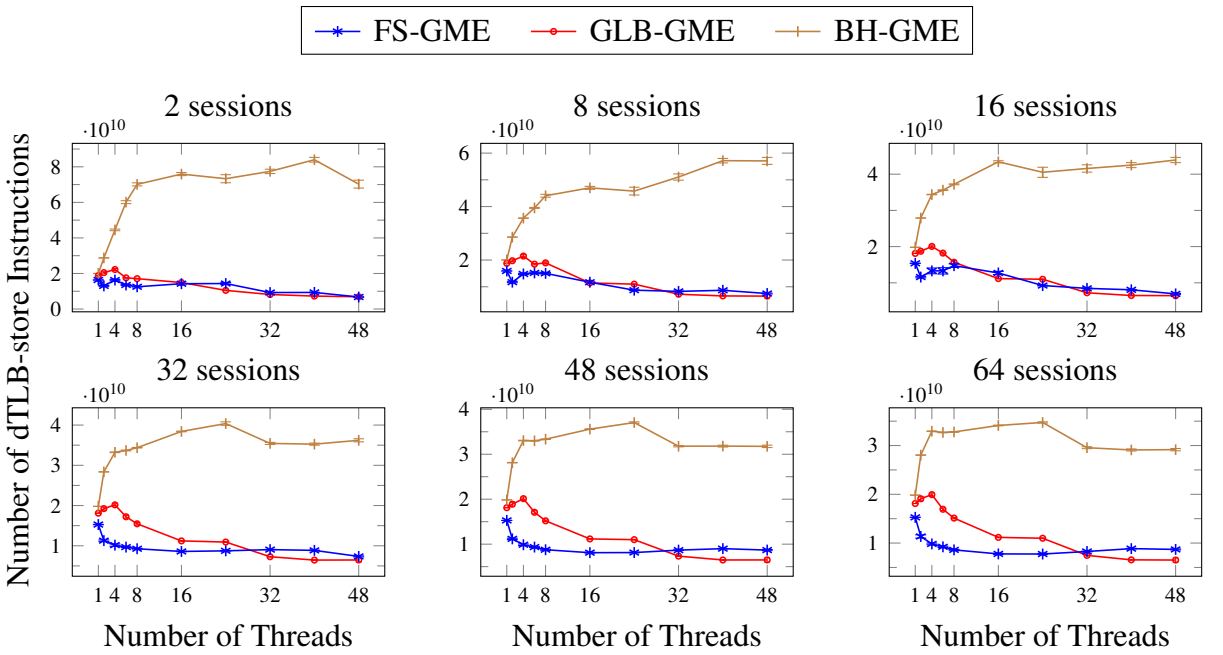
(a) Uniform session distribution



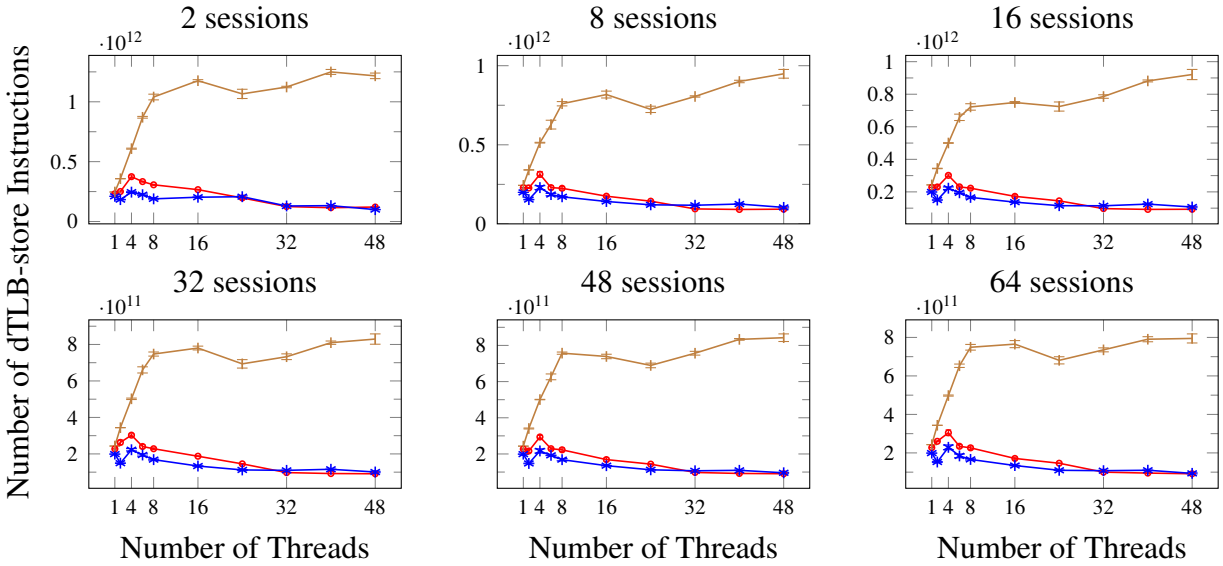
(b) Non-uniform session distribution

Figure 5.4: Comparison of store micro-operations of different algorithms.

3. **Maximum Degree of Contention:** This depends on number of threads that can concurrently request entry to their critical sections. We varied the number of threads from 1 to 48 in suitable increments.



(a) Uniform session distribution



(b) Non-uniform session distribution

Figure 5.5: Comparison of data TLB store instructions of different algorithms.

Testing framework: In each run of the experiment, every thread repeatedly generated requests for a (single) GME lock. Upon obtaining the lock, in its critical section, each thread executed an RMW instruction (FAA) on one shared variable and a simple write instruction on a certain number

of local variables (chosen randomly between 1 and 100 each time). The non-critical section was essentially empty.

Run details: For the uniform distribution, each experiment was run for eight seconds and the results were averaged over ten runs. For the non-uniform distribution, each experiment was run for two minutes and the results were averaged over five runs. Longer running time was required to conform to the desired probability distribution. To generate random numbers, we used the Mersenne Twister pseudo-random number generator. For both experiments, each run had a two second "warm-up" phase whose numbers were excluded from the calculations to minimize the effect of initial caching on the computed statistics.

Evaluation metric: We compared the performance of different implementations with respect to *system throughput*, which is given by the number of critical section executions completed per unit time.

5.3 Results

Figure 5.1 depicts the system throughput of the three GME algorithms for the parameter values discussed above. As the graphs clearly show, FS-GME outperformed the other two GME algorithms in *almost all* the cases. The difference was really stark at medium and larger thread count values when the throughput of FS-GME was sometimes as much as 189% (almost three times) higher than the next best performer. Even though, BH-GME has the lowest (worst-case) RMR complexity among the three algorithms, it had the worst performance.

To understand the reasons for the differences in the performance, we used Linux performance analyzing tool `perf`. Figure 5.2 shows the number of L3 cache references generated by the three GME algorithms. Also, Figure 5.3 shows the number of branch instructions generated by the three GME algorithms.

Recall that GLB-GME has $\Omega(n)$ RMR complexity. In the entry section of GLB-GME, a thread examines the request of every other thread and busy waits on that request to complete if it conflicts with its own and has a higher priority. As the graphs in 5.2 confirm, GLB-GME generates significantly larger number of L3 cache references than the other two algorithms and, moreover, the gap grows with the number of threads. Also, note that, as either the number of threads or the number of different sessions increases, the probability that requests of different threads conflict also increases. Recall that GLB-GME satisfies the strict FCFS property. Joung proved analytically in [43] that, as the likelihood of conflicts increases, a GME algorithm that satisfies strict FCFS property will degenerate to a traditional ME algorithm in which only one thread is able to execute its critical section at a time.

In the entry section of BH-GME, a thread has to perform many checks before it can enter its critical section. As the graphs in Figure 5.3 show, the execution history of BH-GME exhibited higher branching compared to that of FS-GME and GLB-GME. Excessive branching is undesirable and may adversely impact the performance of an algorithm significantly because branching inhibits many of the compiler and hardware optimizations. BH-GME uses the f-array data structure to implement a global counter and a wait-free queue. f-array has a tree-like structure. Whenever there is any change to the queue or global counter, that change is propagated from the leaf node to the root node invalidating cached values of these nodes. Thus, BH-GME also exhibits very high cache activity which is evident in the graphs for Figure 5.4. This graph shows the number of store micro-operations across the entire cache hierarchy. Further evidence of this is indicated in Figure 5.5 which shows the graphs for store operations in the data TLB. Finally, `perf-record` and `perf-annotate` tools also indicated that f-array based queue operations were the bottleneck and responsible for a large fraction of the execution time (of BH-GME). A more efficient implementation of a concurrent priority queue may help improve the performance of BH-GME.

For the non-uniform case, we conducted experiments using 80/20 and 70/30 session distributions as well. The gap between our GME algorithm and the other two GME algorithms narrowed by 10-15%, but the trend was still the same.

We also conducted experiments in which threads were bound to cores using `pthread_setaffinity_np()` function available in `sched.h` library. We observed that binding threads to cores had no significant impact on the performance and, thus, we have not included those results here.

CHAPTER 6

GME ALGORITHM FOR DISTRIBUTED SHARED MEMORY (DSM) MODEL

The GME algorithm for the CC model can be modified to work for the DSM model as well. The DSM model algorithm is similar to the algorithm for the CC model. However, we have to modify it so that it is asymptotically optimal in terms of the number of remote memory references that are generated in the DSM model (which is known to be $\Omega(n)$ in the worst case [17]).

6.1 Overview

We show how to modify our GME algorithm for the CC model to achieve the optimal RMR complexity of $O(n)$ for the DSM model, while maintaining all the other desirable properties.

We present the complete algorithm for the DSM model. However, Algorithm 21 shows the main changes/additions we made to the pseudocode in Algorithms 2-11 to adapt our GME algorithm for the DSM model.

The main idea is that, instead of busy waiting on session state until the session adjourns, a process busy waits on a variable in its local memory (but which is still accessible to other processes). We use $ready[i]$ to denote the local memory of process p_i (line 299). We refer to the node hosting the session that a process needs to wait on until it is adjourned as the *anchor node*. In order to busy wait, a process p_i writes the address of the anchor node in $ready[i]$; p_i then spins until some process clears that address from $ready[i]$ (lines 300-303). A process p_j notifies a spinning process p_i that the "relevant" session has adjourned (by clearing $ready[i]$) under the following conditions: (1) if p_j is the last process to leave the session provided it is also responsible for adjourning the session (line 311), (2) if p_j is trying to establish the next session and p_i is the leader of the newly established session (line 312), and (3) if p_j is the leader of the immediate next session (line 310). We also say that p_j *releases* p_i. We refer to the node hosting the session that was most recently

Algorithm 12: Data types and variables used.

```
// Node of a list
struct Node {
    integer session;                          // session associated with the node
    integer instance;                   // instance identifier of the GME object
    integer number;                             // the next process to be helped
    {bool,bool,bool,bool} state;                // four flags representing state
    integer size;               // number of processes currently in the session
    NodePtr prev, next;               // address of the previous and next nodes
    integer owner;                            // the last process to own the node
    bool condition;   // a boolean flag to indicate whether the node is safe
      or unsafe
};

shared variables
    head: array [1...m] of NodePtr;     // to store references to head nodes of
      lists
    announce: array [1...n] of NodePtr, initially [null,...,null];     // to announce CS
      requests
    hp: array [1...n][1...2] of NodePtr;                // to store hazard pointers

private variables
    snapshot: array [1...n] of NodePtr; // to store snapshots of the head nodes
    // snapshot[i] is a private variable of process p_i
    pool: array [1...n][1...2][1...3n] of NodePtr;       // to store pools of nodes
    which: array [1...n] of integer;    // to indicate which of the two pools is
      active
    marker array [1...n] of integer;    // pointer to the first safe node in the
      active pool
```

adjourned in the above three conditions as the *consumed node*. Finally, to ensure that only relevant processes are notified, a process p_j clears the local memory of a spinning process p_i using a CAS instruction, which succeeds only if the address of the anchor node in $ready[i]$ matches the address of the consumed node (line 306).

Note that a process invokes the RELEASEALL method at most two times during its passage. Moreover, a process invokes the RELEASE method (directly) only when a new session is established. Thus the RMR complexity of a passage increases by only $O(n)$.

Algorithm 13: Initialization.

```
initialization
begin
    // initialize shared variables
    foreach i ∈ [1...m] do
        head[i] := new Node;                                  // create a new node
        head[i] → state := LEADERLESS ;                       // session has no leader
        head[i] → size := 0;                                  // session has no processes
        head[i] → next := null;                               // node has no successor
        // all other fields can be initialized arbitrarily
    foreach i ∈ [1...n] do
        announce[i] := null ;                                 // process has no outstanding request
    // initialize shared variables
    foreach i ∈ [1...n], j ∈ [1...2] do  hp[i][j] := null;
    // initialize private variables
    foreach i ∈ [1...n] do
        foreach j ∈ [1...2], k ∈ [1...3n] do
            pool[i][j][k] := new Node;                        // create a new node
            pool[i][j][k] → owner := me;                      // set the owner as myself
        which[i] := 1;                                        // designate pool[i][1] as active
        marker[i] := 1;                                       // designate pool[i][1][1] as the first safe node
```

6.2 Proof And Complexity Analysis

We show that our GME algorithm for the DSM model satisfies the following two properties. First, every process that is busy waiting for a session to be adjourned stops spinning eventually once that happens. Second, if the system is in a homogeneous state, a process executes at most one iteration of its busy wait loop at line 303. With the two properties, assuming no memory reclamation, the correctness proof and complexity analysis of the GME algorithm for the CC model also carries over to the DSM model.

To prove the first property, it suffices to show that, once the current session has adjourned, some process eventually establishes a new session. This in turn guarantees that the leader of this new

Algorithm 14: Functions operating on session state.

```
// returns true if the session is closed and false otherwise
bool IsClosed(integer state)    { return (state & LEADERLESS) and (state & CONFLICT); }

// returns true if the session is adjourned and false otherwise
bool IsAdjourned(integer state)    { return (state & VACANT); }

// returns true if the node is retired and false otherwise
bool IsRetired(integer state)    { return (state & RETIRED); }

// sets a given guard flag (LEADERLESS or CONFLICT) in the session
   state
SetGuardFlag(NodePtr node, bool flag )
begin
    while true do
        integer state := node → state ;                    // read the current state
        if (state & flag) then return;                     // flag already set
        if CAS(node → state, state, state | flag) then return; // successfully set the
          flag

// sets the vacant flag in the state if possible
SetVacantFlag(NodePtr node)
begin
    integer state := node → state;                         // read the current state
    if not(IsClosed(state)) then return;                   // session is still open
    if (node → size ≠ 0) then return;                      // session still has participants
    CAS(node → state, state, state | VACANT);

// mark the node as retired
MarkAsRetired(NodePtr node)    { node → state := LEADERLESS | CONFLICT | VACANT
  | RETIRED; }
```

session eventually stops spinning (if applicable), that, in turn, releases all processes busy waiting on the previous session to be adjourned. We have,

Lemma 25. *Once the current session has adjourned, some process eventually establishes a new session.*

Algorithm 15: Functions operating on list head.

```
   // reads the current head pointer of the list
196 READHEAD(integer instance)
197 begin
198     repeat
199         snapshot[me] := head[instance];  // read the current head pointer of the
             list
200         hp[me][1] := snapshot[me];              // declare it as a hazard pointer
201     until (snapshot[me] = head[instance]);

   // returns true if the head of the list has not moved and false
      otherwise
202 bool TESTHEAD(integer instance)
203 begin
204     if (head[instance] ≠ snapshot[me]) then  return false;       // head has advanced
205     else  return true;

   // advances the head of a list to the given node if the head has not
      moved
206 ADVANCEHEAD(integer instance, NodePtr successor)
207 begin
208     CAS(head[instance], snapshot[me], successor);
```

Proof. Note that, if some process is busy waiting on a session to be adjourned, then the CONFLICT flag in the session state is guaranteed to be set before that process examines the session state (at line 301). If the last process to leave the session is not able to adjourn the session, then it implies that the session size was incremented spuriously. In that case, the session is guaranteed to be adjourned by the time the SETVACANTFLAG method—invoked by the last process to increment the session size spuriously—completes. Thus there is at least one process in the system that attempts to establish a new session. □

We now prove the second property. For the next lemma, we use the same notation as used in the proof of Lemma 14 and the following additional notation. Given a node U whose session has adjourned, let $f(U)$ denote the first process to join the session (*i.e.*, the leader of the session). Likewise, let $\ell(U)$ denote the last process to leave the session. We have,

Algorithm 16: Entry section.

```
// code for entry section
ENTER(integer myinstance, integer mysession)
begin
    // initialize a node and announce the request to other processes
    GETNEWNODE(myinstance, mysession);
    NodePtr mynode := announce[me];
    while true do
        READHEAD(myinstance);                    // read the head pointer of the list
        NodePtr current := snapshot[me];         // find the last node in the list
        if (announce[me] = current) then
            // join the session as a leader and retire the predecessor node
            RETIRENODE(mynode → prev);
            return;
        if (current → session = mysession) then   // my request is compatible with the current session
            if not(ISCLOSED(current → state)) then          // the session is open
                // attempt to join the session as a follower
                FAA(current → size, 1);                     // increment the session size
                if not(ISCLOSED(current → state)) then      // the session is still open
                    // join the session as a follower and retire own node
                    RETIRENODE(mynode);
                    return;
                else                                        // the session is no longer open
                    FAA(current → size, -1);    // abort the attempt and decrement the session size
                    SETVACANTFLAG(current);     // set VACANT flag if applicable
        else                          // my request conflicts with the current session
            SETGUARDFLAG(current, CONFLICT);                // set CONFLICT flag
            SETVACANTFLAG(current);                         // set VACANT flag if applicable
        while not(ISADJOURNED(current → state)) do ;        // do nothing
        ;                                                   // spin
        if TESTHEAD(myinstance) then APPENDNEXTNODE(myinstance); // establish a new session
```

Algorithm 17: Exit section.

```
// code for exit section
EXIT(integer myinstance)
begin
    READHEAD(myinstance);                          // find the head node of the list
    NodePtr current := snapshot[me];
    if (current → owner = me) then                 // joined the session as a leader
        SETGUARDFLAG(current, LEADERLESS);         // set the LEADERLESS flag
    FAA(current → size, -1);                       // decrement the session size
    SETVACANTFLAG(current);                        // set VACANT flag if applicable
```

Lemma 26. *Assume that the system is in a homogeneous state at the beginning of an iteration of the inner while-do loop at line 303. Then the process executes the body of the while-do loop at most once.*

Proof. Let p denote the process executing the loop mentioned in the lemma statement, t_0 the time at which p starts executing the current iteration of the loop, and ρ the outstanding request of process p at time t_0. Also, let H denote the head of the list at the time when p starts executing the iteration, and U the head of the list read by p using the READHEAD method most recently (before time t_0). Note that U is also the anchor node of p.

By assumption, the system is in a homogeneous state at time t_0, which, in turn, implies that there is no outstanding request at time t_0 that conflicts with ρ. There are two cases to consider:

Case 1 ($U = H$): In this case, we claim that $s(\rho) \neq s(H)$. Otherwise, if $s(\rho) = s(H)$, then it would imply that $s(H)$ is already closed at time t_0. Lemma 8 implies that there exists an outstanding request σ at time t_0 such that $s(\sigma) \neq s(H)$. This is turn implies that ρ and σ are both outstanding requests at time t_0 with $s(\rho) \neq s(\sigma)$—a contradiction. Thus, for the rest of this case, assume that $s(\rho) \neq s(H)$.

Since the system is in a homogeneous state at time t_0, we can infer that there are no spurious increments on the size field of H. Otherwise, it would imply that exists a pending request

Algorithm 18: Functions operating on a list node.

```
// get a new node, initialize it and announce it to other processes
GETNEWNODE(integer instance, integer session)
begin
    NodePtr node := pool[me][which[me]][marker[me]];  // get a safe node from the active pool
    node → condition := UNSAFE;                        // mark it as unsafe
    node → instance := instance;                       // initialize node's instance
    node → owner := me;                                // set the owner as myself
    node → session := session;                         // initialize node's session
    node → size := 1;                                  // initialize session size
    node → next := null ;                              // node has no successor
    node → prev := null ;                              // node has no predecessor
    node → state := 0;                                 // session is open with no condition flag set
    node → number := 0;                                // set the sequence number to a sentinel value
    announce[me] := node;                              // make the node visible to other processes

// get the next node to be appended to the list
NodePtr SELECTNEXTNODE(integer instance)
begin
    NodePtr mine := announce[me];                                          // my node
    NodePtr helpee := announce[snapshot[me] → number];                     // helpee's node
    hp[me][2] := helpee;                                                   // declare reference to the helpee's node as a hazard pointer
    if (announce[snapshot[me] → number] ≠ helpee) then return mine;        // request already fulfilled

    // ascertain that the helpee's node is usable
    if (helpee = null) then return mine;                                   // process has no outstanding request
    if (helpee → instance ≠ instance) then return mine;                    // request is for a different GME object
    if ISRETIRED(helpee) then return mine;                                 // node has been retired
    if not(TESTHEAD(instance)) then return mine;                           // head has moved
    return helpee;                                                         // helpee's node passed all the tests
```

σ at time t_0 such that $s(\sigma) = s(H) \neq s(\rho)$. In other words, ρ and σ are both outstanding requests at time t_0 with $s(\rho) \neq s(\sigma)$—a contradiction.

Now, let $q = \ell(H)$; q exists because there are no current or future requests for $s(H)$ at time t_0. Note that both p and q invoke the SETVACANTFLAG method on U—p after setting

Algorithm 19: More Functions operating on a list node.

```
// append a new node to the list
APPENDNEXTNODE(integer instance)
begin
    NodePtr current := snapshot[me];                    // get the last node in the list
    NodePtr successor := SELECTNEXTNODE(instance);  // choose a node to append
    CAS(current → next, null, successor);    // set the next field of the current
     last node
    NodePtr successor := current → next ;                    // read the next field
    hp[me][2] := successor;     // declare reference to the successor node as a
     hazard pointer
    if not(TESTHEAD(instance)) then  return;                    // head has moved
    successor → prev := current;     // set the previous field of the successor
    successor → number := (current → number + 1)  mod n + 1;  // set the sequence
     number used in helping
    ADVANCEHEAD(instance, successor);                    // advance the head

// retire the node
RETIRENODE(NodePtr node)
begin
    announce[me] := null;                              // help is no longer needed
    node → owner := me;                          // claim the ownership of the node
    pool[me][which[me]][marker[me]] := node;   // replace in case reclaiming the
     predecessor node
    marker[me] := marker[me] + 1;     // advance the pointer for the safe nodes
    MARKASRETIRED(node);                               // mark the node as retired
```

the CONFLICT flag in the state field of H and q after decrementing the size field of H. Let $r \in \{p,q\}$ denote the process that invoked the method *later*. Note that, when r invokes the SETVACANTFLAG method, the following must hold: (a) both LEADERLESS and CONFLICT flags are already set in the state field of H, and (b) the size field of of H is zero and stays zero thereafter. Clearly, when the SETVACANTFLAG method invoked by r returns, $s(H)$ is guaranteed to be adjourned.

It now remains to argue that *ready* entry of p is clear at time t_0. If $r = p$, then it follows trivially from the code. In this case, p itself clears the entry before it starts executing the

Algorithm 20: Cleanup algorithm.

```
// used to identify safe nodes in the passive pool; executing the
   method once corresponds to one epoch
CLEANUP( )
begin
    integer other := 3 − which[me];
    // mark all nodes in the passive pool as retired and safe
    foreach i ∈ [1...3n] do
        MARKASRETIRED(pool[me][other][i]);
        pool[me][other][i] → condition := SAFE;
    // scan all the hazard pointers
    foreach i ∈ [1...n], j ∈ [1...2] do
        NodePtr node := hp[i][j];
        if ISRETIRED(node) then                          // node is retired
            if (node → owner = me) then                  // I own the node
                node → condition := UNSAFE;              // mark the node as unsafe
    collect all safe nodes in the passive pool toward the end of the array using a method
      similar to the partition procedure used in quick sort, which has linear running time;
    // start a new epoch
    marker[me] := index of the first safe node in the passive pool;
    which[me] := 3 − which[me];          // switch the designations of the pools
```

loop. On the other hand, if $r = q$, then, by definition, the system cannot be in homogeneous state until q has finished executing its exit section. At the end of its exit section, q releases all processes with H as their anchor node by clearing their *ready* entries. This is because, as mentioned earlier, $s(H)$ is guaranteed to be adjourned when the SETVACANTFLAG method invoked by q returns.

Case 2 ($U \neq H$): Let H_p denote the predecessor node of H. There are two subcases to consider:

Case 2.1 ($U = H_p$): There are possible scenarios:

Case 2.1(a) ($s(\rho) = s(H)$): The proof for this case is similar to the proof for Case 1 with H replaced by H_p.

Algorithm 21: Changes for the DSM Model.

```
298 additional shared variables
299     ready: array [1...n] of NodePtr;  // used for spinning - ready[i] is local to
         process p_i

    // changes to ENTER method - replace lines 231-232 with lines 300-303
300 ready[me] := current;                  // the node hosting the current session
301 if ISADJOURNED(current → state) then
302     ready[me] := null;            // session already adjourned - no need to spin
303 while (ready[me] ≠ null) do       // spin until the entry contains null pointer
        ;                                                          // do nothing

    // notify a specific process to stop spinning
304 RELEASE(integer i, NodePtr node)
305 begin
306     CAS(ready[i], node, null);            // signal the process to stop spinning

    // notify all processes to stop spinning
307 RELEASEALL(NodePtr node)
308 begin
309     foreach i ∈ [1,n] do RELEASE(i, node);

    // changes to the ENTER method - insert line 310 just before line 217
310 RELEASEALL(mynode → prev);

    // changes to the EXIT method - insert line 311 just after line 241
311 if ISADJOURNED(current → state) then RELEASEALL(current);

    // changes to the APPENDNEXTNODE method - insert line 312 just after
       line 276
312 RELEASE(successor → owner, current);
```

Case 2.1(b) ($s(\rho) \neq s(H)$): Let $q = f(H)$. Also, let t_e denote the time when q completes its entry section. Clearly, $t_e < t_0$. By design, q releases all processes busy waiting with H_p as their anchor node before completing its entry section.

Case 2.2 ($U \neq H_p$): Let U_s denote the successor node of U. Note that, in this case, U_s is either H_p or its ancestor. In either case, the session hosted by U_s adjourns before time t_0. Let $q = f(U_s)$. Also, let t_e denote the time when q completes its entry section.

Clearly, $t_e < t_0$. By design, q releases all processes busy waiting with H_p as their anchor node before finishing its entry section.

In all cases, the *ready* entry of p is guaranteed to be clear at time t_0 (and thereafter), and thus p is guaranteed to quit the loop after completing the current iteration. □

Note that, before a process starts busy waiting by spinning on its entry in *ready*, it would have already declared the reference to the anchor node as a hazard pointer (and then examined its session state field). This ensures that the anchor node cannot be reused until the process stops spinning. Also, before a process releases a spinning process, it (the former) either would have declared the reference to the consumed node as a hazard pointer or would become its new owner. In either case, the consumed node cannot be reused until the RELEASE or RELEASEALL method has completed. Thus the modified GME algorithm also works in the presence of memory reclamation.

CHAPTER 7

LOCK-BASED CONCURRENT RED BLACK TREE

In the previous chapters, we described advanced concurrency techniques such as Group Mutual Exclusion which can be applied to concurrent data structures such as Skip Lists, Unrolled Linked Lists, etc. In this chapter, we focus our attention on a concurrent algorithm for a balanced binary search tree data structure, *viz.*, a concurrent red black tree. We first describe the background needed to understand our work - mainly, the top-down framework which lies at the core of the algorithm. We then present a *lock-based* algorithm for a concurrent, strictly-balanced red black tree data structure that supports search, insert, and delete operations followed by its proof of correctness and experimental evaluation.

7.1 Top-Down-Framework

7.1.1 Tsay and Li's Framework

Tsay and Li described a general framework for deriving a wait-free algorithm for a tree-based data structure from its sequential version provided all operations on the tree work in a top-down manner, modifying the tree as they move down. The framework is based on the concept of *window*, which is basically a small *rooted* subtree of the tree. When an operation starts, its window is rooted at the root of the tree. As the operation proceeds, it modifies the portion of the tree within its window, and then the window slides down. To modify the portion of the tree within its window, the operation copies the nodes in its window to its local memory, modifies the local copy as appropriate and then replace the portion of the tree within its window with the one its local memory. We refer to the actions performed by an operation to move its window down once as a *transaction*. Also, as mentioned before, we refer to the framework as the *TL-framework*. More details about the framework are available in [64].

The TL-framework has some limitations that makes it impractical or inefficient to use: (a) It assumes the existence of check_valid instruction that is not currently implemented in hardware. (b) A single word is required to store two distinct addresses. (c) It uses a dual node structure for a tree node: pointer node and data node. The pointer node contains the address of the data node. The data node in turn contains the addresses of the pointer node of left and right children. As a result, to visit a tree node, a process has to dereference *two* addresses. (d) The window of every operation is initially rooted at the root of the tree. As a result, the root of the tree becomes a single point of contention. (However, operations can execute concurrently on the tree once their paths diverge.)

In [56], on a wait-free algorithm for a concurrent red-black tree, Natarajan *et al.* modified the TL-framework to remove the first two limitations. In this work on a lock-based algorithm for a concurrent red-black tree, we make further modifications to the framework to remove the last two limitations to give better performance in practice. Specifically, (i) A tree node now consists of a single physical node and not two physical nodes (a pointer node and a data node) as in [64, 56]), thereby making the traversal of the tree faster. (ii) An operation can now start performing window transactions from the middle of the tree and not necessarily from the root of the tree again as in [64, 56], thereby reducing contention among operations that work on different regions of the tree. Both modifications are non-trivial and require careful synchronization among modify operations to ensure correct behavior.

7.1.2 Tarjan's Sequential Top-Down Algorithm for Red-Black Tree

Traditional algorithm for maintaining a red-black tree involves a top-down phase (to add or remove the key) followed by a bottom-up phase (to rebalance the tree). Tarjan presented an algorithm for maintaining an *external* red-black tree that involves traversing the tree *once* in a top-down manner [63].

In Tarjan's algorithm, an operation works on a *constant size* window of the tree at a time such that some invariant property holds at the *root of the window*. To avoid confusion, we refer to the

root node of a window as the *anchor node* of the window. For an insert operation, the invariant property is that the anchor node of the window is black and has a black child. For a delete operation, the invariant property is that either the anchor node is red or has a red child or has a red grandchild. Note that an anchor node of an operation's window is always a node on the access-path induced by the operation's key. Also, note that, if the invariant property does not hold at the root of the tree (which is the starting point of every operation), then a simple recoloring of the root node of the tree can be used to guarantee that the invariant property holds when the operation starts. To slide its window down, an operation applies one or more transformations (*e.g.*, recoloring of nodes, left or right rotation) to the nodes in the window to ensure that the invariant property now holds at some *descendant node* of the anchor node along the access path. This allows the window to move down to this descendant node, which now becomes the new anchor node of the operation's window. More details about the algorithm are available in [63].

Note that Tarjan's top-down algorithm, which is window-based, is a natural fit for Tsay and Li's framework, which is also window-based. It can be verified that, in Tarjan's algorithm, when a window transaction is applied to a red-black tree at a node that satisfies the operation-specific invariant property, the resultant tree is also a valid red-black tree. This leads us to observe that it is not necessary to start executing the Tarjan's algorithm from the root node of the tree. In fact, we can start executing the algorithm from *any* internal node of the tree that satisfies the operation-specific invariant property.

Consider a red-black tree T. Given a node X in T that satisfies an operation-specific invariant property, we use $i\text{-}path(X)$ to denote the path in T, starting from X, that consists of all the nodes over which the invariant property is evaluated. Note that $i\text{-}path(X)$ contains at most three nodes. We use $p\text{-}path(X)$ to denote the path consisting of the parent nodes of the nodes in $i\text{-}path(X)$. Finally, we use $c\text{-}path(X)$ to denote the union of the nodes in $i\text{-}path(X)$ and $p\text{-}path(X)$. Note that $c\text{-}path(X)$ contains at most four nodes. Further, $p\text{-}path(X)$ and $c\text{-}path(X)$ have the same head node, and $i\text{-}path(X)$ and $c\text{-}path(X)$ have the same tail node. We refer to the head node of

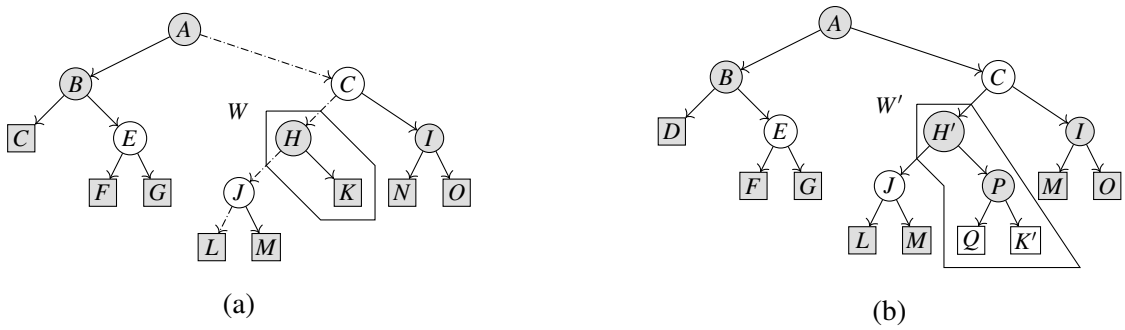


Figure 7.1: An illustration of a red-black tree. Shaded nodes represent black nodes and unshaded nodes represent red nodes.

i-path(X), which is same as node X, as an anchor node (because it can act as the root node of a window). We refer to the head node of *p-path*(X) as a *guardian* node (because replacing a window will involve changing a child pointer at a guardian node). Basically, guardian node is the parent node of anchor node. Also, note that we do not consider guardian node to be part of the window. The window technically starts from the anchor node.

As an illustration, consider the red-black tree shown in Figure 7.1. Consider an operation with the access-path consisting of nodes A, C, H, J and L. If the operation is an insert operation, then nodes A and H satisfy the invariant property. On the other hand, if the operation is a delete operation, then nodes A, C, H and J satisfy the invariant property. Assume that the operation is an insert operation. Now, consider node H, which satisfies the invariant property. Then, *i-path*$(H) = \{H,K\}$, *p-path*$(H) = \{C,H\}$ and *c-path*$(H) = \{C,H,K\}$. For a window rooted at H, H is its anchor node and C its guardian node.

7.1.3 Optimizations on the Top-Down Framework

We further modify the framework to minimize copying of nodes, decrease interaction with the dynamic memory management system, and include a fast-path optimization to boost the performance of our algorithm. To minimize copying of nodes, we analysed certain cases of Tarjan's algorithm wherein we would not need to copy nodes since those nodes do not undergo any structural changes.

To minimize calls to *new*, we are able to reuse external nodes that will be discarded as part of an insert or delete operation. The fast-path optimization allows us to do a quick traversal down the tree to find the leaf node, its parent, and its grandparent, and if they satisfy certain conditions, then we are able to avoid doing a more expensive lookup (wherein we have to check for the invariant property as we traverse the tree down to the leaf node) to find the starting point of the operation i.e. the closest internal node that satisfies the invariant property.

7.2 A Lock-Based Algorithm

7.2.1 Overview of the Algorithm

Search Operation: A search operation simply traverses the tree from the root node to a leaf node along the access-path induces by the target key. On reaching a leaf node, the operation returns true if the target key matches the key stored at the leaf node; otherwise, it returns false.

Modify Operation: A modify operation consists of multiple phases as explained below:

1) **LightSeek:** This is the fast-path optimization that is crucial to the design of the algorithm. A modify operation proceeds by first finding the injection point in the tree by doing a traversal starting at the root node. We have identified cases in Tarjan's algorithm wherein it would suffice to do a quick traversal down to the leaf node (without checking the invariant at each step) and if the leaf node or its parent satisfy certain conditions, then we can proceed to the Execution phase directly. In this phase, the operation traverses the tree from the root node to a leaf node along the access path induced by the operation's key. At the end of this phase, the leaf node, its parent, and grandparent are stored in the operation's CandidateRecord. This record's elements are inspected to check for satisfaction of the invariant property. For an insert operation, if the leaf node is a black leaf node, or for a delete operation, if the leaf's parent satisfies the delete invariant, then the HeavySeek phase can be skipped and

the operation can proceed directly to the Execution phase. We call these cases the *Fast-Insert* case and the *Fast-Delete* case respectively. Note that the cases are *fast* because no rebalancing takes place during the operation. Nodes may only undergo color changes. The rest of the insert & delete operations are called *Slow-Insert* & *Slow-Delete*, respectively.

2) **HeavySeek:** In this phase, the operation traverses the tree from the root node to a leaf node along the access path induced by the operation's key. On the way, it keeps track of the *deepest* node in the access-path (*i.e.*, closest to the leaf node) at which the operation-specific invariant property holds. Let that node be denoted by X. A heavyseek phase, on termination, returns the address of the leaf node in the access-path along with information about all the nodes in $c\text{-}path(X)$. Each operation is associated with a data record that contains various items of information about the operation (*e.g.*, its type, its target key, its injection point, and so on). It is created before the injection phase.

3) **Injection:** In this phase, the operation attempts to inject itself into the tree by obtaining ownership of all the nodes in $p\text{-}path(X)$ in a top-down manner by acquiring locks on those nodes. If the CAS instruction in the TTAS lock fails, then it implies that there is another conflicting operation in progress concurrently. The currently executing operation needs to wait for the conflicting operation to finish before it can proceed further. Locking nodes in $p\text{-}path(X)$ ensures that that the operation-specific invariant property continues to hold when the first window transaction is executed. After Injection, we again check to make sure that the locked nodes have not changed just before they were locked.

4) **Execution:** In this phase, the operation performs a sequence of window transactions similar to those in the TL-framework until the window reaches a leaf node, at which point the operation completes. More details about the execution phase are presented later.

We analyzed Tarjan's algorithm and classified nodes into 3 categories to help decide which nodes will undergo structural & color changes that may affect concurrent operations. This helped

us reduce the number of nodes that needed to be copied per modify operation instead of copying all nodes in the window. The types are as follows:

1) *Type-1* Nodes: These are access path nodes and have to be locked, marked, and copied during the execution phase because their pointers may change as part of the rebalancing process. Only during the *Fast-Delete* case, we do not copy the root node because it will be deleted from the tree alongwith the leaf node of the delete operation.
2) *Type-2* Nodes: These are children of access path nodes. Their color may change during the rebalancing process. For a delete operation, they need to be locked, marked, and copied to ensure the window does not change due to a concurrent operation. The parent of a *Type-2* node is always a *Type-1* node. Thus, for an insert operation, these need not be locked, marked, or copied.
3) *Type-3* Nodes: These are nodes whose *color* field is examined to decide which rebalancing operation needs to be applied. No changes occur to these types of nodes. The parent of a *Type-3* node is always a *Type-1* or *Type-2* node.

As explained earlier, in the execution phase, an operation performs repeated window transactions until the window reaches a leaf node, at which point the execution phase terminates and the operation completes. The execution of a single window transaction by an operation consists of the following steps (assume that the window is anchored at node x):

4a) **Expand-Lock-Mark-And-Copy:** In this step, the operation expands its window in a depth-first manner based on the Tarjan's algorithm, which, at the beginning, consists of only nodes in $p\text{-}path(X)$. Upon visiting a node, the operation locks the node (if it is not already locked during injection), marks it by setting the LSB bit of the *scm* field, and then makes a copy of the node in its local memory. After this step, all *Type-1* nodes in the operation's window are guaranteed to be locked & marked. This locking ensures that the window cannot change any more because no operation can now be injected at any node in the window. The marking

ensures that other concurrent operations do not try to inject onto a marked node since it will be removed from the tree. If the *Fast-Delete* case applies, then the root node of the window is not copied or marked. For an insert operation, the leaf node is replaced with 3 new nodes - an internal node and 2 external nodes. The leaf node is added to the reuse stack so that it can be reused as one of the 2 external nodes described above for future operations. For a delete operation, the leaf node that is to be deleted is added to the reuse stack as well.

4b) **Transform-And-Acquire:** In this step, the operation applies Tarjan's transformations to its local window. Let the next window of the operation be anchored at node Y. We use a slightly larger window than the one used in Tarjan's algorithm to guarantee that all the nodes in $p\text{-}path(Y)$ are part of the local window. The operation then locks the nodes in $p\text{-}path(Y)$. As explained before, this ensures that that the operation-specific invariant property continues to hold when the next window transaction is executed.

4c) **Replace:** In this step, the operation replaces the window in the tree with its local copy. This step involves updating a child pointer at the guardian node to point to the root node of the local window. Note that this step slides the window down from its current anchor node to its next anchor node.

4d) **Release:** In this step, the operation releases locks on all nodes that it had locked as part of the window transaction. Thus, all marked nodes in the original window are removed from the tree. The external node that was removed in the delete operation is added to a stack of nodes that are reused when allocating new external nodes during an insert operation.

As an illustration, consider the red-black tree shown in Figure 7.1a. Consider the window W consisting of nodes $\{H,K\}$; H is the anchor node of W. Figure 7.1b shows the same red-black tree in which the window W has been replaced with the window W'; W' consists of nodes $\{H',K',P,Q\}$. Note that nodes H' & K' are copies of nodes H & K, respectively. To replace W with W', the pointer at C, the guardian node of W, is switched from H to H'.

7.2.2 Details of the Algorithm

For convenience, we assume that the tree contains two sentinel keys, denoted by ∞_1 and ∞_2, such that both ∞_1 and ∞_2 are larger than any other key and $\infty_1 < \infty_2$. These keys are never removed from the tree. The *topmost* node in the tree is a special sentinel node, denoted by $\mathbb{R}$, which is an internal/routing node and contains the key ∞_1. The sentinel node $\mathbb{R}$ is never removed from the tree, and the actual red-black tree is given by the left subtree of $\mathbb{R}$. The right child of $\mathbb{R}$ is the node with the key ∞_2.

Algorithm 1: Algorithm for Quick Traversal of Tree (without checking Invariant)

```
QUICKTRAVERSE( type, key, candidateRecord )
begin
    // initialize the variables used in the traversal
1   node := ℝ→left;
2   pNode := ℝ;
3   gpNode := null;
4   while true do
        // check if the current node is a leaf node
5       if node→left = null then break;
        // keep track of parent & grandparent of
        // current node to store in candidateRecord
6       gpNode := pNode;
7       pNode := node;
        // visit the next node in the access-path
8       node := key ≤ node→key ? node→left : node→right;
    // read the node's timeStamp field, its key, and its timeStamp again
    // If timeStamps do not match, then this node will be reused soon
9   initialize the entries of the candidate record with node, pNode, & gpNode accordingly;
```

Data Structures Used

We use six different types of objects in our algorithm: *tree node*, *segment record*, *seek record*, *candidate record*, *reuse stack* and *data record*.

Algorithm 2: Algorithm to Search for key in tree

```
Boolean SEARCH( key )
begin
    QUICKTRAVERSE( SEARCH, key, candidateRecord );
    return (candidateRecord → leaf) → key = key;
```

Algorithm 3: Algorithm for Slow traversal of tree (with checking of invariant)

```
SLOWTRAVERSE( type, key, seekRecord )
begin
    // initialize the variables used in the traversal
    node := ℝ → left;
    candidate := node;
    while true do
        // check if the current node is a leaf node
        if node → left = null then break;
        if node satisfies the operation-specific invariant property then candidate := node ;
        // visit the next node in the access-path
        node := key ≤ node → key ? node → left : node → right;
    // initialize the entries of the seek record
    seekRecord → leaf := node;
    initialize seekRecord → segmentRecord with nodes in c-path(candidate);
```

Algorithm 4: Algorithm to abort operation

```
ABORT( segmentRecord )
begin
    // release the ownership of all the nodes owned in a bottom-up
       manner
    index := segmentRecord.length − 2;
    while index ≥ 0 do
        node := segmentRecord[index] → address;
        // release the ownership of the node
        node.unlock();
        index := index − 1;
```

Algorithm 5: Algorithm to Check presence of key based on operation type

```
Boolean CHECKTYPEMATCH( type, match )
begin
    if (type = INSERT) and match then
        // key already present in the tree
        return false;
    if (type = DELETE) and not(match) then
        // key not present in the tree
        return false;
```

Algorithm 6: Procedure to check if Fast case applies

```
Boolean CHECKFASTCASE( type, match )
begin
    if (type = INSERT) and (candidateRecord → leaf) → color = BLACK then
        // Fast-Insert case applies
        initialize segmentRecord with nodes in c-path(candidateRecord → leaf);
        skipSeek := true;
    else if (type = DELETE) and checkDeleteInvariant(candidateRecord → pLeaf) =
      true then
        // Fast-Delete case applies
        initialize segmentRecord with nodes in c-path(candidateRecord → leaf);
        skipSeek := true;
```

A tree node consists of the following fields: (a) *key*: the key stored at the node, (b) *color*: the color (red or black) of the node, (c) *left* and *right*: reference to the left and right child node, respectively, (d) *parent*: reference to a node's parent, only used during rebalancing, (e) *scm*: An AtomicInteger used to implement the TTAS lock. The LSB bit of this field, if set, indicates that the node is marked for removal. In the algorithm, we will refer to the mark field as *scm.mark*, (f) *timeStamp*: an integer used to indicate to concurrent operations traversing the same window that this node will be reused and so, the concurrent operation must not save this node as part of its seek record. This is checked as part of the fast-path optimization.

Algorithm 7: Core Algorithm for Update operations

```
Boolean MODIFY( type, key )
begin
35  while true do
        // Phase 1:  LightSeek Phase
36      QUICKTRAVERSE( type, key, candidateRecord );
        // find if the target key matches the stored key
37      match := (candidateRecord → leaf) → key = key;
38      if !CHECKTYPEMATCH( type, match ) then
39          return false;
        // Check for fast cases
40      CHECKFASTCASE( type, candidateRecord, skipSeek );
41      if !skipSeek then
            // Phase 2:  HeavySeek Phase
42          SLOWTRAVERSE( type, key, seekRecord );
            // find if the target key matches the stored key
43          match := (seekRecord → leaf) → key = key;
44          if !CHECKTYPEMATCH( type, match ) then
45              return false;
46      dataRecord := create a new data record and initialize it;
        // Phase 3:  Injection Phase
        // attempt to inject the operation into the tree
47      INJECT( segmentRecord, dataRecord );
48      if dataRecord → status = INJECTED then
            // Phase 4:  Execution Phase
            // find the current location of the window
49          node := dataRecord → location;
50          while node ≠ null do
                // execute one window transaction
51              EXECUTEONE( node, state );
                // find the new location of the window
52              node := dataRecord → location;
53          return dataRecord → outcome;
```

Algorithm 8: Algorithm to Inject Insert/Delete operation in tree

```
INJECT( segmentRecord, dataRecord )
begin
    // try to acquire the ownership of all the nodes in the p-path
54  index := 0;
55  while dataRecord → status = TRYING do
        // find the node whose ownership should be acquired next
56      node := segmentRecord[index] → address;
        // find the next node in the c-path
57      next := segmentRecord[index + 1] → address;
        // find the relevant child field of the node; used to verify that
           the link from node to next still exists
58      which := segmentRecord[index] → which;
59      child := which = 0 ? node → left : node → right;
        // try to acquire the ownership of the node
60      if (child.address ≠ next) or state.mark then
            // the link from node to next no longer exists or node has been
               marked for removal
61          dataRecord → status := ABORTED;    break;
        // try to own the node
62      node.lock()
63      if index < segmentRecord.length − 2 then
            // advance to the next node
64          index := index + 1;
65      else dataRecord → status := INJECTED ;
66  if dataRecord → status = ABORTED then
        // release ownership of all the owned nodes
67      ABORT( segmentRecord );
```

Algorithm 9: Algorithm to Unlock nodes in tree

```
UNLOCKNODES( guardian, dataRecord )
begin
68    if dataRecord→type = INSERT then
69        unlock nodes along access path starting with the guardian node;
70    else
71        guardian.unlock();
72        if Fast-Delete case applies then
73            unlock root node & sibling of leaf node if it was locked;
74        else
75            unlock nodes in window starting with wRoot;
```

Algorithm 10: Algorithm to execute one iteration of the execution phase

```
EXECUTEONE( guardian, targetKey, dataRecord )
begin
      // read the contents of the window root
76    wRoot := guardian→key ≤ targetKey ? guardian→left : guardian→right;
77    guardianChild := guardian→key ≤ targetKey ? 0 : 1;
      // Step 4a:  Expand-And-Copy
78    EXPANDANDCOPY( wNode, dataRecord );
      // Step 4b:  Transform-And-Acquire
79    TRANSFORMANDACQUIRE( wNode, current, dataRecord );
      // Step 4c:  Replace
      // install the new window
80    if guardianChild = 0 then
81        guardian→left := current;
82    else
83        guardian→right := current;
      // unlock all locked nodes in the window
84    UNLOCKNODES( node, dataRecord );
```

Algorithm 11: Algorithm to expand, lock, mark, and copy

```
EXPANDANDCOPY( wNode, dataRecord )
begin
    wNode := wRoot;
    while true do
        // lock & mark the node
        wNode.lock();
        wNode→scm.mark := 1;
        add a copy of wNode to local memory;
        // Unless (dataRecord→type = DELETE) and (Fast-Delete case
           applies)
        if should expand the window further then
            wNode := next node to visit;
        else break ;
```

Algorithm 12: Algorithm to transform local window as per Tarjan's algorithm

```
TRANSFORMANDACQUIRE( wNode, current, dataRecord )
begin
    apply transformations to the local window as per Tarjan's algorithm;
    if dataRecord→type = DELETE then
        reuseStack.push(wNode);
        wNode→timeStamp++;
    current := address of the root node of the local window;
    if not(last transaction) then
        next := address of the anchor node of the next window;
        parent := address of the parent node of next;
        foreach X ∈ p-path(next) do
            acquire ownership of X with flags in state field set appropriately;
    else
        parent := null;
        dataRecord→outcome := return value of the operation;
    dataRecord→location := parent;
```

A segment record contains information about nodes in a *c-path*. It is an array consisting of four entries, where each entry is a 3-tuple; the first value of the tuple specifies the address of a node in *c-path*, its second value specifies the color of the node at the time of populating the seek record, and its third value indicates which child of the node (left or right) is the next node in *c-path* (if it exists).

A seek record consists of the following fields: (a) *leaf*: the address of the leaf node at which the traversal of the tree ended, and (b) *segmentrecord*: information about all the nodes in a *c-path*.

A candidate record consists of 3 fields: (a) *leaf*: the address of the leaf node at which the traversal of the tree ended, (b) *pLeaf*: parent of the leaf node, (c) *gpLeaf*: grandparent of the leaf node,

A reuse stack is a per thread stack that stores the leaf node with the target key that is deleted during each delete operation. Once nodes are stored on this stack, each thread will try to reuse nodes from this stack instead of making a call to *new* to allocate an external node during an insert operation.

A data record consists of the following fields: (a) *type*: the type of the operation (search, insert or delete), (b) *key*: the key associated with the operation, (c) *location*: the address of the guardian node of the operation's current window, and (d) *status*: indicates whether the operation has been injected into the tree; it has three possible values: TRYING, INJECTED and ABORTED, and (e) *outcome*: the return value of the operation.

Tree Operations

A formal description of our algorithm is given In the pseudocode, we use '$\rightarrow$' to refer to a field of an object (*e.g.*, $node \rightarrow key$, $node \rightarrow color$) and '.' to refer to a subfield of a word/field (*e.g.*, *scm.mark*).

As explained earlier, a search operation simply traverses the tree from the root node to a leaf node along the access-path induced by the target key. So we mainly focus on a modify operation. As explained earlier, the execution of a modify operation (lines 35-53) consists of multiple phases.

LightSeek Phase: In the lightseek phase (lines 35-34), the operation first invokes QUICKTRAVERSE function (line 39). In QUICKTRAVERSE function (lines 1-9), it traverses the tree from the root node to leaf node along the access-path induced by the target key (lines 4-8). During the traversal, it keeps track of the parent & grandparent of *last* node it visited in the access-path. At the end of the lightseek phase, the operation checks for the *Fast-Insert* or *Fast-Delete* case, and either terminates or advances to the injection phase (lines 37-39) (skipping the heavyseek phase). In the latter case, it creates a new data record and initializes all its fields (line 46).

HeavySeek Phase: In the heavyseek phase (lines 42-46), the operation first invokes SLOWTRAVERSE function (line 42). In SLOWTRAVERSE function (lines 12-19), it travels the tree from the root node to a leaf node along the access-path induced by the target key (lines 13-17). During the traversal, it keeps track of the *last* node it visited in the access-path for which the operation-specific invariant property evaluated to true (line 16). It also keeps track of the set of nodes that are involved in satisfying the invariant property (not shown in the pseudocode). At the end of the heavyseek phase, the operation compares its target key with the key stored in the leaf node of the access-path and, depending on its type, either terminates (lines 43-45) or advances to the injection phase. In the latter case, it creates a new data record and initializes all its fields (line 46).

Injection Phase: In the injection phase (line 47), the operation invokes INJECT function (line 47). In INJECT function (lines 54-67), the operation tries to acquire the ownership of all the nodes in the *p-path* of the anchor node found in the heavyseek phase in a top-down manner (lines 61-62). Prior to obtaining ownership of the node, it verifies that the next link in the path still exists and the node has not been marked for removal by another operation (line 60). If the verification fails, then it aborts the injection phase (line 61); otherwise it continues further. Finally, it tries to obtain the ownership of the node by locking it (lines 61-62). If the operation successfully locks the node, then it moves to the next node in the *p-path* if one exists (line 64) and stop otherwise (line 65).

Execution Phase: In the execution phase (lines 49-52), the operation executes window transactions by repeatedly invoking EXECUTEONE function (line 51) until it completes. In EXECUTEONE function (lines 76-84), the operation first expands the window (lines 85-92) starting from its anchor node (line 85). To that end, it first marks the node by setting the LSB bit of the node's *scm* field (line 88). Lastly, it makes a local copy of the node (line 89). Next, the operation applies appropriate transformations to its local copy of the window (line 93) and acquires the ownership of all the nodes in the next *p-path* (lines 100-102). Then, it replaces the window in the tree with its local copy (lines 79-83). It updates the data record with the new location of the window (line 105). Finally, it invokes UNLOCKNODES function (line 84) to unlock all nodes that were locked in the operation's window.

7.3 Correctness Proofs

In this section, we prove the correctness of our algorithm. Specifically, we show that our algorithm is linearizable (the outcome is equivalent to that of some sequential execution of operations) [34] and deadlock-free.

If a node is reachable from the topmost sentinel node of the tree, then we say that it is an *active* node; otherwise we say that it is a *passive* node. Also, we refer to the anchor node returned by the heavyseek phase of a modify operation as the *injection point* of the operation.

7.3.1 The Main Idea

Linearizability: This proof consists of two parts. We describe them one-by-one.

First, we prove that modify operations are injected correctly.

Note that our algorithm tries to maintain a red-black tree on the set of active nodes. We refer to this tree as the *global tree*. When a modify operation starts its heavyseek phase, it starts by traversing this global tree. As it is traverses the tree, the global tree may undergo structural changes. Depending on the current position of the operation, it may only see a *subset* of these structural

changes. As a result, the tree as seen by an operation traversing nodes may be *different* from the global tree. We refer to the tree as seen by a modify operation during its heavyseek phase as its *local tree*. As window transactions are performed, the global tree and the local tree of a modify operation may "evolve" in different manner. Specifically, the local tree only sees a subset of the window transactions seen by the global tree. We prove, however, the two trees stay "consistent" with each other throughout this "evolution" period (which lasts until the modify operation dereferences a leaf node). This property, in turn, allows us to deduce the following. First, if the injection point of a modify operation is correct with respect to the global tree and the injection point is visible to the local tree, then it is also correct with respect to the local tree. Second, if the injection point of a modify operation is correct with respect to the local tree and the injection point is an active node, then it is also correct with respect to the global tree. Using these properties, we prove the following. First, the global tree is always a valid red-black tree. Second, a modify operation in the heavyseek phase traverses a valid red-black tree. This result can be easily extended to show that a search operation traverses a valid red-black tree as well.

Second, we prove that operations yield correct results. We use *linearizability* as the correctness condition for executions of our algorithm [34]. Intuitively, a sequence of operations is linearizable if each of the operations appears to take effect at a single moment between the time that the application invoked the operation and the time that the application received the response, referred to as the *linearization point*. Further, the result of all operations is the same as it would be if the operations were performed sequentially on the data structure in the order of their linearization points. In our proof, we define the linearization point of a "completed" operation as follows. Consider an operation α. If α is a modify operation that completes in execution phase, then the linearization point of α is taken to be the time when α performed its *terminal* window transaction. If α is a search operation or a modify operation that completes in heavyseek phase, the linearization point of α is taken to be the time when the *last* terminal window transaction that is *visible* to α is performed by some modify operation working on the same key as α. If no such modify operation

exists, then the linearization point of α is taken to be the time when α began its traversal. We use these linearization points to establish that every execution of our algorithm is correct.

Deadlock-Freedom: A common technique for ensuring that a set of concurrent operations is deadlock-free is to impose a total order on locks. If all operations acquire locks in the same order, then we are guaranteed that the concurrent operations are deadlock-free. Our search operations are lock-free. However, our modify operations require locks. Our algorithm assumes a tree-ordering on locks *i.e.* it uses the property of the binary search tree to define an order on lock acquisition. A thread that holds no locks may acquire a lock on any node. A thread that has already locked a node may lock only one of the children of the node it previously locked at a time. Once a node is locked, it may undergo structural as well as color changes. Each modify operation, after identifying its injection point, acquires a lock on the operation's guardian node followed by the anchor node (which is a child of the guardian node). A delete operation may require a lock on one of the children of the anchor node as well. Thereafter, the modify operation can acquire locks on only descendants of the initial set of nodes locked during injection. Consider a thread T_i that holds a lock on a node. Let b_i be the node most recently locked by T_i and let z_i be the node least recently locked by T_i. Either b_i & z_i are the same node or b_i is a descendant of z_i. Let T_i & T_j try to acquire a lock on a_i which is a child of b_i. If T_i succeeds, T_j will wait until T_i releases all locks in a top-down manner and vice-versa. Since there are no cycles possible in the tree, the locks cannot be acquired in a cyclic manner. Thus, in spite of concurrent changes to the tree structure, our protocol is deadlock-free.

7.3.2 Executions are Linearizable

In our algorithm, nodes in the tree are replaced by their copies as operations perform window transactions. We treat copies of the same node in the tree as different nodes. So, when we speak of a node, we are referring to a specific copy of a node. We assume that the pointer node of the

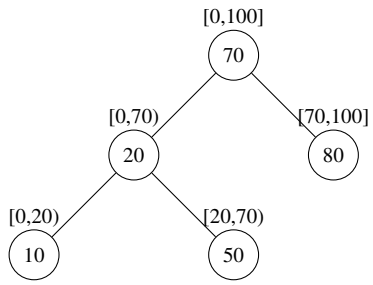


Figure 7.2: An illustration of the coverage sets of various nodes in a binary search tree assuming that the range of keys is [0,100].

root of the tree is a special node that is never replaced or removed from the tree. This can be easily ensured by assuming that the tree contains a special key initially that is never removed from the tree, and is larger than any other key. For ease of exposition, we also assume that initially the tree contains no other key besides the special key.

Modify Operations are Injected Correctly

We first provide a few definitions that we use in our proof. Given a tree T, we use $nodes(T)$ to denote the set of nodes in T.

Definition 1 (coverage-set of a node)**.** *Given a tree T and a node $X \in nodes(T)$, the coverage-set of X in T, denoted by coverage-set(X,T) is defined as the set of keys k such that the search (or access) path for k in T contains X.*

For an illustration, please refer to Figure 7.2. Note that the coverage-set of a node in a tree is a function of the set of keys stored at its *anscestor* nodes in the tree.

Definition 2 (extant operation)**.** *Consider a tree T and a modify operation α. We say that α is* extant *in T, denoted by extant(α,T), if α has been successfully injected into the tree (the status field of its most recent data record has the value INJECTED).*

Definition 3 (anchor node of an operation)**.** *Consider a tree T and a modify operation α such that α is extant in T. The* anchor node *of α in T, denoted by anchor(α,T), is the node that is currently owned by α, is unmarked and satisfies the operation-specific invariant property.*

Note that, in the trees we consider in this proof, the anchor node of an operation is uniquely defined. The next definition describes what it means for an extant operation to be "correctly positioned" in a tree. Given a modify operation α, let $key(\alpha)$ denotes its key.

Given a tree T, a node X in T and a modify operation α, we use $invariant(\alpha, X, T)$ to denote the fact that the invariant property required by the Tarjan's algorithm for α holds at X in T.

Definition 4 (admissible operation)**.** *Consider a tree T and a modify operation α such that α is extant in T. We say that α is* admissible *with respect to T, denoted by $admissible(\alpha, T)$, if (i) the invariant required by the Tarjan's algorithm for α holds at α's anchor node in T, and (ii) α's key lies within the coverage-set of α's anchor node in T. Formally,*

$$admissible(\alpha, T) \quad \triangleq \quad extant(\alpha, T) \wedge invariant(\alpha, A, T) \wedge (key(\alpha) \in coverage\text{-}set(A, T))$$

where $A = anchor(\alpha, T)$.

Given a tree T and a modify operation α that is admissible with respect to T, we use $W(\alpha, T)$ to denote the set of nodes in T that lie in the window of α rooted at $anchor(\alpha, T)$ as specified by the Tarjan's algorithm.

Definition 5 (enabled operation)**.** *Consider a tree T and a modify operation α such that α is admissible with respect to T. We say that α is* enabled *in T, denoted by $enabled(\alpha, T)$, if the window of α in T does not contain any guardian node owned by some other operation. Formally,*

$$\begin{aligned} enabled(\alpha, T) \quad \triangleq \quad & admissible(\alpha, T) \wedge \\ & \langle \forall X : X \in W(\alpha, T) : X \text{ is not owned by any other} \\ & \text{operation as its guardian node} \rangle \end{aligned}$$

We now define the concept of a tree which is basically is a red-black tree with a (possibly empty) set of "partially executed" modify operations.

Definition 6 (legal tree)**.** *A tree T is said to be* legal*, denoted by $legal(T)$, if it satisfies the following properties:*

1. *T is a valid red-black tree.*

2. *Every extant operation in T is admissible with respect to T.*

We use the next concept to relate two different trees that may have one or more nodes in common.

Definition 7 (consistent trees)**.** *We say that two trees S and T are* consistent *with each other, denoted by $S \overset{c}{\approx} T$, if every node that is common to both S and T has identical coverage-sets in both the trees. Formally,*

$$S \overset{c}{\approx} T \quad \triangleq \quad \langle \forall X : X \in nodes(S) \cap nodes(T) : coverage\text{-}set(X,S) = coverage\text{-}set(X,T) \rangle$$

The following axiom essentially captures the functioning of the Tarjan's algorithm modified slightly as follows: the terminal window of a delete operation is extended to include the children of the sibling of the leaf node being removed, if they exist. Note that, in an external red-black tree, the sibling of a leaf node is either a leaf node or the parent of two leaf nodes. As a result, this modification increases the size of a window by at most two nodes only.

Axiom 1. *A window transaction applied to a legal tree yields a legal tree. Moreover, it does not change the coverage-set of nodes outside the window. Formally, consider a legal tree T and a modify operation α such that α is enabled in T. Let S denote the tree obtained by executing one window transaction of α in T. We have:*

$$legal(T) \wedge enabled(\alpha, T) \implies legal(S) \wedge (S \overset{c}{\approx} T)$$

The consistency of the two trees follows from the fact that recoloring some nodes or performing a left or right rotation at a node can change the coverage-set of the nodes in the window only (if at

all). Moreover, it can be verified that, with the modification described above, this holds even if a leaf node is added to or removed from the tree.

We are now ready to show that a modify operation is never injected at a "wrong" node in the tree. We first define what it means for a modify operation to be injected correctly in a tree. Given a modify operation α that eventually completes its injection phase successfully, we use $ip(\alpha)$ to denote the node that act as the injection point of α. Specifically, $ip(\alpha)$ is the primary node of the invariant path returned by the seek phase of α.

Definition 8 (correct injection)**.** *Consider a modify operation α that eventually completes its injection phase successfully. Let T be a tree such that $ip(\alpha) \in nodes(T)$. We say that α is* injected correctly *in T if $key(\alpha) \in coverage\text{-}set(ip(\alpha), T)$.*

Note that our algorithm tries to maintain a red-black tree on the set of active nodes. We refer to this tree as the *global tree*. When a modify operation starts its seek phase, it starts by traversing this global tree. As it is traverse the tree, the global tree may undergo structural changes. Depending on the current position of the operation, it may only see a *subset* of these structural changes. As a result, the tree as seen by an operation traversing nodes may be *different* from the global tree. We refer to the tree as seen by a modify operation during its seek phase as its *local tree*. (Note that the same discussion is also applicable to search operations.) As window transactions are executed, the global tree and the local tree of a modify operation may "evolve" in different manner. Specifically, the local tree only sees a subset of the window transactions seen by the global tree. We prove, however, the two trees stay consistent with each other throughout this "evolution" period (which lasts until the operation dereferences a leaf node). This property, in turn, allows us to deduce the following. First, if the injection point of a modify operation is correct with respect to the global tree and the injection point is visible to the local tree, then it is also correct with respect to the local tree. Second, if the injection point of a modify operation is correct with respect to the local tree and the injection point is an active node, then it is also correct with respect to the global tree.

We now formalize the ideas explained above. Consider a modify operation α. Let G denote the current global tree and let L denote the current local tree (for α). Note that G and L are identical when α starts its seek phase. Let c denote the current location of α in L and is given by the node L that was most recently derefenced by α. (In the beginning, when α has not deferenced any node, c is set to $\perp$.) We use α-$\langle G, L, c\rangle$ to model the current *configuration of the system with respect to* α. The configuration with respect to α "evolves" by performing an *action*. An action may be one of the three types:

1. *movement*: α dereferences the next node and moves to a new location in its local tree.

2. *injection*: a new modify operation, say β, is injected into the global tree.

3. *transaction*: a modify operation extant in the global tree, say β, performs a window transaction.

Depending on the current location of α, the last two actions may or may not be visible to α. Each action has a *point-of-occurrence*. For an action of type movement, the point-of-occurence is defined to be the new location of α. For an action of type injection, the point-of-occurence is defined to be the injection point of β. Finally, for an action of type transaction, the point-of-occurence is defined to be the anchor node of β (same as the root of the window transaction). For an action t, let $occurAt(t)$ denote the point of occurence of t. Now, an action t performed in the configuration α-$\langle G, L, c\rangle$ is *visible* to α if and only if $occurAt(t)$ is reachable from c in L using one or more edges. Note that, by definition, the action of type movement is always visible to α (as expected).

Note that we can order modify operations that are eventually injected into the tree by the time at which their injection phase completes. Let α_x denote the xth modify operation to be injected into the global tree. To prove that a modify operation is injected correctly, we use induction on x. Clearly, the first modify operation is always injected at a correct node into the global tree because,

until the first modify operation is injected, the global tree does not undergo any structural changes whatsoever (as a result the local tree of α_1 is identical to the global tree at all times). For induction hypothesis, we have:

Induction Hypothesis: the first $x-1$ modify operations are injected at correct nodes in the global tree

Using the above induction hypothesis and Axiom 1 possibly multiple times, it can be easily verified that:

Proposition 1. *The global tree remains legal at all times until (but not including) the injection of α_x into the tree.*

To complete the proof, it only remains to show that α_x is also injected correctly into the global tree. To that end, we model the "evolution" of the configuration of the system with respect to α_x using a sequence of configurations α_x-$\langle G_i, L_i, c_i \rangle$ for $i = 0, 1, \ldots, s$, where α_x-$\langle G_0, L_0, c_0 \rangle$ denotes the initial configuration and α_x-$\langle G_s, L_s, c_s \rangle$ denotes the final configuration (at the time α_x completes its seek phase). Note that, for the initial configuration, $L_0 = G_0$ and $c_0 = \perp$. Let t_i denote the ith action performed on the configuration. In other words, α_x-$\langle G_i, L_i, c_i \rangle$ is obtained by performing t_i on α_x-$\langle G_{i-1}, L_{i-1}, c_{i-1} \rangle$. Intuitively, L_s represents the local tree traversed by α_x during its seek phase and is the result of performing actions t_1, t_2, …, t_s one-by-one to L_0.

Lemma 27. *L_s is a legal tree and is consistent with G_s.*

Proof. We prove a stronger property, namely, for each $i \in [0, s]$, L_i is a legal tree and is consistent with G_i. The proof is by induction on i. We have:

Base Case (i = 0): From Proposition 1, G_0 is a legal tree. Since $L_0 = G_0$, L_0 is also a legal tree. Also, since $L_0 = G_0$, trivially, L_0 is consistent with G_0.

Induction Step: Assume that the property holds for L_{i-1} for $i \geq 1$. We need to prove that the property also holds for L_i. Before continuing with our proof, we first prove the following claim:

Claim 1. *Consider the four trees G_{i-1}, L_{i-1}, G_i and L_i such that: (i) $G_i \overset{c}{\approx} G_{i-1}$, and (ii) $L_i \overset{c}{\approx} L_{i-1}$. Then*

$$\langle \forall X : X \in nodes(G_i) \cap nodes(L_i) : X \in nodes(G_{i-1}) \cap nodes(L_{i-1}) \implies coverage\text{-}set(X, G_i) = coverage\text{-}set(X, L_i) \rangle$$

Proof. Consider a node $X \in nodes(G_i) \cap nodes(L_i)$. We have:

$$\begin{aligned}
& X \in nodes(G_{i-1}) \cap nodes(L_{i-1}) \\
\implies \quad & coverage\text{-}set(X, G_{i-1}) = coverage\text{-}set(X, L_{i-1}) && \because G_{i-1} \overset{c}{\approx} L_{i-1} \text{(induction step)} \\
\implies \quad & coverage\text{-}set(X, G_i) = coverage\text{-}set(X, L_{i-1}) && \because G_i \overset{c}{\approx} G_{i-1} \\
\implies \quad & coverage\text{-}set(X, G_i) = coverage\text{-}set(X, L_i) && \because L_i \overset{c}{\approx} L_{i-1}
\end{aligned}$$

This establishes the claim. □

Now, continuing with our proof, there are three cases to consider depending on what type of action t_i is.

- **Case 1 ($\mathbf{t_i}$ is of type movement):** Note the primary impact of this action is on the visibility of future actions to α_x. It does not change G_{i-1} or L_{i-1} in any way, that is, $G_i = G_{i-1}$ and $L_i = L_{i-1}$. Since L_{i-1} is legal and is consistent with G_{i-1}, trivially, L_i is legal and is consistent with G_i.
- **Case 2 ($\mathbf{t_i}$ is of type injection):** Note the primary impact of this action is to change the set of extant modify operations in the global tree and possibly the local tree in case t_i is visible to α_x. But otherwise the two trees do not undergo any structural changes. Let

β be the new modify operation injected into the global tree on performing t_i. By our assumption, β is among the first $x-1$ modify operations to be injected into the global tree, and, is therefore, injected correctly.

Legality: If t_i is not visible to α_x, then L_i is identical to L_{i-1}. Since L_{i-1} is a legal tree, it follows that L_i is also a legal tree. On the other hand, if t_i is visible to α_x, then we have to show that β is injected correctly in L_i, that is, $key(\beta) \in coverage\text{-}set(ip(\beta), L_i)$. Note that, in this case, $ip(\beta)$ is present in all the four trees G_{i-1}, G_i, L_{i-1} and L_i. Trivially, G_i is consistent with G_{i-1} and L_i is consistent with L_{i-1}. Therefore, using the claim proved above, $coverage\text{-}set(ip(\beta), G_i) = coverage\text{-}set(ip(\beta), L_i)$. Since $key(\beta) \in coverage\text{-}set(ip(\beta), G_i)$, it follows that $key(\beta) \in coverage\text{-}set(ip(\beta), L_i)$.

Consistency: Note that, every node that is common to G_i and L_i is also common to G_{i-1} and L_{i-1}. Also, trivially, G_i is consistent with G_{i-1} and L_i is consistent with L_{i-1}. Therefore, using the claim proved above, it can be easily verified that G_i is consistent with L_i.

- **Case 3 (t_i is of type transaction):** Note that this action will cause structural changes to the global tree and possibly the local tree in case t_i is visible to α_x.

  Legality: If t_i is not visible to α_x, then L_i is identical to L_{i-1}. Since L_{i-1} is a legal tree, it follows that L_i is also a legal tree. On the other hand, if t_i is visible to α_x, then using Axiom 1 and the fact that L_{i-1} is a legal tree, we can conclude that L_i is also a legal tree.

  Consistency: Using Axiom 1 and the fact that G_{i-1} is a legal tree (using Proposition 1), it follows that G_i is consistent with G_{i-1}. If t_i is not visible to α_x, then $L_i = L_{i-1}$

and thus L_i is consistent with L_{i-1}. Otherwise, using Axiom 1, it follows that L_i is consistent with L_{i-1}. In either case, L_i is consistent with L_{i-1}. Now, there are two subcases to consider:

- **Subcase (a) ($\mathbf{t_i}$ is not visible to $\alpha_\mathbf{x}$):** Consider a node $X \in nodes(G_i) \cap nodes(L_i)$. Note that X is outside the window because it belongs to L_i. Therefore it also belongs to G_{i-1} and L_{i-1}. We can now use the claim proved above to infer that $coverage\text{-}set(X, G_i) = coverage\text{-}set(X, L_i)$. Since X was chosen arbitratily, it follows that G_i is consistent with L_i.
- **Subcase (b) ($\mathbf{t_i}$ is visible to $\alpha_\mathbf{x}$):** Consider a node $X \in nodes(G_i) \cap nodes(L_i)$. There are two possibilities: either X is outside the window or is part of the window. If X is outside the window, then, as in subcase (a), it can be verified that $coverage\text{-}set(X, G_i) = coverage\text{-}set(X, L_i)$. If X is part of the window, then observe that the root of the window lies outside the window and thus has identical coverage sets in G_i and L_i. Due to the tree structure, the coverage sets of nodes in the window are *derived* from the coverage set of the root of the window. Since both G_i and L_i see identical transformations, it follows that, in this case as well, X has coverage sets in G_i and L_i.

Therefore, using induction, we can conclude that L_s is legal and is consistent with G_s. □

Recall that $ip(\alpha_x)$ denotes the injection point of α_x, which α_x finds by traversing its local tree L_s. Note that G_s denotes the global tree at the time α_x completes its seek phase. Let $G_{s'}$ denote the global tree at the time α_x completes its injection phase. Again, using the induction hypothesis (first $x-1$ modify operations are injected correctly), Axiom 1 and Proposition 1 possibly multiple times, it can be easily verified that:

Proposition 2. *G_s and $G_{s'}$ are consistent with each other.*

Note that a modify operation can only be injected at an active node. Hence, we have:

$$ip(\alpha_x) \in nodes(L_s) \cap nodes(G_s) \cap nodes(G_{s'}) \tag{7.1}$$

Using Lemma 27, α_x traverses a valid red-black tree (namely L_s to locate $ip(\alpha_x)$). Thus $key(\alpha_x)$ $\in$ $coverage\text{-}set(ip(\alpha_x), L_s)$. We have:

$$\begin{aligned}
& key(\alpha_x) \in coverage\text{-}set(ip(\alpha_x), L_s) & \\
\implies\ & key(\alpha_x) \in coverage\text{-}set(ip(\alpha_x), G_s) & \because ip(\alpha_x) \in nodes(G_s) \cap nodes(L_s) \text{ (using 7.1) and} \\
& & G_s \overset{c}{\approx} L_s \text{ (using Lemma 27)} \\
\implies\ & key(\alpha_x) \in coverage\text{-}set(ip(\alpha_x), G_{s'}) & \because ip(\alpha_x) \in nodes(G_s) \cap nodes(G_{s'}) \text{ (using 7.1) and} \\
& & G_s \overset{c}{\approx} G_{s'} \text{ (using Proposition 2)}
\end{aligned}$$

Finally, using induction, we can conclude that:

Theorem 11 (correct injection)**.** *Every modify operation is injected at a correct node in the global tree.*

As mentioned earlier, the notion of local tree for a modify operation can also be extended to a search operation, which, in turn, implies from Lemma 27 that:

Theorem 12. *Every search operation traverses a valid red-black tree.*

Operations Yield Correct Results

We use *linearizability* as the correctness condition for executions of our algorithm [34]. Intuitively, a sequence of operations is linearizable if each of the operations appears to take effect at a single

moment between the time that the application invoked the operation and the time that the application received the response, referred to as the *linearization point*, and the result of all operations is the same as it would be if the operations were performed sequentially on the data structure in the order of their linearization points.

As is usually the case, we model an operation using two events: *invocation* and *response*. The invocation event of an operation occurs when the operation is initiated by an application, and its response event occurs when its result is returned to the application. Given an operation α, its invocation and response events are denoted by $inv(\alpha)$ and $resp(\alpha)$, respectively. Note that there may be a delay between when an operation completes and when its response event is generated because the process to which the operation belongs may still be helping other operations complete. If an event e occurs before an event f, then we denote it by $e \sqsubset f$. If an event e occurs before or at the same time as an event f, then we denote it by $e \sqsubseteq f$. It can be verified that $\sqsubset$ is a transitive relation.

We are now ready to prove that every execution of our algorithm is linearizable. We use the *locality property* of linearizability [34]. For an execution history H and an object x, let $H|x$ consists of only those operations of H that involve x. The locality property states that [34]:

A history H is linearizable if and only if, for each object x, $H|x$ is linearizable.

Therefore, if we can prove that an execution of our algorithm projected on an arbitrary key is linearizable, then we have proven that the entire execution is linearizable. For the remainder of the proof, *we fix an execution of our algorithm as well as a key, and consider the projection of the execution on that key.*

To simplify the proof, we treat a modify operation that completes after its seek phase only as a search operation. Specifically, an insert operation that finds the key it is looking for during its seek phase as well as a delete operation that does not find the key it is looking for during its seek phase is treated as a search operation. Note that the concept of local tree of a modify operation defined in Section 7.3.2 can be extended to that for a search operation as well. If an operation is

still pending in the history (that is, it has no response event yet), then we discard it if it is either a search operation or a modify operation that has not performed its terminal window transaction yet.

To capture dependencies between operations, we associate a *rank* with each operation. The rank of an operation α, denoted by $rank(\alpha)$, is defined as follows:

- **Case 1 (α is a modify operation):** Note that we can order all modify operations by the time at which they performed their terminal window transaction. Thus $rank(\alpha)$ is given by the position of α in this sequence (the first operation has the rank of one, the next has the rank of two, and so on). For convenience, we assume the existence of a fictitious delete operation with the rank of zero.

- **Case 2 (α is a search operation):** Consider the local tree of α as described in Section 7.3.2. We need to identify the modify operation that α "reads-from". Let $\mathscr{M}(\alpha)$ denote the set of all modify operations (with the same key as that of α) that contains: (a) all modify operations that performed their terminal window transaction on the local tree of α, (b) all modify operations that performed their terminal window transaction on the global tree before α began its traversal, and (c) the fictitious delete operation. Clearly, $\mathscr{M}(\alpha)$ is non-empty. The modify operation that α "reads-from" is given by the largest rank operation in $\mathscr{M}(\alpha)$, say β. Further, $rank(\alpha)$ is defined to be same as $rank(\beta)$. It can be verified that the type of β (insert or delete) is *compatible* with the result returned by α.

Clearly, we have:

Proposition 3. *No two modify operations have the same rank.*

To obtain a total ordering on operations, we construct a graph on operations, referred to as *dependency graph*. The graph has a vertex for each operation in the history. (Recall that, by our assumption, the history contains operations belonging to a single key only.) The graph has two types of edges:

- *Rank-based edge:* there is a rank-based edge from operation α to operation β, denoted by $\alpha \xrightarrow{r} \beta$, if either: (i) $rank(\alpha) < rank(\beta)$ or (ii) $rank(\alpha) = rank(\beta)$ and α is a modify operation.

- *Time-based edge:* there is a time-based edge from operation α to operation β, denoted by $\alpha \xrightarrow{t} \beta$, if $resp(\alpha) \sqsubset inv(\beta)$.

The next lemma relates rank-based edges with invocation and response events.

Lemma 28. *Given two operations α and β, we have:*

$$\alpha \xrightarrow{r} \beta \quad \Longrightarrow \quad inv(\alpha) \sqsubseteq resp(\beta)$$

Proof. There are two cases to consider:

- **Case 1 (α and β have the same rank):** It implies that α is the modify operation, β is a search operation, and β "reads-from" α. From the way α was selected when defining $rank(\beta)$, $inv(\alpha) \sqsubset resp(\beta)$.

- **Case 2 (α and β have different ranks):** It implies that $rank(\alpha) < rank(\beta)$. Let γ be the modify operation such that $rank(\gamma) = rank(\alpha)$. Likewise, let δ be the modify operation such that $rank(\delta) = rank(\beta)$. Clearly, $rank(\gamma) < rank(\delta)$. (Note that γ may be same as α and δ may be same as β.) Given a modify operation τ, let $last(\tau)$ denote the event when τ completes its terminal window transaction.

  If $\delta = \beta$, clearly, then $last(\delta) \sqsubset resp(\delta) = resp(\beta)$. Otherwise, β is a search operation that "reads-from" δ. By definition, $last(\delta) \sqsubset resp(\beta)$. In either case, we have:

$$last(\delta) \sqsubset resp(\beta) \tag{7.2}$$

Assume, on the contrary, that $resp(\beta) \sqsubset inv(\alpha)$. Then, we have:

$$
\begin{array}{lll}
 & (inv(\gamma) \sqsubset last(\gamma)) \ \wedge \ (resp(\beta) \sqsubset inv(\alpha)) & \\
\Longrightarrow & (inv(\gamma) \sqsubset last(\gamma) \sqsubset last(\delta)) \ \wedge \ (resp(\beta) \sqsubset inv(\alpha)) & \because rank(\gamma) < rank(\delta) \\
\Longrightarrow & (inv(\gamma) \sqsubset last(\gamma) \sqsubset last(\delta) \sqsubset resp(\beta)) \ \wedge & \text{using 7.2} \\
 & (resp(\beta) \sqsubset inv(\alpha)) & \\
\Longrightarrow & (inv(\gamma) \sqsubset inv(\alpha)) \ \wedge \ (last(\gamma) \sqsubset last(\delta) \sqsubset inv(\alpha)) & \\
\Longrightarrow & (\gamma \neq \alpha) \ \wedge \ (last(\gamma) \sqsubset last(\delta) \sqsubset inv(\alpha)) & \\
\Longrightarrow & (\alpha \text{ is a search operation}) \ \wedge & \because rank(\alpha) = rank(\gamma) \text{ and} \\
 & (last(\gamma) \sqsubset last(\delta) \sqsubset inv(\alpha)) & \gamma \text{ is a modify operation} \\
\Longrightarrow & (\alpha \text{ is a search operation}) \ \wedge \ (\{\gamma, \delta\} \subseteq \mathscr{M}(\alpha)) & \text{definition of } \mathscr{M}(\alpha) \\
\Longrightarrow & rank(\alpha) \geq rank(\delta) & \text{definition of } rank(\alpha) \\
\Longrightarrow & rank(\alpha) \geq rank(\beta) & \because rank(\delta) = rank(\beta) \\
\Longrightarrow & \text{a contradiction} &
\end{array}
$$

This estalishes the lemma. □

The next two lemmas prove a useful property about rank-based and time-based edges:

Lemma 29. *The relation induced by rank-based edges is transitive.*

Proof. Consider three distinct operations α, β and γ such that $\alpha \xrightarrow{r} \beta$ and $\beta \xrightarrow{r} \gamma$. By definition of $\xrightarrow{r}$, $rank(\alpha) \leq rank(\beta) \leq rank(\gamma)$. We claim that either $rank(\alpha) < rank(\beta)$ or $rank(\beta) < rank(\gamma)$. Otherwise, $rank(\alpha) = rank(\beta)$, which implies that α is a modify operation, and $rank(\beta) = rank(\gamma)$. which implies that β is a modify operation. Combining the two, we obtain that $rank(\alpha) = rank(\beta)$ and both α and β are modify operations. This contradicts Proposition 3. □

Lemma 30. *The relation induced by time-based edges is transitive.*

Proof. Consider three operations α, β and γ such that $\alpha \xrightarrow{t} \beta$ and $\beta \xrightarrow{t} \gamma$. By definition of $\xrightarrow{t}$, $resp(\alpha) \sqsubset inv(\beta)$ and $resp(\beta) \sqsubset inv(\gamma)$. Clearly, $inv(\beta) \sqsubset resp(\beta)$. Combining all three inequalities, we obtain that $resp(\alpha) \sqsubset inv(\gamma)$. This, in turn, implies that there is a time-based edge from α to γ. □

A smallest cycle in the graph is defined as a cycle with the least number of edges. The following lemma follows from the transitive property of rank-based and time-based edges:

Lemma 31. *A smallest cycle in the dependency graph consists of alternating rank-based and time-based edges.*

Proof. If a cycle contains a rank-based edge from operation α to operation β and a rank-based edge from operation β to operation γ, then two edges can be replaced with a single rank-based edge from α to γ, thereby yielding a smaller cycle. A similar reasoning can be applied to time-based edges. □

We next show that the dependency graph does not contain any cycles.

Lemma 32. *The dependency graph is acyclic.*

Proof. Assume, by the way of contradiction, that the dependency graph contains a cycle. Consider a smallest cycle in the graph. From Lemma 31, the cycle is of the form $\alpha_1 \xrightarrow{r} \alpha_2 \xrightarrow{t} \alpha_3 \xrightarrow{r} \alpha_4 \xrightarrow{t} \cdots \xrightarrow{r} \alpha_{2x} \xrightarrow{t} \alpha_1$, for some x. From Lemma 28, $inv(\alpha_1) \sqsubseteq resp(\alpha_2)$. Also, from the definition of $\xrightarrow{t}$, $resp(\alpha_2) \sqsubset inv(\alpha_3)$. Combining the two, we obtain that $inv(\alpha_1) \sqsubset inv(\alpha_3)$. By repeating this argument, we can show that $inv(\alpha_1) \sqsubset inv(\alpha_3) \sqsubset inv(\alpha_5) \sqsubset \cdots \sqsubset inv(\alpha_{2k-1}) \sqsubset inv(\alpha_1)$. This implies that $inv(\alpha_1) \sqsubset inv(\alpha_1)$—a contradiction. □

As the dependency graph is acyclic, we can sort all operations in some topological order. Note that, in the topological sort, all operations with the same rank occur contiguously with the first operation being a modify operation. It can be verified that the resulting sequence of operations

satisfies all the correctness conditions of linearizability: (i) the sequence respects the real-time order between operations, which is captured by real-time edges, and (ii) all operations are legal, which is captured by the rank-based edges. Formally,

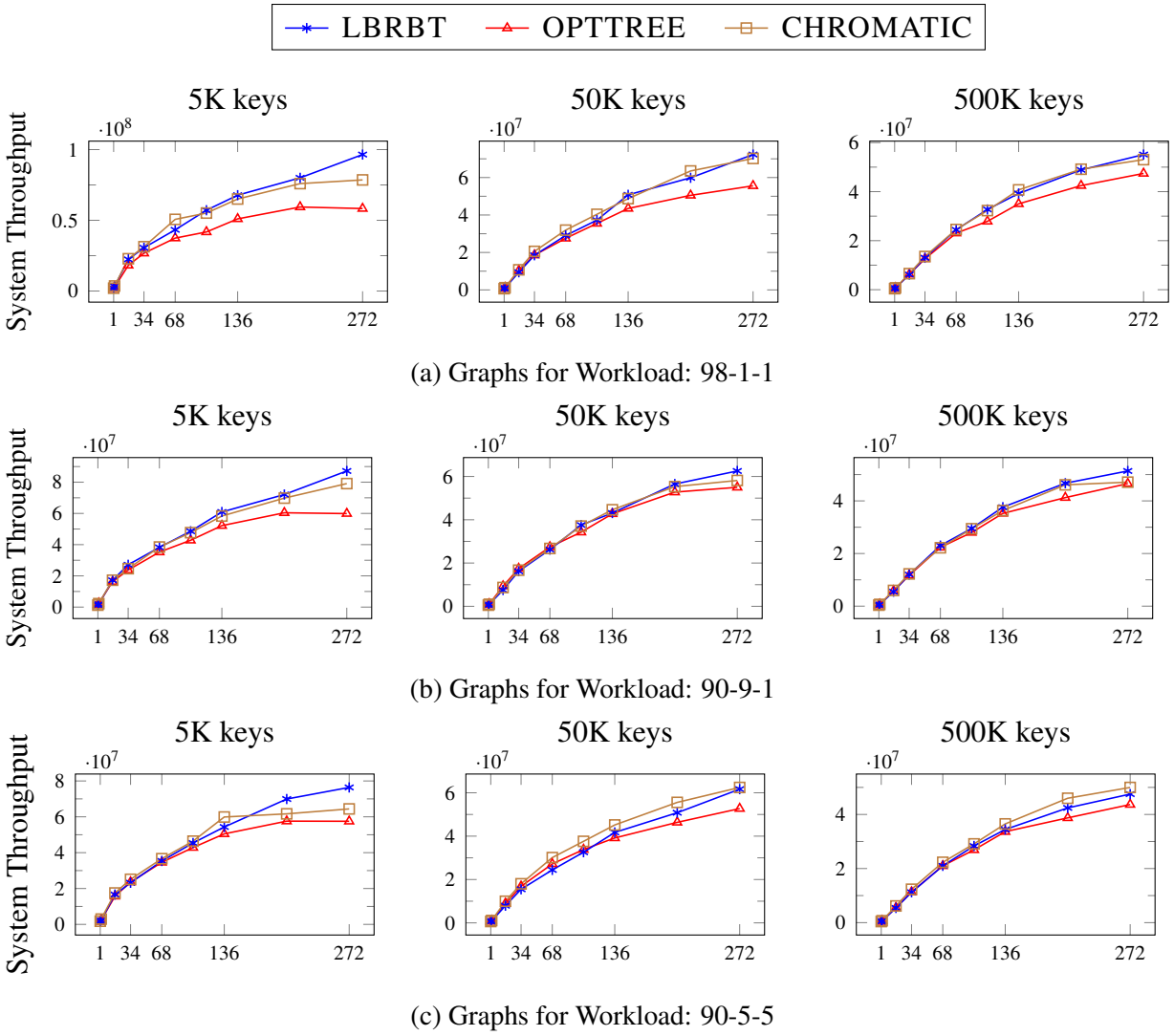
Theorem 13 (linearizability)**.** *Consider a history H generated by our algorithm. Given a key k, let $H|k$ denote the projection of H on k with incomplete operations removed as described earlier. Further, let $DG(H|k)$ denote the dependency graph constructed as described earlier. Then any topological sort of the operations in $DG(H|k)$ yields a sequential history that (i) is legal, (ii) is equivalent to $H|k$ and (iii) respects the order of non-overlapping operations in $H|k$.*

7.4 Experimental Evaluation

Other Implementations Considered: For our experiments, we considered two other implementations of concurrent balanced BST besides the one based on this work, denoted by LBRBT. They are: (i) the lock-based relaxed balanced AVL tree implementation based on Bronson *et al.*'s algorithm [9], denoted by OPTTREE, and (ii) the lock-free relaxed balanced Chromatic tree implementation based on Brown *et al.*'s algorithm described in [10], denoted by CHROMATIC, with rebalancing taking place after 0 violations.

All implementations were compiled using Java SE 8 compiler.

Experimental Setup: To compare the performance of different implementations, we considered 3 common workload distributions for read-dominated workloads consisting of (a) 98% search, 1% insert and 1% delete operations, (b) 90% search, 9% insert and 1% delete operations, and (c) 90% search, 5% insert and 5% delete operations which are considered to be typical workloads likely to be encountered in practice. We varied the maximum size of the tree by considering three different key space sizes consisting 5K keys, 50K keys and 500K keys. Finally, we varied the maximum degree of contention by varying the number of threads from 1 to 272 in appropriate increments. To



(a) Graphs for Workload: 98-1-1

(b) Graphs for Workload: 90-9-1

(c) Graphs for Workload: 90-5-5

Figure 7.3: Comparison of system throughput of different algorithms. Higher the throughput, better the performance of the algorithm.

ensure consistent results, as in [10], rather than starting with an empty tree, we *pre-populated* the tree prior to starting the simulation run.

We compared the performance of different implementations with respect to *system throughput*, which is given by the number of operations executed per unit time.

We conducted our experiments on a Intel Xeon Phi 7250 system containing 68 physical cores each operating at 1.4 GHz with 4 hardware threads per core (272 hardware threads total). The

system contains 16 GB of "near" Multi-Channel DRAM and 112 GB of "far" DDR4 RAM running Linux version 3.1.0.

Simulation Results: Each simulation run was carried out for 8 seconds with a warm up time of 2 seconds and the results were averaged over multiple runs. Figure 7.3 shows the results of our experiments. As all the graphs show, the concurrent AVL tree implementation (OPTTREE) has the worst performance. This is because traversing the search operations are blocking. In the presence of a concurrent write (insert or delete) operation in the same window as the search operation, the write operation may set a `Growing` or `Shrinking` bit associated with a link that the search operation is trying to traverse. During this interval, the search operation has to wait until the bit is cleared. For smaller key space of 5K keys, LBRBT has the best performance. However, as the key space grows to 50K and 500K keys, the gap between LBRBT and CHROMATIC decreases. At its peak in highly read-dominated workloads ($98-1-1$), LBRBT performs as much as 18.5% better than CHROMATIC and 39.5% better than OPTTREE. Also, for a workload of $90-5-5$, it performs 8.2% better than OPTTREE. On the other hand, CHROMATIC performs 4.8% better than LBRBT. However, as the proportion of insert & delete operations increases (esp. beyond 5% of modify operations, the throughput of LBRBT decreases. One of the main reasons for lower throughput of LBRBT is the overhead of the complex insert & delete operations which involves dereferencing up-to 4 pointers for insert operations and up-to 6 pointers for delete operations, combined with locking of more nodes as compared to the simple insert & delete operations. This adversely impacts the performance. However, for read-dominated workloads which are the most common workloads found in practice, LBRBT does have a performance advantage.

Clearly, our experimental results indicate that our lock-based algorithm for a strictly balanced binary search tree performs quite competitively against a lock-free algorithm for a relaxed balanced binary search tree for larger trees, and, in some cases, even beats the latter. This is especially surprising because a red-black tree has generally been considered hard to parallelize because of

its strict balancing requirements. We believe that further research can yield even more efficient concurrent red-black tree algorithms.

CHAPTER 8

CONCLUSION

In this dissertation, we presented a suite of Group Mutual Exclusion algorithms for an asynchronous shared memory system for the cache-coherent model and the distributed shared memory models. Both algorithms use bounded space variables and satisfy the four most important properties of the GME problem, namely group mutual exclusion, lockout freedom, bounded exit and concurrent entering. At the same time, the algorithms have $O(1)$ step-complexity in the absence of any conflicting requests, and $O(1)$ space-complexity per GME object when the system contains $\Omega(n)$ GME objects. To the best of our knowledge, our algorithms are the *first* GME algorithms that have constant complexity for both metrics. Finally, the RMR complexity of our GME algorithm in the cache coherent model depends on the contention encountered by a request, whereas it is optimal for the distributed shared memory model. In our experimental results, our GME algorithm vastly outperformed two of the well-known existing GME algorithms especially for higher thread counts.

We also presented a lock-based algorithm for concurrent manipulation of a red black tree in an asynchronous shared memory system that supports search, insert and delete operations. An important property of our algorithms is that the tree is *strictly balanced*, in other words it is *always* a valid red black tree. The insert and delete operations require locks but the search operations are lock-free. However, using a combination of several ideas (e.g. minimizing copying, reusing nodes, fast-path-slow-path optimization), we have reduced the overhead of modify (insert and delete) operations. Our experimental results indicate that our algorithm has very competitive performance when compared to other concurrent algorithms for binary search trees that provide only relaxed (rather than strict) balancing guarantees. Thus, it may be worth investing time in further research on strictly balanced binary search trees since they are shown to give on-par performance with relaxed variants at least for read-dominated workloads.

For avenues of future work, we plan to extend our GME algorithm so that it provides stronger fairness or concurrency guarantees such as some combination of first-come-first-served (FCFS) [28], first-in-first-enabled (FIFE) [41], strong concurrent entry [41] and pulling [7] among others. We also plan to investigate the *trade-off* between the RMR complexity of a GME algorithm (in the presence of conflicting requests) and its space complexity with large number of GME objects under the CC model. At this point, it is not clear to us if we can design a GME algorithm that has $O(1)$ complexity for both the metrics. We plan to introduce these techniques to other hierarchical data structures such as k-ary trees, B-trees, and other types of hash tables to improve their performance.

For the red black tree algorithm, we plan to provide support for other tree operations such as the (a) replace [58] operation, which inserts a new key in place of an existing key, (b) snapshot [60] operation, which returns a consistent view of the data-structure, and, (c) predecessor and successor, which return the next smallest and largest keys, respectively

REFERENCES

[1] *AMD64 Architecture Programmerś Manual Volume 3: General Purpose and System Instructions*. URL: `http://support.amd.com/us/Processor_TechDocs/24594_APM_v3.pdf`.

[2] *Intel 64 and IA-32 Architectures Software Developerś Manual, Volume 2A: Instruction Set Reference, A-M*. URL: `http://www.intel.com/Assets/en_US/PDF/manual/253666.pdf`.

[3] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan. CBTree: A Practical Concurrent Self-Adjusting Search Tree. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 1–15, 2012.

[4] Z. Aghazadeh, W. M. Golab, and P. Woelfel. Making Objects Writable. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 385–395. ACM Press, July 2014.

[5] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. *Acta Informatica*, 9:1–21, 1977.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent Cache-Oblivious B-Trees. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, July 2005.

[7] V. Bhatt and C. C. Huang. Group Mutual Exclusion in $O(\log n)$ RMR. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 45–54, July 2010.

[8] A. Braginsky and E. Petrank. A Lock-Free B+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 58–67, 2012.

[9] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, January 2010.

[10] T. Brown, F. Ellen, and E. Ruppert. A General Technique for Non-blocking Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[11] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems. In *Proceedings of the 27th International Parallel and Distributed Processing Symposium (IPDPS)*, May 2013.

[12] P. Chuong, F. Ellen, and V. Ramachandran. A Universal Construction for Wait-Free Transaction Friendly Data Structures. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 335–344, 2010.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1991.

[14] T. S. Craig. Building FIFO and Priority-Queueing Spin Locks from Atomic Swap. Technical report, Department of Computer Science, University of Washington, 1993.

[15] T. Crain, V. Gramoli, and M. Raynal. A Speculation-Friendly Binary Search Tree. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 161–170, 2012.

[16] T. Crain, V. Gramoli, and M. Raynal. A Contention-Friendly Binary Search Tree. In *Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 229–240, Aachen, Germany, 2013.

[17] R. Danek and V. Hadzilacos. Local-Spin Group Mutual Exclusion Algorithms. In *Proceedings of the 18th Symposium on Distributed Computing (DISC)*, pages 71–85, October 2004.

[18] T. Davis and R. Guerraoui. Concurrent Search Data Structures Can Be Blocking and Practically Wait-Free. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 337–348, July 2016.

[19] F. Ellen, P. Fataourou, E. Ruppert, and F. van Breugel. Non-Blocking Binary Search Trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, July 2010.

[20] C. Ellis. Concurrency in Linear Hashing. *ACM Transactions on Database Systems (TODS)*, 12(2):195–217, 1987.

[21] P. Fatourou and N. D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2011.

[22] F. E. Fich, D. Hendler, and N. Shavit. On the Inherent Weakness of Conditional Primitives. *Distributed Computing (DC)*, 18(4):267–277, 2006.

[23] M. Fomitchev and E. Ruppert. Lock-Free Linked Lists and Skiplists. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 50–59, July 2004.

[24] D. J. Frank, R. H. Dennard, E. Nowak, Solomon, Y. P. M., Taur, and H. S. P. Wong. Device scaling limits of si mosfets and their application dependencies. In *Proceedings of theInstitute of Electrical and Electronics Engineers*, volume 89, pages 259–288. IEEE, 2001.

[25] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, September 2003.

[26] K. Fraser and T. L. Harris. Concurrent Programming Without Locks. *ACM Transactions on Computer Systems*, 25(2), May 2007.

[27] J. Gibson and V. Gramoli. Why Non-Blocking Operations Should be Selfish. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 200–2014. Springer-Verlag, October 2015.

[28] V. Hadzilacos. A Note on Group Mutual Exclusion. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, August 2001.

[29] T. Harris. A Pragmatic Implementation of Non-blocking Linked-lists. *Distributed Computing (DC)*, pages 300–314, 2001.

[30] Y. He, K. Gopalakrishnan, and E. Gafni. Group Mutual Exclusion in Linear Time and Space. In *Proceedings of the 17th International Conference on Distributed Computing And Networking (ICDCN)*, January 2016.

[31] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.

[32] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[33] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

[34] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[35] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

[36] M. Herlihy, N. Shavit, and M. Tzafrir. Concurrent Cuckoo Hashing. Technical report, Brown University, Providence, Rhode Island, USA, 2007.

[37] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.

[38] S. V. Howley and J. Jones. A Non-Blocking Internal Binary Search Tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 161–171, June 2012.

[39] M. Hsu and W. P. Yang. Concurrent Operations in Extendible Hashing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 241–247, San Francisco, California, USA, 1986.

[40] P. Jayanti. f-arrays: Implementation and Applications. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 270–279, July 2002.

[41] P. Jayanti, S. Petrovic, and K. Tan. Fair Group Mutual Exclusion. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 275–284, July 2003.

[42] M. T. Jones. Inside the Linux 2.6 Completely Fair Scheduler, December 2009. URL: `http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/`.

[43] Y.-J. Joung. Asynchronous Group Mutual Exclusion. *Distributed Computing (DC)*, 13(4):189–206, 2000.

[44] Y.-J. Joung. The Congenial Talking Philosophers Problem in Computer Networks. *Distributed Computing (DC)*, pages 155–175, 2002.

[45] P. Keane and M. Moir. A Simple Local-Spin Group Mutual Exclusion Algorithm. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, 1999.

[46] J. H. Kim, H. Cameron, and P. Graham. Lock-Free Red-Black Trees Using CAS. *Concurrency and Computation: Practice and Experience*, pages 1–40, 2006.

[47] V. Kumar. Concurrent Operations on Extendible Hashing and its Performance. *Communications of the ACM (CACM)*, 33(6):681–694, 1990.

[48] Y. Lev, M. Herlihy, V. Luchangco, and N. Shavit. A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, Castiglioncello, Italy, June 2007.

[49] P. Magnussen, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the International Parallel and Processing Symposium (IPPS)*, pages 165–171. ACM Press, April 1994.

[50] J. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[51] M. M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 73–82, 2002.

[52] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(6):491–504, 2004.

[53] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[54] A. Natarajan and N. Mittal. Brief Announcement: A Concurrent Lock-Free Red-Black Tree. In *Proceedings of the 27th Symposium on Distributed Computing (DISC)*, Jerusalem, Israel, October 2013.

[55] A. Natarajan and N. Mittal. Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Orlando, Florida, USA, February 2014.

[56] A Natarajan, L. H. Savoie, and N. Mittal. Concurrent Wait-Free Red-Black Trees. In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 45–60, Osaka, Japan, November 2013.

[57] K. Platz. *Saturation in Lock-Based Concurrent Data Structures*. PhD thesis, Department of Computer Science, The University of Texas at Dallas, 2017.

[58] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent Tries with Efficient Non-Blocking Snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 151–160, 2012.

[59] R. Sedgewick. Left-leaning Red-Black Trees. URL: `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`.

[60] N. Shafiei. Non-blocking Patricia Tries with Replace Operations. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 216–225, JUL 2013.

[61] H. Sundell and P. Tsigas. Scalable and Lock-Free Concurrent Dictionaries. In *Proceedings of the 19th Annual Symposium on Selected Areas in Cryptography*, pages 1438–1445, March 2004.

[62] M. Takamura and Y. Igarashi. Group Mutual Exclusion Algorithms Based on Ticket Orders. In *Proceedings of the Annual International Conference on Computing and Combinatorics (COCOON)*, pages 232–241, July 2003.

[63] R. E. Tarjan. Efficient Top-Down Updating of Red-Black Trees. Technical Report TR-006-85, Department of Computer Science, Princeton University, 1985.

[64] J.-J. Tsay and H.-C. Li. Lock-Free Concurrent Tree Structures for Multiprocessor Systems. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, pages 544–549, December 1994.

[65] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing (DC)*, 9(1):51–60, 1995.

BIOGRAPHICAL SKETCH

Shreyas Sanjeev Gokhale was born on March 15, 1989 in Maharastra, India. He spent a large part of his childhood in the United States before moving back to India for his high school education. He received his Bachelor of Engineering in Computer Engineering from the University of Pune at Pune in 2012. Thereafter, he moved back to the United States to pursue his Master of Science in Computer Science at The University of Texas at Dallas in 2012. He continued his education there and entered the PhD program in Computer Science at The University of Texas at Dallas in 2014. In his free time, he enjoys watching movies, tv shows, and playing squash.

CURRICULUM VITAE

Shreyas Sanjeev Gokhale

June 28, 2019

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: `shreyas.gokhale@utdallas.edu`

Educational History:

B.E., Computer Engineering, University of Pune, 2012
M.S., Computer Science, University of Texas at Dallas, 2014
Ph.D., Computer Science, University of Texas at Dallas, 2019

Advanced Concurrency Techniques for Concurrent Data Structures
Ph.D. Dissertation
Computer Science Department, University of Texas at Dallas
Advisor: Dr. Neeraj Mittal

Employment History:

Research Assistant, The University of Texas at Dallas, August 2014 – present
Teaching Assistant, The University of Texas at Dallas, August 2014 – May 2019
Software Developer Intern, The MathWorks, Inc., May 2016 – August 2016
Research Intern, Tata Research Development & Design Centre, Pune, August 2011 – May 2012

Professional Recognitions and Honors:

Outstanding Teaching Assistant Award, Engineering and Computer Science, UTD, 2016
Best Project - Research, Tata Research Development & Design Centre, Pune, 2012