2014-09-30

Performance Comparison of Randomized and Deterministic Mutual Exclusion Algorithms

Kahlon, Amandeep

UNIVERSITY OF CALGARY

Performance Comparison of Randomized and Deterministic Mutual Exclusion Algorithms

by

Amandeep Kahlon

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTERS IN APPLIED SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2014

Abstract

Mutual exclusion is a well known in distributed computing. Mutual exclusion comes into exsistence when n processes try to access the Critical Section at the same time. It prevents any two processes from accessing the Critical Section simultaneously. Mutual exclusion is a standard building block for shared memory algorithms.

This thesis presents the performance comparison of various Randomized and Deterministic mutual exclusion algorithms. The performance of these algorithms is compared in the same environment and using the same platform. To perform these comparison tests, time taken by processes to execute mutual exclusion algorithms is measured in isolation, and in data structures (implemented based on mutual exclusion algorithms). Different test cases have been considered to gain some insight about how different algorithms behave under different levels of contention. These test cases involve various combinations of insertion, deletion and look-up operations.

From this comparison tests, we gain some insight about which mutual exclusion algorithms are most resilient to contention. We can use this knowledge while doing concurrent programming. We can choose our mutual exclusion locks based on insertions, deletions and look-ups in the concurrent programming.

Acknowledgements

I wish to thank the following people who encouraged and helped me at each and every point of this thesis. Firstly, I want to thank my supervisor and mentor Dr. Philipp Woelfel for providing me with this wonderful opportunity to come to Calgary and work under your guidance. Your wisdom and encouragement has guided me in maturing academically, professionally, and personally. Thanks for your limitless support, feedback, and for being a source of inspiration during the course of this thesis.

I would also like to thank my co-supervisor Dr. Wojciech Golab for his insight, supervision and constant guidance that he provided during my Masters.

I would like to thank my committee members, Dr. Lisa Higham and Dr. Robert Woodrow for their time and valueable feedback on my thesis.

I would like to thank my family (mom, dad and my brother) for all the love, encouragement and support. I could not have done this without your support.

Finally, I would like to thank Zahra, Maryam for always being there for me. Doing this work without your support would have been infinitely harder.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
U of C	University of Calgary

Chapter 1

Introduction

The Mutual exclusion problem was first defined by Dijkstra in 1962. Then, in 1965, Dijkstra provided the first solution to it [14]. Mutual exclusion algorithms can be used to protect a shared resource such as some parts of shared memory from concurrent access. A mutual exclusion algorithm provides methods `lock()` and `release()`. After a process completes the `lock()` method and before it starts executing the `release()` method call, a process is in the Critical Section. The methods `lock()` and `release()` ensure that only one process is in the Critical Section at any time. This property is called mutual exclusion property. Progress properties ensure that some processes make progress towards capturing the lock. The most important progress properties are deadlock freedom and starvation freedom. Deadlock freedom assures that out of all the processes calling `lock()`, at least one completes the `lock()` method call, provided all processes (executing the lock or release methods) take sufficiently many steps and the Critical Section is finite. Starvation freedom assures that all processes calling `lock()` will eventually succeed provided all processes (executing the lock or release methods) take sufficiently many steps and the Critical Section is finite.

Since Dijkstra proposed the problem, researchers proposed several mutual exclusion algorithms with various properties. Initially, mutual exclusion algorithms were designed for two processes only, e.g. the algorithms by Dekker [14] and Peterson [21]. Later, researchers designed mutual exclusion algorithms for any number of processes. In the beginning, research did not take in account how hardware can affect the performance of algorithms, i.e. how the speed of the processor memory interconnect and locality of data can affect the performance. When an operation needs to traverse the processor memory interconnect for accessing the shared memory, it is refered to remote memory reference (RMR). Starting with Yang and

Anderson [1], reseachers started taking *remote memory references* into account when designing mutual exclusion algorithms.

In recent years, a major focus of mutual exclusion research was on minimizing the number of remote memory references. Reseachers came up with local spin algorithms, in which processes busy-wait on shared memory variables that are locally accessible. This bounds the number of remote memory references a process incurs. The time complexity in these algorithms is measured in terms of the remote memory references a process incurs in the lock and release methods. There are several mutual exclusion algorithms which have O(1) RMR complexity, e.g. the MCS lock [26] and the CLH lock [22], but they use strong primitives such as `swap()` and `getAndIncrement()`.

In a randomized mutual exclusion algorithm, an adversary controls the scheduling of the steps by different processes. Different types of adversary models that have been considered are the oblivious, the weak, and the strong adaptive adversary. The oblivious adversary is one which has to make all scheduling decisions in advance, independent of processes random choices. This means that coin flips by a process will have no impact on the scheduling. The strong adaptive adversary is one which sees the result of every coin flip and can use that knowledge for later scheduling decisions. The weak adversary sees the coin flip result only after the process has taken a step after the coin flip. Hendler and Woelfel [13] presented a randomized mutual exclusion algorithm with $O(\log n \ / \ \log\log n)$ RMR complexity against the strong adaptive adversary which uses `compareAndSwap` objects (CAS) (see Section 2.2.2) and read-write registers. Giakkoupis and Woelfel [16] also presented a tight lower bound of $\Omega(\log n \ / \ \log\log n)$ for the RMR complexity of deadlock-free randomized mutual exclusion algorithms against the strong adaptive adversary in both Cache-Coherent (CC) and Distributed Shared Memory (DSM). In an unpublished paper, Giakkoupis and Woelfel [16] present a randomized mutual exclusion algorithm against the oblivious adversary (having lower bound of $O(\log n/ \log\log n)$).

The main motivation behind this thesis is comparing the performance of the randomized mutual exclusion algorithm proposed by Giakkoupis and Woelfel against other well known mutual exclusion algorithms such as the MCS lock [26], the CLH lock [22], the TAS [4], the TTAS [4], mutual exclusion algorithm using registers [1], the randomized mutual exclusion algorithm by Hendler and Woelfel [13] and the Java re-entrant locks. The performance of these algorithms is compared in the same environment and using the same platform. The main focus of the thesis is to test these algorithms under different test conditions. From these performance tests, we gain some insight about the mutual exclusion algorithms in various scenarios. We also gain some information about how different algorithms behave under different levels of contention (number of processes executing an algorithm concurrently).

To perform these comparison tests, time taken by processes to execute mutual exclusion algorithms is measured in isolation, and in data structures (implemented based on mutual exclusion algorithms). The time taken by each process to execute the `lock()` and `release()` method is recorded in both the cases. Then the average of total time is represented graphically. When comparing the performance of mutual exclusion algorithms in data structures, coarse-grained and fine-grained locking techniques are considered.

The rest of the thesis is organized as follows. Chapter 2 describes the asynchronous shared memory model. It also defines the mutual exclusion problem and its different progress properties. In Chapter 3, we describe several mutual exclusion algorithms in detail and the methodology of our experiments is explained in Chapter 4. Chapter 5 presents the detailed test results of the experiments conducted on mutual exclusion algorithms, and in Chapter 6, we summarize our contributions, limitations of this work, and some oppurtunities for future work.

Chapter 2

Preliminaries

2.1 System Assumptions

The model used is *asynchronous*, in which processor activites can be delayed by the events such as interrupts, preemptions, failures etc. These delays are unpredictable and can vary in durations. The asynchronous shared memory system consists of fixed number of processes n which have unique and consecutive IDs, starting from either 0 or 1. In this thesis, we assign consecutive IDs to the processes in order to implement mutual exclusion algorithms. The processes communicate by operations on *shared objects*. An operation is either atomic or non-atomic. An operation is said to be atomic if it completes in a single step. A non-atomic operation is an operation which completes in multiple steps. Each process executes its program at varying speed. Processes do not fail in the system. A process take steps in the system until it terminates i.e. there are no more steps to take.

2.2 Standard Shared Object Primitives

The common shared objects provided by many multiprocessor architectures are `TestAnd-Set` (TAS) and `Compare-And-Swap` (CAS). Atomic TAS and CAS operations can be used to protect a shared resource (shared data structure or shared device) from being concurrently accessed by multiple processes. TAS and CAS objects are often used in the implementation of the mutual exclusion algorithms. TAS and CAS objects are explained in detail in Section 2.2.1 and 2.2.2.

2.2.1 Test And Set (TAS)

A `TAS` object `X` stores a boolean value. Initially, the value of the object is false. The object supports two operations, `X.TAS()` and `X.reset()`. `X.TAS()` operation writes true and returns the previous value of the object [38]. `X.reset()` operation sets the value of `X` to false.

2.2.2 Compare and Swap (CAS)

A `CAS` object `X` supports two atomic operations `X.CAS()` and `X.Read()`. Operation `X.Read()` returns the value stored in `X`. The operation `X.CAS(`*exp*`,`*new*`)` takes two arguments *exp* and *new* and returns a boolean value. The `X.CAS(`*exp*`,`*new*`)` operation tries to change the value of X from *exp* to *new*. If the value of `X` equals *exp* then `X.CAS(`*exp*`,`*new*`)` succeeds and returns true and the value of `X` is changed to *new*; otherwise it fails and false is returned and the value of `X` remains unchanged.

2.3 Atomic Integer (AI)

An Atomic Integer is most commonly available in Java and extends a *Number* class [28] in Java. Atomic Integer class is useful in the implementing mutual exclusion and other concurrent algorithms. An Atomic Integer `X` stores integers only and supports the operations `getAndIncrement()`, `getAndDecrement()` and `getAndSet()`. The intial value of `X` is 0. The `X.getAndIncrement()` operation increments the current value of `X` by one and returns its previous value. The `X.getAndDecrement()` operation decrements the current value of `X` by one and returns its previous value. Operation `X.getAndSet(`*newvalue*`)` takes one argument *newvalue* and atomically sets the value of `X` to *newvalue* and returns the previous value [27].

2.4 Asynchronous Shared Memory Architecture

In this thesis, the asynchronous shared memory architectures based on cache coherence (CC) and distributed shared memory (DSM) are considered (see Figure 2.1). They are defined as follows:

2.4.1 CC model

In a CC machine, each processor has a local cache. A cache coherence protocol is used to obtain cache coherence i.e. to maintain consistency of data in the memory. In the CC model, the shared memory is external memory which can be accessed by all processors. Each shared variable is stored in shared memory. When the value of a shared variable changes, the copies of that shared variable in the local caches are invalidated. Two types of cache coherence protocols are write-through and write-back caching.

In write-through caching, to read the value of register R, a process p checks whether it has a valid cached copy of R or not. If it does, p obtains the value of R from the cache ("cache hit"). Otherwise it accesses the shared memory to obtain the value of R ("cache miss"); this is called a *remote memory reference*. To write the value, a process p updates the value of R in the shared memory causing an RMR and all copies of R in all local caches are invalidated.

In write-back caching, each cached copy is held in one of two modes, shared or exclusive. To read R, a process p first checks whether it has a cached copy of R in either mode, and if it does, p reads R without accessing the remote memory. Otherwise, it incurs a remote memory reference that creates a cached copy of R in shared mode and invalidates any copy of R held by other processes in exclusive mode. To write R, process p makes sure that it has R in exclusive mode. If it has a cached copy of R in exclusive mode, then it updates R in its cache without causing any remote memory reference. Otherwise it incurs a remote memory reference to create a cached copy of R in exclusive mode and writes back R to the shared memory invalidating all other cached copies of R [17].

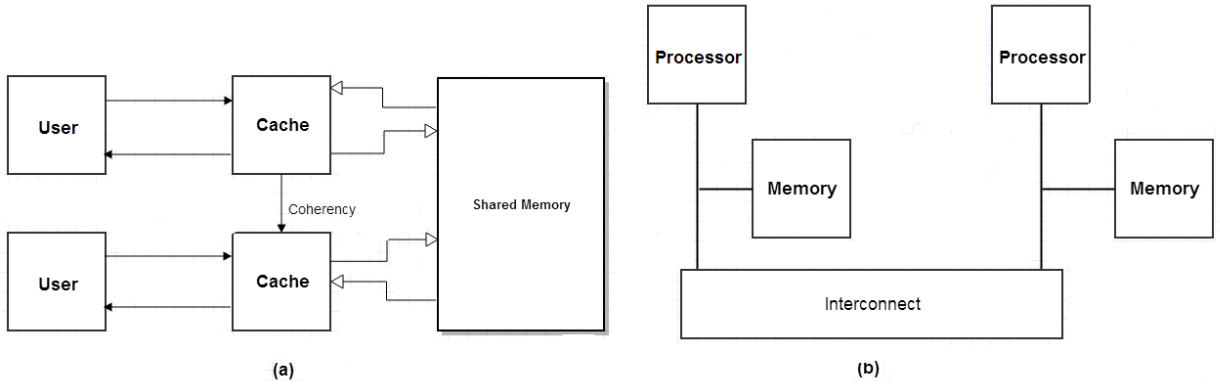


Figure 2.1: (a) CC model (b) DSM model

2.4.2 DSM model

In DSM systems, each process has its own local memory segment. Each memory segment is local to the process owning it and is remote to all other processes. The local memory segments of all processes form the shared memory of the system. The local memory of each process can be accessed using the interconnection network. Each process can access the local memory segment it owns without traversing the interconnection network. A process can access the variables in other process's local memory segments by using the interconnection network; each such access is called a *remote memory reference*.

2.5 Mutual Exclusion

The mutual exclusion problem was first identified by Dijkstra in 1962 and later in 1965, Dijkstra [14] presented a solution to the mutual exclusion problem. Mutual exclusion is useful when n processes, concurrently try to access the same shared resource. Dijkstra's algorithm provided a solution which prevents any two processes from using the same shared resource simultaneously. Since then the mutual exclusion problem gained the interest of many researchers. Mutual exclusion is a standard building block for shared memory algorithms.

The mutual exclusion problem can be defined in terms of *Lock* objects. The type Lock provides the methods, `lock()` and `release()`, which do not return any value. A process must alternate `lock()` and `release()` method calls and it can call the `release()` method only after it called the `lock()` method.

A process owns the lock if it has completed its `lock()` method but has not started its `release()` method. A process which owns the lock can release it by calling `release()`. A process which has started but not finished the `lock()` method is in the Entry Section. A process which has started but not finished the `release()` method is in the Exit Section. A process which has completed the `lock()` method call but has not called `release()` is in the Critical Section. A process outside the Entry, Exit and Critical Sections is in the Remainder Section.

A mutual exclusion algorithm should satisfy the following properties:

Mutual Exclusion: At any time, at most one process can be in the Critical Section.

Deadlock Freedom: If some of the processes are in the Entry Section, then some process will eventually be in the Critical Section provided all the processes which are not in the Remainder Section take enough steps [21].

Mutual exclusion is a *safety* property, i.e. it ensures that the program is correct. Deadlock freedom is a *progress* property, i.e. out of all the processes trying to access the lock at least some of them make progress and complete the `lock()` method call, provided all processes

take sufficiently many steps in the system. This ensures that the system never freezes.

There are some stronger progress properties which some mutual exclusion algorithms satisfy:

Starvation Freedom: Every process that attempts to capture the lock by calling the `lock()` method will eventually succeed provided all processes which are not in the Remainder Section take sufficiently many steps, i.e. every `lock()` method call will eventually return. Starvation freedom is stronger than deadlock freedom, i.e. it ensures deadlock freedom but not vice versa.

First Come First Served (FCFS): A `doorway` is a piece of code which is wait-free. To define the FCFS property, processes are required to execute a `doorway` at the beginning of the `lock()` method. A mutual exclusion algorithm satisfies the `FCFS` property, if in any execution, where some process P finishes the `doorway` before some process Q starts it, P enters the Critical Section before Q does.

Wait Freedom: A piece of code is said to be wait-free if a process completes it in finite number of steps. **Bounded Exit**: Bounded Exit means that the processes execute the `release()` method in a bounded number of steps i.e. the `release()` method is wait-free.

Re-entrant Mutual Exclusion: Re-entrant mutual exclusion provides a recursive locking mechanism, i.e. the same process can acquire the same lock multiple times while it is holding the lock. A mutual exclusion algorithm is re-entrant, if a process calls a `lock()` method while it is in the Critical Section. A lock in the mutual exclusion algorithm is made re-entrant if it guarantees thats the progress properties will hold even if a process execute the `lock()` method call in the Critical Section. Note that in a non re-entrant mutual exclusion algorithm, a process which executes the `lock()` method in the Critical Section then may deadlock itself as it is trying to acquire a lock which it already holds.

2.6 Local Spin Algorithms

We say that a process is *busy-waiting* or *spins* on a variable if it repeatedly reads that variable until the variable has a value satisfying some condition. *Local* spinning means that the process is busy-waiting on a locally accessible spin variable (the variable which is stored in the local caches or the local memory segment of the spinning process and which can be accessed without causing remote memory reference. In CC machines, a process spins on a cached copy of the shared variable and does not incur an RMR until a "cache miss" occurs. In DSM machines, every process needs to use its own spin variable, i.e. a spin variable stored in its local memory segment. Busy-waiting on these spin variables does not cause any remote memory reference.

2.7 RMR complexity of Mutual Exclusion Algorithms

Let e be an execution in which process p_i calls the `lock()` and `release()` method k_i times, where $k_i \in \{0, \infty\} \cup N$. Then the RMR complexity of e is:

$RMR(e) = MAX_{\{P_{i,1 \leq i \leq n\}, \{1 \leq j \leq ki\}}}\ RMR(e)_{\{P_{i,j}\}}$

where, $RMR(e)_{\{P_{i,j}\}}$ is the number of RMRs incurred by process P_i in its passage j i.e. its j^{th} `lock()` and `release()` calls combined. The RMR complexity of mutual exclusion algorithm A is the maximum number of $RMR(e)$ for all executions e. Often, the performance of shared memory algorithms is measured in terms of step complexity i.e., the maximum number of steps taken by any process to finish its algorithm. In mutual exclusion algorithms, busy-waiting of a process can be unbounded (a process can incur an unbounded number of steps) as some other process owning the lock might be in the Critical Section for an unbounded amount of time. There are some local spin algorithms, where processes incur only a bounded number of RMRs. RMRs take much more time than "cache hits" and local memory accesses. Therefore, it is better to use RMR complexity as a metric to measure the time complexity of mutual exclusion algorithms [24].

Chapter 3

Related Work

3.1 2-Process Mutual Exclusion

There are various well-known mutual exclusion algorithms for n-process. The most common 2-processes mutual exclusion algorithms [14] are Dekkers's algorithm and Peterson's algorithm. Peterson's algorithm is explained in detail below. Dekker's mutual exclusion algorithm is a bit more complex and cannot be easily extended to n processes. In Peterson's algorithm, a shared variables *victim* and a boolean array *flag*[] is used (see Figure 3.1). In the `lock()` method, process p sets `flag[`p`]` to true in line 1 to indicate that it wants to enter the Critical Section. Then the process writes its ID to *victim* in line 2. In line 3, a process busy waits as long as it reads its own ID in, *victim* or until the *flag* of the other process is true. If either of the above conditions fails then a process enters the Critical Section.
Assume that processes have IDs 0 and 1.

Class `Lock`

shared: boolean $flag$[0,1]
int *victim*

Method `lock()`

```
1  flag[p].write(True)                    /* I'm interested*/
2  victim.write(p)                        /* you go first*/
3  while flag[1 − p].read()=True ∧ victim.read()=p do
4  end
```

Method `release()`

```
5  flag[p].write(False)                   /* I'm not interested*/
```

Figure 3.1: Peterson's Lock Implementation

Lemma 1. *Peterson's lock provides mutual exclusion for two processes*

Proof. Suppose at time t, both processes, q and $1-q$, are in the Critical Section. Each process must have set its $flag$ to True in line 1 and later written its ID to *victim* in line 2. Assume that process $q \in \{0, 1\}$ executes the write operation in line 2 at time t′ after process $1-q$ does. Then at time t′, both processes must have already executed line 1. Then throughout [t′,t], `flag[`$1-q$`]` = `flag[`q`]` = `True` and $victim = q$. So, process q does not finish the while-loop during [t′,t] and does not enter the Critical Section at time t. A contradiction. □

Lemma 2. *Peterson's lock provides starvation freedom.*

Proof. Assume that the Critical Section is finite and also assume for the purpose of contradiction, that processes q and $1-q$ execute infinitely many steps but q does not finish. This means that every time process q, reads *victim*, the value of *victim* is q or it reads the value of $flag[1-q]$ true. While process q busy-waits, process $1-q$ may be repeatedly entering and leaving the Critical Section or $1-q$ may be either in the Remainder Section or it may be spining in the while loop for an unbounded number of steps. These cases are discussed seperately:

Case 1: Process $1-q$ is repeatedly entering and leaving the Critical Section

If process $1-q$ ever enters the Critical Section again, it will set `victim =` $1-q$ in line 2 of `lock()` and never changes *victim* back to q. Once process q reads `victim =` $1-q$, it will finish the while loop and enter the Critical Section.

Case 2: Process $1-q$ is in the Remainder Section forever

If $1-q$ is in the Remainder Section, then $flag$`[`$1-q$`] = false` and q will finish the while loop and enter the Critical Section.

Case 3: Process $1-q$ busy-waits in the while loop forever

In this case `victim =` $1-q$ and thus q will enter the Crtitical Section eventually.

This proves that process q does not starve. □

RMR Complexity of Peterson's lock. In the DSM model, the RMR complexity is unbounded. Suppose *victim* is in q's memory segment. If process $1 - q$ busy-waits in line 3 in Figure 3.1, it repeatedly reads the value of *victim* from q's memory segment. Because q can be in the Critical Section for an arbitrary amount of time, process $1 - q$ may incur an unbounded number of RMRs.

In the CC model, the RMR complexity of Peterson's lock is constant. A process incurs at most a constant number of RMRs (i.e. 3 RMRs) in lines 1, 2 and 5 in Figure 3.1. A process q incurs a constant number of RMRs in line 3 of Figure 3.1 because neither $flag$ nor *victim* can change a constant number of times until q enters the Critical Section. Suppose either $flag$ or *victim* change more than a constant number of times during the time interval during which q does not finish the while loop. As the algorithm is starvation free so while process q is waiting in the while loop, $1 - q$ can get the lock a constant number of times. The above statement implies that process $1 - q$ can change $flag$ and *victim* a constant number of times.

3.2 Mutual Exclusion Algorithms for More than Two Processes

There are various n-process mutual exclusion algorithms. Examples are Lamport's Bakery Algorithm [21] and the Filter lock [21]. Both are not local spin algorithms. A very early local spin mutual exclusion algorithm with constant RMR complexity is the CLH queue lock [22]. One drawback of this algorithm is that it performs poorly on cache-less memory models. Later, Mellor-Crummey and Scott [26] presented another n-process mutual exclusion algorithm that uses local spinning. Since then, the local spin mutual exclusion algorithms have gained considerable interest of researchers, and various local spin mutual exclusion algorithms have been devised. The overview of some of better known local spin mutual exclusion algorithms are explained in Section 3.2.1.

Adaptive local spin mutual exclusion algorithms are those in which the RMR complexity depends on the number of processes calling the `lock()` method concurrently, i.e., the RMR complexity is a function of the contention. Adaptive mutual exclusion algorithms will be explained in detail in Section 3.3.3.

3.2.1 Overview of Local Spin Algorithms

This section gives the high level description of some of the better known local spin mutual exclusion algorithms.

Some of the first local spin algorithms [4, 26] used *read-modify-write* operations. Read-modify-write operations read the memory location and write a new value to the memory location atomically. Some of the examples of read-modify-write operations are `testAndSet()`, `fetchAndAdd()` and `getAndSet()`.

In the CLH mutual exclusion algorithm, the `getAndSet()` operation is used. The CLH algorithm has O(1) RMR complexity. Each process has a node, $myNode$ of type $Node$ which has a boolean *locked* field and a reference to some other node object. Processes which call the `lock()` method form a list. The list has a *tail* pointer which points to the most re-

cent added node in the list. Each process appends its node at the *tail* by performing a `getAndSet(myNode)` operation. The `getAndSet()` operation returns reference to its predecessor node in the list. Each process then sets its *locked* field to *true* to indicate that it wants to enter the Critical Section. It then spins on the *locked* field of the predecessor node until the value of the *locked* field of the predecessor node changes to $false$. When the *locked* field becomes $false$ then the process enters the Critical Section. The CLH lock is explained in detail in the Section 3.3.1.1.

Mellor-Crummey and Scott presented a locking algorithm which also forms a queue and has O(1) RMR complexity. The MCS mutual exclusion uses the same primitives as the CLH lock. The only difference between the two is that each process, after appending its node in the list and setting its *locked* field to *true*, spins on its own node until the value the of *locked* field changes to $false$. When the *locked* field reads $false$ then the process enters the Critical Section. This algorithm is better suited for cache-less architectures in comparison to the CLH lock. The major drawback of this algorithm is that in the `release()` method, processes have to wait if there is any slow process i.e., there is no "bounded exit" (for details see Section 3.3.1.2 on Page 23).

Anderson [4] presented two simple mutual exclusion algorithms, the *Test and Set (TS)* and the *Test and Test and Set (TTS)* lock, which have unbounded RMR complexity. In the TS and TTS lock, the `compareAndSwap()` operation is used. In the TS mutual exclusion algorihtm, each process performs `compareAndSwap(True, False)` on a shared object. If the `compareAndSwap()` operation is successful, then that process enters the Critical Section. Otherwise it repeatedly performs `compareAndSwap()` on the shared variable until it enters the Critical Section. In the `release()` method, each process resets the TS by performing `compareAndSwap(False, True)`. In theory, to implement the TS mutual exclusion algorithm, a `testset` object is used (see Section 3.3.1.3 on Page 26) but in Java, the `compareAndSwap()` operation is used to implement it. In the TTS mutual exclusion algo-

rithm, each process first reads the shared object and spins on it if it reads *true*. When the shared object becomes $false$ then the process performs a `compareAndSwap()` operation on it. Like TS, in the `release()` method, each process resets TS by performing `compareAndSwap()`. Both algorithms are explained in detail in Section 3.3.1.3.

Peterson and Fischer [30] proposed an implementation of an n-process mutual exclusion by applying 2-process mutual exclusion to an arbitration tree. The algorithm uses the `compareAndSwap()` operation. In this algorithm, each process is associated with a unique leaf of the tree. Each internal node in the arbitration tree has a lock associated with it. Each process acquires the locks of all nodes on the path from its leaf node to the root node. In order to win a lock on a node, a process performs a `compareAndSwap()` operation. All busy-waiting on the nodes is done by local spinning. The process that wins the lock at the root node enters the Critical Section. In the release() method, the process follows the same path from the root node to the leaf node, releasing all locks that it acquired while going up. The RMR complexity of this algorithm is $\theta(\log\ n)$. The major drawback of this algorithm is that the RMR complexity is $\theta(\log\ n)$ on CC and DSM models even when there is no contention.

Yang and Anderson [1] proposed a mutual exclusion algorithm for n processes that uses atomic read and write operations and in addition, achieves constant RMR complexity if a process runs alone. In Anderson and Yang's algorithm, 2-process mutual exclusion is applied to an arbitration tree. Peterson's mutual exclusion algorithm is used to implement 2-process mutual exclusion on each node. The other difference to Peterson and Fischer's algorithm is the "fast-path mechanism". This mechanism enables a process to bypass the arbitration tree when there is no contention. In this algorithm, a process p first determines whether there are any competing processes. If there is no competing process then process p follows the *fast path* otherwise it is *deflected* from the fast path and it has to capture all the locks on the nodes from the leaf node to the root node. Under no contention, the RMR complexity

is O(1) and under contention, it is O(n) in the worst-case rather than O(log n). An RMR complexity of O(n) in the worst case is a major drawback of this algorithm. This is because the fast path has to be "re-opened" after the contention ends. To do this, each process polls every other process to see whether it is still contending.

Anderson and Kim [3] introduced a new fast path mechanism which, when used with Yang and Anderson's [1] mutual exclusion algorithm, provides O(1) RMR complexity without contention and O(log n) otherwise. The fast path mechanism has the novel feature that it can be re-opened without having to poll each process to see whether it is still contending.

Attiya, Hendler and Woelfel [8] presented tight lower bounds for deterministic mutual exclusion algorithms. Their main result is an $\Omega(\log n)$ lower bound of the RMR complexity of deterministic mutual exclusion algorithms.

Hendler and Woelfel [13] presented a strong-adversary randomized mutual exclusion algorithm which has an expected RMR complexity of O(log n/ loglog n). A strong adversary can observe the entire history of the system, including the results of the coin flips, the states of the processes and the state of objects but it cannot predict the future coin flip outcomes [6]. In Hendler and Woelfel's algorithm, a tree of height θ(log n/ loglog n) with n leaves is used. A process p can enter the Critical Section either by capturing all the locks on the path from a leaf node to the root node or p can be promoted by a process q when q exits the Critical Section. A process, while capturing the locks on the path from a leaf node to the root node, performs a `compareAndSwap()` operation at each node. When process p is promoted, then it need not climb up the tree but it busy-waits until it is notified to enter the Critical Section. Two types of promoting mechanisms, randomized and deterministic promotions are used. They are explained in detail in Section 3.3.1.6.

Giakkoupis and Woelfel [16] presented a tight lower bound of $\Omega(\log n \ / \ \log\log n)$ for the RMR complexity of deadlock-free randomized mutual exclusion algorithms against strong adaptive adversary in both CC and DSM model.

Bender and Gilbert [9] presented an oblivious-adversary mutual exclusion algorithm that achieves amortized $O(\log^2 \log n)$ RMR complexity with high probability. In this algorithm, a shared approximate counter C and a waiting array is used. Counter C supports increment and decrement operations and in the waiting array, the processes spin, waiting for their turn to enter the Critical Section. When a process calls `lock()`, it increments C and then reads the value of C. It then uses that value to find a free spot in the waiting array. For instance, if a process reads $C = k$, then it randomly searches for a spot in the first $\theta(k)$ slots in the array and when it finds a spot, it spins on it. A process spins on a slot in a waiting array until it is notified to enter the Critical Section. When a process exits the Critical Section, it reads the value of C. Suppose the value of C is k. The exiting process then randomly searches for a process in the first $\theta(k)$ slots of array. It then removes itself from the array and notifies the process in the selected slot to enter the Critical Section and decrements the counter C.

3.3 Detailed Decription of Local Spin Algorithms

This section gives the detailed descritpion of the various well known local spin mutual exclusion algorithms.

3.3.1 Queue Locks

In queue locks, processes calling the `lock()` method form a queue. In the queue, each process waits for its turn to enter the Critical Section by repeatedly checking that its predecessor has finished. Cache coherence traffic is reduced because all processes spin on different node locations rather than spinning on the same location. Queue locks also provide First-Come-First-Served fairness. Each process that calls the lock() method, appends its node at the tail of the queue. Typically, this append operation is wait-free and the process which appends its node first will enter the Critical Section first. This ensures the FCFS property in queue locks. There are different queue locks, for instance the Array-Based Lock [19], the CLH Queue Lock [22] and the MCS Queue Lock [26]. In this thesis, only the CLH lock (see Section 3.3.1.1) and the MCS lock (see Section 3.3.1.2) are considered as the Array-Based Lock is not space efficient. It allocates an array of size n for every lock, where n is number of threads calling the `lock()` method.

3.3.1.1 The CLH Queue Lock

Figure 3.2 explains the CLH lock. Each process has a node of type $Node$ which has a boolean *locked* field and a reference, *ptr*, to some other node object. Processes that call the `lock()` method form a list. The list has a *tail* pointer which points to the last node of the list. If the value of *tail* is null, then the list is empty. Each node in the list uses *ptr* to point to a predecessor field in the list. If the *ptr* is null, then the node has no predecessor, and thus is the first node of the list. Each process appends its node at the *tail* by performing a `getAndSet()` operation with its own node as an argument.

In line 1 of the `lock()` method of the CLH mutual exclusion algorithm (see Figure 3.2), each

process sets the *locked* field of its node to true to indicate that it wants to enter the Critical Section in line 1, and then in line 2 it performs a `getAndSet(myNode)` to append its node at the *tail* and it simultaneously acquires the reference to its predecessor node (if any). It then sets the value of $myPred$ to its predecessor in line 3. Later, each process checks the *locked* field of its predecessor in line 4. If the *locked* field of the predecessor is true then it spins in line 4 until the *locked* field becomes false indicating that the lock has been released by the predecessor.

In the `release()` method, a process sets the *locked* field of its node to false in line 6. For future lock accesses, a process can re-use the predecessor node, as at that point in time the predecessor node is no longer in use. Therefore, it stores the reference to that node in $myNode$ in line 7.

It is obvious from the structure of the list that the CLH lock satisfies deadlock freedom and FCFS; processes enter the Critical Section in the order in which they perform `getAndSet()` at the *tail*.

The CLH lock satisfies mutual exclusion and FCFS

Suppose at time t, two distinct processes p and q, are in the Critical Section. Each process must have set the *locked* field of its node to true in line 1 and must have obtained a reference to its predecessor node in line 2. Assume that process q executes its `getAndSet()` in line 1 at time t' after p does. As per the structure of the list, the process which performs the `getAndSet()` operation first enters the Critical Section first. So, process q cannot be in the Critical Section at the same time as p. This proves that at most one process can be in the Critical Section at any time.

Expected RMR Complexity of the CLH lock

In the CC model, the RMR complexity of the CLH lock is O(1): Each process p gets a cached copy of the *locked* field of the predecessor node and repeatedly reads the *locked* field until the value of the *locked* field changes to false. Process p incurs a "cache miss" after p's

predecessor has released the lock and invalidated the value of the *locked* field in p's cache. When process p reads `locked = false`, it enters the Critical Section, hence p incurs only a constant number of RMRs in the `lock()` method. In the `release()` method, each process incurs a constant number of RMRs as it only sets its *locked* field to false. Therefore, in overall in the CLH mutual exclusion algorithm, each process incurs a constant number of RMRs (at most 3 RMRs).

In the DSM model, the RMR complexity is unbounded: Suppose the *locked* field of p's node is in the local memory segment of process q (possibly $p = q$). Let r be a process, where $r \in \{p, q\}$. Consider a configuration in which all processes are in the Remainder Section. Suppose p runs solo and enters the Critical Section and then stops taking steps. Then process r runs and its `getAndSet()` operation at *tail* returns p's node. Process r repeatedly read the value of the *locked* field of p's node from q's memory segment until process p releases the lock. Therefore, r can incur an unbounded number of RMRs.

For that reason, the CLH lock shows poor performance on cache-less architectures [22].

`define` Node: struct
Boolean *locked*
Pointer *ptr*

Class `Lock`

shared: Node *tail*, *pred*
local: Node *myPred*, *myNode*

Method `lock()`

```
1 myNode.locked:= true                    /* sets the locked field of it's node*/
2 pred.write(tail.getAndSet(myNode))      /* appends at tail*/
3 myPred:= pred                           /* stores the predecessor in myPred*/
4 while myPred.locked do
5 end
```

Method `release()`

```
6 myNode.locked:= false                   /* sets its locked field to false*/
7 myNode:= myPred                         /* reuse the predecessor node*/
```

Figure 3.2: The CLH Lock Implementation

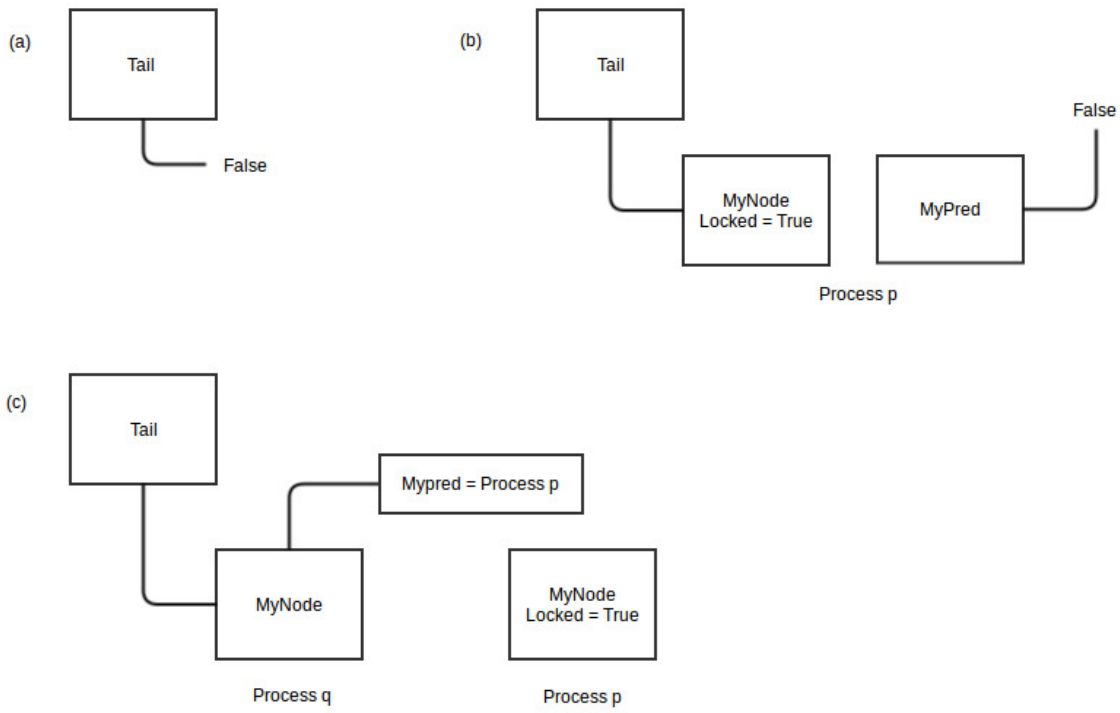


Figure 3.3: The CLH Lock: (a) Initially *tail* is false. (b) Process p calls the `lock()` method so p appends its node at the *tail*. Process p sets its *locked* field to *true* and gets the lock. (c) Now, process q comes in and performs `getAndSet()` at the *tail* and gets the reference of the predecessor node(i.e. process p). Then process q spins on the *locked* field of node p. Once process p releases the lock and sets its *locked* field to false, process q can get the lock.

3.3.1.2 MCS Queue Lock

Like the CLH lock, in the MCS lock (presented in Figure 3.4), a list of node objects is used. Each process has a node, *myNode*, of the same type *Node* as before, i.e. it has a boolean *locked* field and a *ptr* field. Processes which call the `lock()` method form a list. Unlike in the CLH lock, the *ptr* pointer now stores the reference to the successor of the node. The list has a *tail* pointer which points to the last node of the list. If *tail* is null, then the list is empty. If the *locked* field of a node is true, then the process who owns that node is waiting for the lock or currently holds the lock. Otherwise, it has released the lock.

In the `lock()` method, in line 1 of Figure 3.4, each process appends its node at the tail of the list by performing a `getAndSet()` operation with its own node as the argument and writes its return value of to the *pred* field. Then in line 2, each process checks whether it has a

predecessor by reading the *pred*. If *pred* is null, then process p is the first process which has called the `lock()` method and it gets the lock instantly. Otherwise it sets the *locked* field of its node to true in line 3 and then sets the *ptr* field of the predecessor node to its own node in line 4. Process p spins on the *locked* field of $myNode$ in line 6 until the field is set to false by the predecessor process in line 15.

In the `release()` method, a process p reads the *ptr* field of $myNode$ in line 9. Then it performs `compareAndSwap(p,null)` at the *tail*. If `compareAndSwap(p,null)` succeeds, then there is no other process contending for the lock and *tail* is set to null and p returns. Otherwise, there is a *slow* process, i.e. a process which has finished its `getAndSet()` operation of its node at the *tail* but has not written the return value of `getAndSet()` operation to *pred*. In that case, p waits until the slow process has finished the above operation in line 13. Once *ptr* is not null, p sets the *locked* field of its successor to false indicating that the lock is free, and then it returns.

Unlike in the CLH lock, the `release()` method of the MCS lock is not wait-free.

Figure 3.5 shows an example of the MCS lock.

The MCS Lock satisfies mutual exclusion and has O(1) RMR complexity

Suppose at time t, two distinct processes p and q, are in the Critical Section. Each process must have performed a `getAndSet()` operation in line 1 and checked its *pred* in line 2. Assume that process q executes its `getAndSet()` in line 1 at time t' after p does. As per the structure of the list (see Section 3.3.1.1), the process which performs the `getAndSet()` operation first enters the Critical Section first. So, process q cannot be in the Critical Section at the same time as p. This proves that at most one process can be in the Critical Section at any time.

Expected RMR Complexity of the MCS lock

In the CC model, the RMR complexity of the MCS lock is O(1): Each process first gets a cached copy of the *locked* field of its own node and then repeatedly reads the cached copy

and incurs only "cache hits" until the *locked* field changes to false. Process p incurs a "cache miss" when the process that own the predecessor node releases the lock and invalidates the value of the *locked* field of p's node in p's cache. When p reads the *locked* field of its node as false, it enters the Critical Section. Hence, process p incurs only a constant number of RMRs in the `lock()` method. In the `release()` method, a process p incurs a constant number of RMRs if the `compareAndSwap()` in line 12 succeeds. Otherwise, there is a slow process and p waits it to complete its operation in line 1. While waiting, process p repeatedly reads the cached copy of the *ptr* field of its own node and incurs "cache hits" only. As soon as the slow process completes its operation in line 1, p incurs a "cache miss". So, p incurs a constant number of RMRs in the `release()` method. Therefore, in the MCS mutual exclusion algorithm, a process incurs only a constant number of RMRs.

In the DSM model, the RMR complexity is O(1): Each process has its own *locked* field in its local memory segment. The process p busy-waits on the *locked* field in its local memory segment. Process p waits until the process which owns its predecessor node sets the *locked* field in p's local memory to false. This indicates that p can enter the Critical Section. Therefore, a process incurs a constant number of RMRs in the `lock()` method. In the `release()` method, if the `compareAndSwap()` in line 12 succeeds then a process p incurs a constant number of RMRs. Otherwise process p waits for a slow process to complete its append operation. While waiting, p repeatedly reads the *ptr* field of its own node. So, p incurs a constant number of RMRs in the `release()` method. Therefore, overall in the MCS mutual exclusion algorithm, a process incurs only a constant number of RMRs.

`define` Node: struct
Boolean *locked*
Pointer *ptr*

Class `Lock`

shared: Node *tail*
local: Node *myNode*, *pred*

Method `lock()`

```
1  pred.write(tail.getAndSet(myNode))     /* appends its node to tail*/
2  if pred ≠ null then
3  |  myNode.locked.write(true)
4  |  pred.ptr.set(myNode)
5  end
6  while myNode.locked do
7  |  /* waits until the lock is released*/
8  end
```

Method `release()`

```
9  if myNode.ptr = null then
10 |  if tail.compareAndSwap(myNode,null) then
11 |  |  return
12 |  end
13 |  await(myNode.ptr.read()= null)/* waits until the successor fills the ptr
   |  field*/
14 end
15 myNode.ptr.locked.write(false)
16 myNode.ptr.write(null)
```

Figure 3.4: The MCS Lock Implementation

3.3.1.3 Test and Set Lock (TS)

The TS lock is presented in Figure 3.6. In this algorithm, a test and set object (see Section 2.2.1), *tas*, stores a Boolean value, which is initially false. Each process executes a `test&Set` instruction on *tas* to change the value from false to true. If *tas*.`test&Set` succeeds, the process enters the Critical Section, otherwise it repeatedly executes the `test&Set` operation until it succeeds. In the release() method, the process resets the `test&Set` object *tas*.

The TS lock satisfies mutual exclusion

Suppose at time t, two processes, p and q, are in the Critical Section. Assume that process

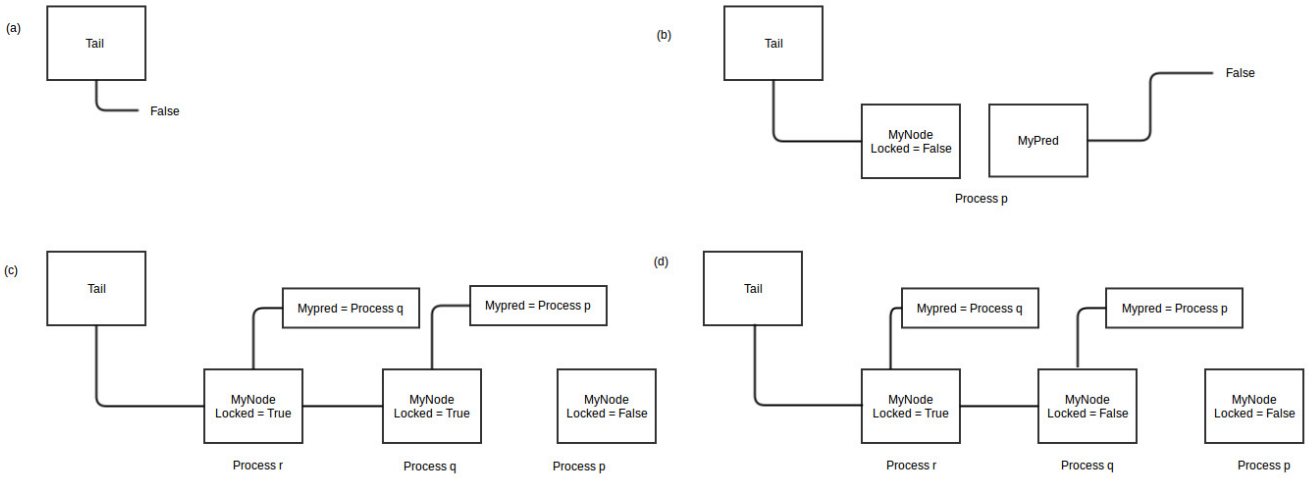


Figure 3.5: The MCS Lock: (a) Initially *tail* is false. (b) Process p in order to acquire the node, it places the node at the *tail*. As process p has no predecessor so it enters the Critical Section. (c) Process q and r enqueue their nodes at the *tail* of the list and gets the reference of predecessor node. Process q sets the ptr pointer of p to point to its own node and process r has q as predecessor. While p still holds the lock, process q spins on its locked field. (d) Process p while releasing the lock sets the locked field of its successor to false and process q gets the lock.

q executes its `test&Set` instruction in line 1 of the `lock()` method at time t' after p does. Then during [t',t], *tas* reads true. Process q cannot finish the while loop in line 1 as *tas* reads true. It spins in the while loop until process p sets to false in line 2. This proves that at most one process can be in the Critical Section at a time.

Class `Lock`

shared: `test&Set` *tas*

Method `lock()`

1 `await`(*tas*.`test&Set` ()) /* waits until the `test&Set` is reset*/

Method `release()`

2 *tas*.`reset`()

Figure 3.6: `test&Set` Lock Implementation

3.3.1.4 Test and Test and Set Lock (TTS)

Class `Lock`

shared: `test&Set` *tas*

Method `lock()`

```
1 await(tas.read(true))
2  tas.test&Set ()
```

Method `release()`

```
3 tas.reset()
```

Figure 3.7: Test and Test and Set Lock Implementation

The *tas* object used in the TTS lock is same as the one used in the TS lock. Unlike the TS lock, in the TTS locks, each process spins by reading the value of *tas*. In the `lock()` method, each process repeatedly reads the value of *tas* until it reads false. If a process reads *tas* as true, then some other process holds the lock; if *tas* reads false, then the lock is free. After a process reads *tas* as false, it performs `test&Set` in line 2 and tries to set the value of *tas* to true. If the `test&Set` operation succeeds, it enters the Critical Section, otherwise it repeatedly reads the value of *tas* again.

In the `release()` method, a process resets the *tas* object.

Expected RMR complexity of the TS and the TTS lock

In the CC model, the RMR complexity of the TS lock is unbounded: Each process p, spinning on the *tas* object performs a `test&Set` operations repeatedly which forces p to discard its cached copies. This causes the process p to incur an RMR each time. Process p might have to wait for an unbounded amount of time to acquire the lock. So, the RMR complexity of the lock() method is unbounded.

In the DSM model, the RMR complexity is also unbounded: Suppose the *tas* object is in q's memory segment. If process $p \neq q$, busy-waits in line 1 in Figure 3.6, it repeatedly performs a `test&Set` operation on *tas* in q's memory segment. As the process which owns the lock can be in the Critical Section for an arbitrary amount of time, process p may incur

an unbounded number of RMRs.

In the CC model, the RMR complexity of the TTS lock is unbounded: It is possible that every time process p performs a `test&Set` operation, it fails and some other process wins. So, in this case, p has to repeatedly perform `test&Set` operations until it enters the Critical Section. Thus, it may incur an unbounded number of RMRs.

In the DSM model, the RMR complexity is also unbounded: Suppose the *tas* object is in q's memory segment. If process $p \neq q$ busy-waits in line 1 in Figure 3.6, it repeatedly reads the value of *tas* from q's memory segment. Hence, each time p reads the value of *tas* from q's memory segment, it incurs a remote memory reference. As the process which owns the lock can be in the Critical Section for an arbitrary amount of time, process p may incur an unbounded number of RMRs.

Both, the Test and Set lock and the Test and Test And Set lock, provide deadlock freedom. This follows from the properties of `test&Set` object. If some processes try to perform a `test&Set` operation, on `test&Set` object when it reads false, then the first process which performs `test&Set` operation on it will win.

Performance of TS and TTS locks

The TAS lock performs well if contention is small. On architectures, where `test&Set` instructions and the normal memory references use the same bus, the traffic on the bus caused by the spinning processes can slow the access to other locations by the lock owner. In the release() method, a process releasing the lock might be delayed because of the same reason [4].

3.3.1.5 The Tree Lock

This mutual exclusion algorithm was presented by Yang and Anderson [3]. The algorithm uses Peterson's lock objects (see Page 11) and has O($\log n$) expected *RMR* complexity for CC and DSM models. The algorithm is presented in Figure 3.8.

Data Structure

`YALock` uses an arbitration tree which the processes climb in order to enter the Critical Section. The tree used in the `YALock` has n leaves and is of height $\lceil \log n \rceil$. Each process p is associated with a unique leaf and from there it climbs towards the root node. Each internal node of the tree has a Peterson's lock object *Lock*. A process p captures the node by winning the Peterson's lock associated to that node. The process that wins the Peterson's lock at a node wins that node and moves one level up, otherwise it waits on the `Lock` object of that node until the lock on that node is released. This continues until the root node. Once a process captures the root node, it can enter the Critical Section. In the release() method, p follows the same path from the root to the leaf node and releases all locks on each node it captured by calling the `release()` method of Peterson's lock.

Expected RMR Complexity of lock()

In the DSM model, the RMR complexity is unbounded. This follows from the RMR complexity of the Peterson's Lock (see Page 11).

In the CC model, the RMR complexity of this lock is O($\log n$). Since the height of the tree T is $\lceil \log n \rceil$, a process has to capture at most $\lceil \log n \rceil$ locks to enter the Critical Section. Each process incurs constant number of RMRs to capture each node (this follows from the proof of Peterson lock. See Page 11). Therefore, the expected RMR complexity of the `lock()` method is O($\log n$).

The YALock satisfies mutual exclusion.

The mutual exclusion property of `YALock` follows from the properties of Peterson's Lock. Suppose at time t, both processes p and q are in the Critical Section. Each process must

Class `YALock`

```
define Node:
Peterson lock object: PObj
shared:
root: Node                          /* root of the arbitration tree*/
leaf: array[0,...,N-1] of type Node
local:
v: Node
```

Method `lock()`

```
1  v :=leaf[p]
2  repeat
3  |   v :=parent(v)
4  |   v.Pobj.Lock()
5  until v=root
```

Method `release()`

```
6  foreach Node v on the path from root node to leaf node do
7  |   v.PObj.release()
8  end
```

Figure 3.8: YALock: Yang and Anderson Mutual Exclusion Lock Implementation

have called the `lock()` method of Peterson's Lock at *root* node in line 4. Peterson's Lock gaurentees that if two processes call Peterson's `lock()` on the root, only one process will succeed. So, there will be only one winner at the root node which will enter the Critical Section.

3.3.1.6 Randomized Mutual Exclusion with O($\log n$ / $\log \log n$) RMRs

A randomized mutual exclusion algorithm presented by Hendler and Woelfel [13] uses CAS objects and read-write registers. The algorithm achieves sub-logarithmic expected RMR complexity for the CC and the DSM model, against the strong adaptive adversary. Assume w.l.o.g that $n = \triangle^{\triangle-1}$ for some positive integer $\triangle$. Then it follows that $\triangle = \theta(\log n \,/\, \log \log n)$. The randomized mutual exclusion algorithm is presented in the Figure 3.9.

Data Structure

Similar to other mutual exclusion algorithms [1, 23, 30], this randomized mutual exclusion algorithm uses an arbitration tree. In order to enter the Critical Section, a process climbs up the tree. The arbitration tree used in this algorithm is a complete $\triangle$-ary tree of height $\triangle$ and with n leaves. The leaves have height 0 and the root has height $\triangle$. A node is said to be at level i if its height is i. Each process p is associated with a unique leaf node, $leaf_p$. Each internal node of the arbitration tree T consist of a CAS object *Lock*. Golab, Hadzilacos, Hendler and Woelfel [34] presented a paper on Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations in which they implemented CAS using only a constant number of RMRs in both the CC and the DSM models. By using this CAS, an algorithm with asymptotic RMR complexity [13] is achieved. In order to win a node v, process p performs `v.Lock.CAS(`$\perp$`,`p`)`. If the CAS operation on v succeedes, then p captures that node. If p fails to capture the node, i.e., `v.Lock.CAS(`$\perp$`,`p`)` returns false, then p spins on `v.Lock` until the CAS object is set to $\perp$ by a process which owns v. Once `v.Lock` is released, process p tries to capture it again and this continues until it captures the root node. When p wins the lock on the root node, it enters the Critical Section.

Suppose that p follows $path_p$ from $leaf_p$ to the root. If there is no contention then a constant number of RMRs is incurred at each node, as process p will win the lock on each node on $path_p$ without any competitor. In this case, the total number of RMRs incurred is

O($\log n$ / $\log \log n$). In the `release()` method, p will release all the locks (acquired by it while going up) on the same path in reverse order, i.e., from the root node to $leaf_p$.

Randomized Promotion

It may be possible that every time a process p tries to acquire the lock on a node, some other process captures it first. The authors provide a randomized mechanism which gives a $1/\triangle$ chance to each process to be promoted for every c failed attempts to capture the lock, where c is some large enough constant. A randomized promotion is performed by each exiting process before releasing the lock at each node. The randomized mechanism is as follows:

Each node has an array, $apply[0, \ldots, \triangle - 1]$. Before trying to capture the lock at node v, process p writes its ID to a unique position of the *apply* array. The unique position is the index of the child of v from which p ascended to v. Suppose that process q currently owns the *Lock* at node v. Before releasing the lock at node v, q performs a randomized promotion. Process q randomly chooses an index $i \in 0,\ldots,\triangle$-1 and checks the apply array position correponding to i, i.e., it checks `v.apply[`i`]`. If it finds a process p' at that location then q tries to promote p' by executing `v.apply[i].CAS(`p'`,`$\perp$`)`. The process p' repeatedly reads its apply array position to know whether it has been promoted. If `v.apply[i].CAS(`p'`,`$\perp$`)` operation succeeds then process p' is added by q to the shared sequential promotion queue, $PromoQ$. Each promoted process p' enqueued into the $PromoQ$, spins on the $notify[p']$ until $notify[p']$ is set to false by a process executing the `release()` method. This signals p' to enter the Critical Section. In the `release()` method, q checks the $PromoQ$. If the $PromoQ$ is empty then q releases the *Lock* on the root node, and otherwise the process at the head of the $PromoQ$, p', is signalled to enter the Critcal section. Process q signals p' by setting to $notify[p']$ to false. In addition to this, q performs `Lock.CAS(`q, p'`)` at the root node in order to hands over the root lock to p', so that p' owns that lock when it enters the Critical Section. A process waiting in the $PromoQ$ incurs at most a constant number of additional

RMRs until it enters the Critical Section (for details see paper [13]). When process p' is in the `release()` method, it further notifies the next process (if any) in *PromoQ* to enter the Crtitical Section, otherwise it releases the root lock. We conclude that each process in the *PromoQ* is guaranteed to enter the Critical Section.

Deterministic Promotion for Starvation Freedom

In the worst case, it is possible that every time a process p' has a chance of being promoted, the promoting process q makes a wrong decision and p' never gets promoted. To deal with this, a deterministic promotion approach is used. Each node v in the arbitration tree has a sequential modulo-$\triangle$ register, `v.token`. This register is increased only by an exiting process. When an exiting process q releases the lock at node v, it performs a deterministic promotion. Process q reads the index `j=v.token` and then promotes the process (if any) at `v.apply[j]`. Process q then increments the register, `v.token`. This approach guarantees that if $\triangle$ promotion events occur at node v, while p' has applied to apply array, then p' is guaranteed to be promoted. In the worst case, at every level of T a process may incur $O(\triangle)$ RMRs. So, the worst case RMR complexity of this algorithm is $O(\triangle^2)$.

Expected RMR Complexity of the lock()

In randomized promotion, for every c failed attempts to capture the lock at node v, $c = O(1)$, each process which is registered in the *apply* array (having size $\triangle$), has at least one chance of being promoted and with probability at least $1/\triangle$ it succeedes. The number of attempts to capture one lock is geometrically distributed and thus it is $O(\triangle)$ in expectation. Once promoted, a process incurs at most $O(1)$ additional RMRs. As the height of the arbitration tree is $\triangle$, each process p has to capture locks on its path to enter the Critical Section. This leads to an expected RMR complexity of $O(\triangle)$. In the `release()` method, each process p incurs $O(\triangle)$ RMRs.

Bounding the Worst case RMR complexity

To bound the worst case RMR complexity to $O(\log n / \log \log n)$, a $\triangle$-process deterministic

mutual exclusion object is used at every node v. The object $v.MX$, can be called with the process unique IDs and provides methods `Getlock()` and `Rellock()`. Both these methods have worst-case RMR complexity of $O(\log \triangle)$. The MX object can be implemented using Yang and Anderson's algorithm [1]. Third method of the MX object, `LockOwner()` returns the ID of the current owner of $v.MX$.

Each process p keeps track of the attempts to capture a node v by counting the number of RMRs it has made. If it has incurred more than $O(\log \triangle)$ RMRs, then it calls $v.MX.getlock_i()$ where i is the rank of the child from which p ascended to v. If p succeeds in capturing `v.MX`, it makes only two more attempts to capture `v.Lock`. The process exiting the Critical Section when releasing the lock at v performs randomized as well as deterministic promotion and also promotes the owner of `v.MX` to the $PromoQ$. To identify the owner of `v.MX`, method `v.MX.LockOwner()` is used.

If process p captures `v.MX`, it will either win `v.lock()` in its next two attempts or it will be promoted. Therefore, the worst case RMR complexity to capture the lock of a node or to be promoted is $O(\log \triangle)$. Since the process needs to capture $\triangle$ nodes in order to enter the Critical Section, the worst case RMR complexity is $O(\triangle \log \triangle) = O(\log n)$.

As the root of the arbiration tree is protected by the CAS object so only one process can be there in the Critical Section at a time. Thus, the Randomized mutual exclusion algorithm satisfies the mutual exclusion property. As full proof of correctness is long and difficult and is beyond the scope of this thesis so, only a brief description is provided.

3.3.2 Randomized Mutual Exclusion Against the Oblivious Adversary

An unpublished paper by Giakkoupis and Woelfel [16] proposed a randomized mutual exclusion algorithm that seems to perform well against an oblivious adversary. Recall that an oblivious adversary makes all the scheduling decisions before any process has flipped a coin. This algorithm is simple to implement and uses read-write registers and compareAndSwap objects. This algorithm is based on an implementation of Backpack and Weaklock objects which are defined below. This algorithm works for the CC model as well as the DSM model and is presented in the Figure 3.12.

The Backpack Object

In this algorithm, an array of Backpack objects is used and each process has a Backpack (i.e., each process owns a Backpack). Each Backpack supports the operations `open()`, `enter()` and `close()`. Each Backpack has a unique owner and each process can open its Backpack by calling `open()`. A process is said to have entered a Backpack, if the `enter()` method returns true. Once a process p enters q's Backpack, then when q calls the `close()` method it returns the ID of p. Only one process can enter q's Backpack between the `open()` and the `close()` method calls. The operations supported by the Backpack object are explained below:

Open(): Only the owner of the Backpack can call this method. This method returns no value. Each process opens its Backpack by performing `CAS(∅, 0)`.

Close(): Only the owner of the Backpack can call this method. This method returns $\perp$, if no process has entered the Backpack otherwise it returns the ID of the process which is found in its Backpack. Each process closes its Backpack by performing `CAS(0,∅)`.

Enter(): A process call this method when it has to enter some other processes backpack. No process can enter its own backpack. This method returns a Boolean value. To enter p's backpack, q performs `p.CAS(0,`q`)`. If this compareAndSwap succeedes i.e. returns true then it implies that q has successfully entered p's backpack otherwise it returns false i.e. it failed

to enter p's Backpack. A Backpack implementation is presented in the Figure 3.13.

The Weaklock Object

A weaklock provides two methods;

- The `try_lock()` returns a Boolean value. In the `try_lock()`, `test&Set` object and a shared register *Count* (initialized to 0) is used.
- The `release()` does not return anything. Each process resets the `test&Set` object in the `release()` method.

A process p can try to win the weaklock by executing `try_lock()`. If `try_lock()` returns true then p successfully won the Weaklock, otherwise it failed. At most one process can own the weaklock at a time. A process which owns the lock has to release it eventually by calling `release()`. A Weaklock object presented in the Figure 3.11 is deadlock free and has O(1) RMR complexity (see Section 3.3.2).

Description of the Weaklock

In the Weaklock implementation, each process p reads *Count* into c, followed by a `test&Set` operation in line 2. A process p wins the Weaklock if the `test&Set` operation succeeds, i.e., p successfully sets T to 1, otherwise it fails. Process p can only complete the while loop in line 3 of `try_lock()` if T has been reset by some other process that currently owns the Weaklock or the counter *Count* has been increased to a value two larger than what p read from *Count*. In the `release()` method, each process resets the `test&Set` object and increments *Count* by 1.

The Weaklock object is starvation-free and has O(1) RMR complexity

Each process busy waits in line 3 of `try_lock()` until it reads a value of *Count* is two larger than what it earlier read in line 1 or it reads `test&Set` object $T = 0$. A process whose `test&Set` operation succeeded will eventually reset the `test&Set` object in the `release()` method. Each process also increments *Count* by 1 in the `release()` method.

In the CC model, the Weaklock object has O(1) RMR complexity: Each process p first gets

the cached copy of the *Count* field and the `test&Set` object T and then each process p repeatedly reads the cached copy of *Count* and T until it reads a value of *Count* that is two larger than what it earlier read or it reads $T = 0$. When the value of *Count* or T changes, p incurs a "cache miss" and accesses remote memory for new value of *Count*. As stated above, each process increments the *Count* variable by 1 and resets T in the `release()` method, so, p incurs only a constant number of RMRs in the `try_lock()` (at most 3 RMRs).

Description of the Randomized Mutual Exclusion Algorithm

The data structures used in this algorithm are: An array of `B[0,...,n-1]` Backpack objects and each process has its own Backpack. A *spin* array which is used to notify processes whether they can enter the Critical Section. A shared sequential queue, *PromoQ*, used for promoting processes into the Critical Section, and a Weaklock object, W.

Each process p sets `spin[p]` to true in line 1. In line 3, p opens its Backpack by executing `B[`p`].open()`. In this algorithm, p can either try to enter some other process's Backpack (note that no process can enter its own Backpack) or writes its own ID into the shared register, S. Which of the above a process performs is decided by the random coin flip. If the coin flip shows '0', then p writes its ID into S and then tries to win the Weaklock. Otherwise, p reads the ID in S, say q, into the local register, s. It then checks if `s`$\neq$ `myId` and `s`$\neq \emptyset$. If the above conditions hold then p tries to enter q's Backpack. Process p tries to enter q's Backpack by executing `B[q].enter()`. If p is successful in entering q's Backpack then it spins on `Spin[p]` in line 11 until `Spin[p]` is set to false by a process executing the `release()` method. If p fails in entering q's BackPack, then p tries to win the Weaklock in line 16. If p succeeds in winning the Weaklock (i.e. `try_lock()` returns true) then it enters the Critical Section. Otherwise p closes its Backpack in line 17 by executing `B[`p`].close()`. If p does not find any other process in its Backpack, then it starts the loop in line 2 again. Otherwise it enqueues the process found in its Backpack, say r, to the local queue and keeps executing the loop in line 2 until it gets the lock or succeeds in entering some other process's

Backpack.

In the `release()` method, p closes its Backpack in line 21 and if it finds any process in its Backpack, it adds that process in the local queue. Finally, p empties its local queue into the shared promotion queue, $PromoQ$ in line 23. Then p checks the $PromoQ$ in line 26. If the $PromoQ$ is empty, then p releases the Weaklock otherwise it signals the first process in the $PromoQ$, say q, to enter the Critical Section by setting the `spin[q]` to false.

The Backpack lock satisfies mutual exclusion

A process enters the Critical Section by either winning the Weaklock, W or by entering some other process Backpack. A process p can return from the `lock()` by executing line 12 or line 16 or line 19 in the Figure 3.12. In line 16, a process p tries to win W. As discussed above in the Weaklock, the `try_lock()` guarantees that at most one process wins W. So, no two processes can complete the `try_lock()` method and enter the Critical Section.

Suppose at time T, two distinct processes p and q, are in the Critical Section. Each process must have tried to win the Weaklock. As we have already stated above that in the Weaklock there will be only one winner so, both p and q cannot win Weaklock. Assume that p wins the Weaklock and q enters its Backpack. Process q cannot finish its while loop in line 11 until its spin is set to false in line 30 by a process executing the `release()` method. So, no two processes can be in the Crtitical Section at the same time.

Note that if the $PromoQ$ is not empty, then there will be at most one process whose spin is false.

3.3.3 Adaptive Mutual Exclusion Algorithms

To measure the performance of the adaptive mutual exclusion algorithms, two notions of contention are considered. They are point and interval contention [2]. Consider a history H.

- Point contention over H is the number of processes that are active at same time in H .
- Interval contention over H is the number of processes active during the entire history H, i.e., execute outside of their non-critcal sections of H.

An algorithnm is adaptive when its time complexity of an algorithm depends on the number of processes calling the `lock()` method i.e. the contention. There are various well known adaptive mutual exclusion algorithms such as Adaptive mutual exclusion algorithm by Attiya and Bortnikov [7], Adaptive Bakery algorithm by Taubfled [32] etc. The adaptive mutual exclusion algorithms are not discussed in detail as they are out of the scope of this thesis.

Class `HWLock`

define Node: struct
`Lock`: **int init** $\perp$
`MX`: Starvation free $\triangle$-process mutual exclusion object
`apply`: array[0,...,$\triangle$-1]
`token`:**int init** 0
shared:
`root`: Node /* root of arbitration tree*/
`PromoQ`: sequential queue init $\emptyset$ /* Promotion queue*/
`leaf`: array[0...n-1] of type `Node`
`notify`: array[0...n-1] of type **boolean init false**
local:
v: Node
i, j, j', tok, ctr: int

Method `lock()`

```
1  notify[p] = false
2   v=leaf[p]
3   repeat
4   |  Let i be an integer such that v is the (i+1)-th child of parent v
5   |  v = parent(v)
6   |  v.apply[i].CAS(⊥,p)
7   |  ctr=0
8   |  repeat
9   |  |  ctr= ctr+1
10  |  |  if (ctr > ⌈log(△)⌉) then
11  |  |  |  if v.apply[i].CAS(p,⊥) then
12  |  |  |  |  v.MX.Getlock_i()
13  |  |  |  |  v.apply[i].CAS(⊥,p)
14  |  |  |  |  await(v.Lock = ⊥ ∨ v.apply[i] ≠ p)
15  |  |  |  end
16  |  |  end
17  |  |  if ¬ v.Lock.CAS(⊥,p) then
18  |  |  |  tok=v.token
19  |  |  |  await(v.token ≠ tok ∨ v.apply[i] ≠ p ∨ v.Lock = ⊥)
20  |  |  end
21  |  |  if v.MX.LockOwner = i then
22  |  |  |  v.MX.Rellock()
23  |  |  end
24  |  until v.apply[i] ≠ p ∨ v.Lock = p
25  |  if ¬ v.apply[i].CAS(⊥,p) then
26  |  |  await(notify[p] = true)
27  |  end
28  until notify[p] = true ∨ v =root
```

Figure 3.9: Randomized mutual exclusion algorithm for process $p \in 1,\ldots,n$ [13]

Method `release()`

```
1  foreach Node v on the path from root node to leaf node do
2  |   tok = v.token
3  |   i = v.MX.lockOwner
4  |   Choose j' randomly from 1,...,△-1
5  |   for j ∈ (j', tok, i) - ⊥ do
6  |   |   q = v.apply[j]
7  |   |   if q ≠ ⊥ ∧ v.apply[j].CAS(q, ⊥) then
8  |   |   |   PromoQ.enq(q)
9  |   |   end
10 |   end
11 |   v.token = (tok+1) mod △
12 |   if v ≠ root then
13 |   |   v.Lock.CAS(p, ⊥)
14 |   end
15 end
16 if PromoQ = ∅ then
17 |   root.Lock.CAS(p, ⊥)
18 else
19 |   q = PromoQ.deq()
20 |   root.Lock.CAS(p,q)
21 |   notify[q]=true
22 end
```

Figure 3.10: Randomized mutual exclusion algorithm for process $p \in 1,\ldots,n$ [13]

Class `WeakLock`

shared: `test&Set` object T, register $Count = 0$

Method *try_lock*()

```
1  c := Count.read()
2  if T.test&Set ()=0 then return True
3  await(Count.read()> c + 2 or T.read()= 0)
4  return False
```

Method `release`

```
5  Count.write(c + 1)
6  T.reset()
```

Figure 3.11: WLock: Weak Lock Implementation

Class `Lock`

shared: register S, `BackPack` $B[0 \dots n-1]$, `WeakLock` W, sequential queue $PromoQ = \emptyset$, register $Spin[0 \dots n-1]$
local: queue $Q[0 \dots n-1] = (\emptyset \dots \emptyset)$, $s = \emptyset$

Method `lock()`

```
1  Spin[myID].write(True)
2  repeat
3  |  B[myID].open()
4  |  Choose coin ∈ 0, 1 uniformly at random
5  |  if coin = 0 then
6  |  |  S.write()
7  |  else
8  |  |  s := S.read()
9  |  |  if s ≠ myID  s ≠ ∅ then
10 |  |  |  if B[s].enter() then
11 |  |  |  |  await(Spin[myID].read()=False)
12 |  |  |  |  return
13 |  |  |  end
14 |  |  end
15 |  end
16 |  if W.try_lock() then return
17 |  q := B[myID].close()
18 |  if q ≠ ∅ then Q[myID].enq(q)
19 until W.try_lock()
20
```

Method `release()`

```
21  q := B[myID].close()
22 if q ≠ ∅ then PromoQ.enq(q)
23 while Q ≠ ∅ do
24 |  PromoQ.enq(Q.deq())
25 end
26 if PromoQ = ∅ then
27 |  W.release()
28 else
29 |  q := PromoQ.deq()
30 |  Spin[q].write(False)
31 end
```

Figure 3.12: General Randomized Lock Implementation

Class `BackPack`

shared: CAS object $C = \emptyset$;

Method `open()`

```
1  C.CAS(∅, 0)
```

Method `close()`

```
2 if C.CAS(0, ∅) then
3 |  return ∅
4 else
5 |  c := C.read()
6 |  C.CAS(c, ∅)
7 |  return c
8 end
```

Method `enter()`

```
9 if C.CAS(0, myID) then
10 |  return True
11 else
12 |  return False
13 end
```

Figure 3.13: BackPack Implementation

Chapter 4

Methodology

4.1 Introduction

This chapter details the methods used to compare the performance of different mutual exclusion algorithms. Section 4.2 discusses real-world cache-coherent shared memory multiprocessors. This section also focuses on the importance of cache-coherency in multiprocessors and provides an insight to different cache-coherence protocols. Section 4.3 discusses the data structures used for comparing the performance of mutual exclusion algorithms. The Java Memory Model and the atomic variables used in Java are discussed in Section 4.4. In Section 4.5, the method used for measuring the performance of the mutual exclusion algorithms both in isolation and in the data structures is described.

4.2 Real World Cache-Coherent Shared Memory Multiprocessor

Real world shared memory mutliprocessors provide increased computing speed in comparison to the single processor systems. This is due to increased parallelism. But in shared memory multiprocessor systems, memory contention (multiple requests from processes to access memory), communication contention (contention in the inter-connection network) and the latency time (time taken by a request to traverse the inter-connection network) tend to slow down the program execution time and increase the memory access time. In cache coherent shared memory multiprocessors, each processor has a private cache attached to it. For instance, when a process needs to access the value of variable x, then the process checks whether it has a cached copy of x. If it has a cached copy of x then it reads the value of x from the cache, otherwise, it retrieves the value of x from shared memory.

Some of the cache coherent multiprocessor systems are the Intel Xeon processors or Sun Microsystem's multiprocessors. Systems like RP3 from IBM, Cedar from University of Illinois, and Butterfly from BBN laboratories, contain about 100 processors connected to memory modules by a multi-stage inter-connection network (having considerable latency).

In shared memory systems, sharing of data structures and code among processes enables parallelism. But this parallelism can result in several copies of a shared data in one or more caches at the same time. To maintain the coherent view of memory and to make the data in the caches consistent, cache coherence protocols are used. There are several cache coherent protocols for general inter-connection networks [5], [11], [31] and for systems using a shared bus [15],[18]. Cache coherent protocols using the shared bus are discussed later in this section.

Quickpath and Hyper-Transport Interconnects

Earlier in the shared memory systems by Intel, the front side bus (also known as FSB) used to carry data between processor and memory controller. But since 2008, Intel replaced the front side bus with quickpath interconnect (QPI) [36]. In the modern multiprocessor systems, FSB was replaced because it is old and slow technology and may result as bottleneck in today's systems. QPI is a point-to-point interconnect which connects the processor to the I/O hub or one or more processors or I/O hubs. This allows all components to access each other directly via a network [36]. Since 2001, the multiprocessor systems by IBM and Apple, the front side bus is replaced by hyper-transport. The reason behind this is the same as described above, i.e. FSB is slow and has long latency time. Hyper-transport is a low latency point to point link which is used for interconnection of processors. Hyper-transport supports message passing, signaling interrupts and I/O transactions. It is used by IBM and Apple in the Power Mac65 and in modern MIPS systems [36]. The only difference between the two is the difference in architecture.

Protocols for Multiprocessors with a Shared bus

The shared bus protocol depends on the cache controller. The Cache controller observes bus transactions and appropriate actions are taken to maintain consistency of data. For instance, multiprocessors such as Firefly (designed by Digital Equipment Corporation), Dragon (designed by Xerox Palo research centre) use shared bus cache coherence protocols. Since companies like Sun microsystem, Intel, Apple etc. introduced modern shared memory multiprocessors, the above mention multiprocessors have lost their value.

Modern Shared Memory Multiprocessors

Among the modern multiprocessors, Sun microsystem's Ultra SPARC T1 microprocessor (also known as Sun Niagara) is one of the oldest and is a multi-threaded, multi-core processor. Sun microsystem's main goal while making Sun Niagara was to run as many concurrent threads as possible and to maximize the utilization of each core. The cores of this multiprocessor are less complex in comparison to high end processors. In the Sun Niagara, eight cores fit on the same chip and each core supports only one thread. Sun Niagara has a single floating point unit which is used by all eight cores and this makes it unsuitable for applications which include many floating point mathemetical computations [37].

Later, Ultra SPARC T2 (also known as Niagara 2) was introduced. Niagara 2 provides eight cores. Each core supports eight threads and one FPU (floating point unit). After SPARC T2, Sun microsystems designed SPARC T3, which provides 16 cores and a total of 128 threads. The cores in this multiprocessor doubled the memory capacity and quadrupled the I/O throughput in comparison to SPARC T2 [37].

Intel Xeon processors, in the E7 family, use hyper-threading technology. Hyper-threading technology delivers two threads per core which enhances the parallelism. The latest Xeon E7-8870 processor family by Intel supports 10 cores and 20 threads in total and 30 MB of cache memory. Write-back caching protocols (explained on Page 6) are used in all the Sun Microsystem's multiprocessors.

4.3 Concurrent Data Structures Used for Testing the Performance of Mutual Exclusion Algorithms

Concurrent data structures are those in which multiple processes can concurrently perform operations. There are different concurrent data structures such as concurrent queues, pools, linked lists, hash tables, search trees etc. In this thesis, we consider lock based concurrent linked lists and AVL trees for testing the performance of mutual exclusion algorithms. Two types of locking techniques are considered to test how different locks affect the performance of data structures. They are described as follows:

Coarse-grained locking: In coarse-gained locking, a process locks the entire data structure, performs the respective operation and then releases the lock on the data structure. The Coarse-grained locking guarantees that only one process at a time can perform an operation on a data structure. A Coarse-grained locking technique works well when the level of concurrency is low, but when the level of concurrency is high then processes may have to wait for a long time to perform their operations. In this thesis, coarse-grained locking is used to implement AVL trees. AVL trees are chosen because today, trees are very important data structure and also the search time in trees is less in comparison to other data structures like arrays, hash tables, queues, stacks etc.

Fine-grained locking: In this locking technique, instead of locking the entire data structure, a small part of the data structure is locked. In this thesis, fine-grained locking is used to implement a concurrent linked list. Each node of the linked list has a lock associated with it.

In the linked list, there are two types of nodes: sentinel nodes and regular nodes. The sentinel nodes are *head* and *tail*. Each regular nodes contains data in addition to a reference to the next node in the list.

Figure 4.1, represents a method used to add an item to the linked list. In the fine-grained strategy, each process locks the predecessor node and the current node where it has to per-

form an operation. Consider a ordered list of nodes. Suppose process p wants to insert value k in the list. Then p uses two local variables *curr* and *pred*. Process p first sets *pred* to `head` and sets *curr* to `head.next` in line 3 and line 4 of Figure 4.1. In line 6, p then compares *curr*'s key value to k. If the two key values match then it means that k is already in the list. If k is already in the list, then p returns false in line 13. Otherwise if $curr's key < k$ then p unlocks *pred* and set *pred* = *curr* in line 8 of Figure 4.1. Later, in line 9, p sets *curr* = *curr.next* and so on it continues until p finds where k is to be inserted. Once p finds a place where it has to insert in the list, it creates a new node in line 15. Later in line 16 and 17, it sets the *newnode.next* = *curr* followed by *pred.next* = *newnode* and then returns true from the method.

Figure 4.2 represents the method used to remove from the linked list. To delete a node from the linked list, process p first finds the node which it has to delete by following the same procedure as above. Similar to the `adItem()`, p uses variables *curr* and *pred*. Process p first sets *pred* to `head` and sets *curr* to `head.next` in line 3 and line 4 of Figure 4.2. In line 6, p then compares *curr*'s key value to k. If the two key values match then it sets *pred.next* = *curr.next* and returns true. Otherwise, it unlocks *pred* and set *pred* = *curr* in line 8 of Figure 4.2. Later, in line 9, p sets *curr* = *curr.next* and so on it continues until it finds the node which it has to delete.

In our thesis, look-ups are lock-free i.e. to find an item from the linked list, processes do not lock the nodes. Figure 4.3 represents the method used to find an item from the linked list. Each process p to find an item from the list, uses variables *curr* and *pred*. Firstly, process p sets *pred* to `head` and sets *curr* to `head.next` in line 3 and line 4 of Figure 4.3. It then reads *key* value of the *curr*, if the *key* value of *curr* matches with the item, then it returns otherwise it sets *pred* = *curr* and *curr* = *curr.next* and so on it continues until it finds an item.

4.4 The Java Memory Model

The Java memory model (JMM) describes the semantics of concurrent memory accesses [25]. Two main aspects of the Java memory model are synchronization and transformations which can be applied to program code. Synchronization means that once one process enters a synchronized block protected by a lock, no other process can enter a block protected by that lock until the first process exits the synchronized block. In the Java memory model, synchronized block is a term for the Critical Section. Another aspect of synchronization is that it ensures that memory writes by a process before or during a synchronized block are made visible to other threads. This is done after process has exited the synchronized block, and it releases the lock and updates the changes from cache to main memory.

The Java memory model also determines the transformations the compiler may apply to a program when producing bytecode (compact numeric code for efficient execution by a software interpreter) and the transformations that may be applied to bytecode when producing machine code (code which is directly executed by the processor). It also describes the optimizations that can be performed on the machine code to minimize the execution time [25].

4.4.1 Atomic Variables in Java

The word atomic refers to linearizable (to learn more about linearizability see this paper [29]). The Java.util.concurrent.atomic package provides atomic variable in Java. This package supports several classes like Atomic Integer, Atomic Boolean etc. These classes provide access to single variables of the corresponding type. An atomic instruction atomically accesses and possibly modifies one or more memory locations.

In this thesis, the classes Atomic Integer and Atomic Boolean are used. Some of the methods which the class Atomic Integer class provides are `get()`, `set(boolean/int new)`, `compareAndSwap(int expect, int update)`, `getAndIncrement()`, `getAndDecrement()` and

`getAndSet(int new)`. Table 4.1 represents Atomic Integer methods and the Table 4.2 represents the methods that the Atomic Boolean class supports. In this thesis, the

Table 4.1: Atomic Integer Methods

Method	Desription	Return value
get()	gets the current value	current value
set(int new)	sets to the given value	N/A
compareAndSwap(int expect, int update)	atomically sets to update if current value is equal to expect	true, if successful and false otherwise
getAndIncrement()	atomically increment by one the current value	the previous value
getAndDecrement()	atomically decrement by one the current value	the previous value
getAndSet(int new)	atomically sets to new and returns previous value	the previous value

Table 4.2: Atomic Boolean Methods

Method	Desription	Return value
get()	gets the current value	current value
set(boolean new)	unconditionally sets to new	N/A

`compareAndSwap(int expect, int update)` method of the Atomic Integer class atomically sets the value to `update` value if the current value is equal to `expect` value. The instructions of the Atomic Integer class, `getAndIncrement()` and `getAndDecrement()` increment and decrement the current value by one. The `getAndSet(int new)` operation sets the current value to `new` and returns the previous value. The `set(int new)` and `get()` methods of Atomic Integer and Atomic Boolean gets the current value and sets to `new` and returns the previous value.

4.5 Overview of Approach used for Measuring the Performance of Mutual Exclusion Algorithms

To compare the performance of different mutual exclusion algorithms, the algorithms are first implemented using the Java programming language (discussed in the Section 4.4). Two approaches that are considered for measuring the performance of algorithms are in isolation and in the implementations of data structures. Isolation means that every process repeatedly gets the lock and releases the lock. The Critical Section in this case is very small. The time taken by each process to get the lock (T_g) and release the lock (T_r) is recorded seperately. The total time (T_l) taken by each process is calculated by adding T_g and T_r. The processes get the lock, execute the Critical Section and release the lock k times where k is a parameter defined later in this thesis. In the another test case, we added a delay of m milliseconds in the Critical Section where m is parameter having different values. Each process after, acquiring the lock, sleeps for m milliseconds. The time taken by the processes to get the lock and release the lock is recorded and is used to analyze how the delay affects the performance of the mutual exclusion algorithms.

In another test case, we have considered two data structures that are implemented using different mutual exclusion algorithms as it is interesting to see how different locks affect the performance of these data structures. Two data structures considered which are Concurrent linked lists and AVL trees. For this test case, the locks are made re-entrant (discussed in Section 5.3) so that they can be used universally in all data structures. We wanted to create realistic lock implementations that can be used in any data structure. In our data structure implementations, the locks need not to be re-entrant. In this case, each process can perform insert or delete or look-up operation on the data structure. The time taken by processes to get the lock, perform the respective operation, and release the lock is recorded. Processes get the lock and release the lock k times where k is a parameter which we choose from several values.

In the Critical Section, there is a counter $count_{mx}$ of type integer, which each process executing the Critical Section, increments by 1 in the Critical Section and then decrements it when it has finished executing the Critical Section. This is a sanity check to test whether the mutual exclusion property holds for an algorithm.

4.5.1 Calculating the Number of RMRs in the Mutual Exclusion Algorithms

In this thesis, the mutual exclusion algorithms which are considered for performance comparison are the the MCS lock [26], the CLH lock [22], the Test and Set lock (TS) [4], the Test and Test and Set lock (TTS) [4], the deterministic mutual exclusion algorithm based on arbitration tree (tree lock) [1], the randomized mutual exclusion algorithm with $O(\log n / \log \log n)$ RMR complexity (RMX) [13], Java re-entrant lock (Java Lock), and the randomized mutual exclusion algorithm against oblivious adversary (Backpack) [16]. For measuring the performance of these mutual exclusion algorithms, experiments are conducted on a Cache Coherent machine. For approximately counting the RMRs each process incurs in mutual exclusion algorithms, a local variable *count* of integer type is taken in each mutual exclusion algorithm which is intialised to zero.

In the MCS lock, each process accesses the remote memory to set *tail* to its own node in line 1 of Figure 3.3.1.2. Then each process gets the cached copy of the *locked* field of its own node (for details see Section 3.3.1.2) and spins on it until it incurs a cache miss. When it incurs a cache miss, this invalidation can evict *locked* field of its own node from cache and it increments the *count* variable. In the `release()` method, each process gets the cached copy of *tail* on which it tries to perform `CAS()` operation in line 10. Here it increments *count* by 1. Then in line 13, if a process has to wait for its successor, it spins on cached copy of *next* field of its own node until it incurs a cache miss. When it incurs a cache miss, it increments *count*. So, by this method, when a process completes the execution of mutual exclusion algorithm, the value of *count* gives the estimated number of RMRs it incurred.

In the CLH lock, each process first accesses remote memory to set *tail* to its own node in

line 2 of Figure 3.3.1.2 and then gets the copy of *pred* to set *pred* to its own node. Here each process increments *count* after each access to the remote memory. Each process then gets a cached copy of the *locked* field of the predecessor node and spins on it until it incurs a cache miss. When a process incurs a cache miss, it increments *count* variable. After a process completes the execution of the CLH lock, the value of *count* gives the number of RMRs each process incurred.

In the TS lock, each process gets the cached copy of the *tas* variable and the *count* variable is incremented every time the process performs `test&Set` on *tas* while the process is busy waiting in line 1 of Figure 3.6. So, when a process completes the execution of TS lock, the value of *count* represents the value of RMRs incurred while in the TTS lock, first each process accesses the shared memory to get the cached copy of *tas* and the spins on it (in line 2 of Figure 3.7) until it incurs a cache miss. Each process increments *count* after it finishes the while loop in line 2.

In the Mutual Exclusion Algorithm using registers, each process increments *count* in the repeat loop in line 2 of Figure 3.8. A process then accesses shared memory to get the cached copies of variables like *victim* and $flag[]$ and spins in the while loop in line 3 of the Figure 3.1 and increments *count* after finishing it. The value of *count* variables gives the number of RMRs incurred by each process.

In the Randomized mutual exclusion algorithm by Giakkoupis and Woelfel, each process increments the *count* variable every time it executes the repeat loop in line 2 of Figure 3.12.

Class `FineGrainedList`

define Node: struct
`Lock`
shared:
Node *head*
local:
Node *pred*, *curr*
int *key*

Method `addItem(int` *item*`)`

```
1  key = item
2  head.lock()
3  pred = head
4  curr = pred.next
5  curr.lock()
6  while curr.key < key do
7  |   pred.unlock()
8  |   pred = curr
9  |   curr = curr.next
10 |   curr.lock()
11 end
12 if key = curr.key then
13 |   return False
14 else
15 |   Node node = new Node(item)
16 |   node.next = curr
17 |   pred.next = node
18 |   return True
19 end
20 curr.unlock()
21 pred.unlock()
```

Figure 4.1: Add method in Fine-Grained Locking on Linked Lists [20]

Method `removeItem(int` *item*`)`

```
1  key = item
2  head.lock()
3  pred = head
4  curr = pred.next
5  curr.lock()
6  while curr.key < key do
7  |   pred.unlock()
8  |   pred = curr
9  |   curr = curr.next
10 |   curr.lock()
11 end
12 if key = curr.key then
13 |   pred.next = curr.next  return True
14 else
15 |   return False
16 end
17 curr.unlock()
18 pred.unlock()
```

Figure 4.2: Remove method in Fine-Grained Locking on Linked Lists [20]

Method `findItem(int` *item*`)`

```
1  key = item
2  pred = head
3  curr = pred.next
4  while curr.key < key do
5  |   pred = curr
6  |   curr = curr.next
7  |   if key = curr.key then
8  |   |   return True
9  |   else
10 |   |   return False
11 |   end
12 end
```

Figure 4.3: Find item method in Fine-Grained Locking on Linked Lists [20]

Chapter 5

Design, Implementation And Results of Mutual Exclusion Algorithms

5.1 Introduction

This chapter details the methods used for measuring the performance of the mutual exclusion algorithms both in isolation and in the data structures for the different test cases. Section 5.2 describes the cache coherent shared memory multiprocessor on which the tests were performed. Section 5.3 discusses different test cases which were considered for measuring the performance of mutual exclusion algorithms. Section 5.3.3 describes results of the experiments.

5.2 Hardware and Software Configuration

The mutual exclusion algorithms were tested on an Intel R910 Poweredge server. The R910 is a 4 socket server with 8 cores per socket. Each core uses hyper-threading to schedule more than one process. This increases the parallelism in the system.

In R910, Xeon CPU X 7500 processors are used. The Intel Xeon 7500 series processors have two integrated memory controllers which manage the data flow to and from the main memory. Each Intel Xeon 7500 processor has 24 MB of cache memory and the cache line size is 64 byte. The R910 has 4 bi-directional Intel QuickPath interconnects. For this particular system, 4 GB of main memory is used. The clock speed of the Xeon processors is 2.3 GHz with a maximum turbo frequency of 2.7 GHz (i.e. the CPU's clock speed can be increased up to 2.7 GHz) [33]. It offers up to 1TB of double data rate type three (DDR3) memory (i.e. dynamic random access memory with high bandwidth interface) [35] and ten Peripheral

Component Interconnect (PCI) slots which connect several hardware devices. The R910 supports write-back caching.

The operating system on this machine is the Scientific Linux release 6.1. Scientific Linux is a Linux produced by Fermi National Accelerator Laboratory and the European Organization for Nuclear Research (CERN). It is an open source operating system based on Red Hat Enterprise Linux. Algorithms were implemented in Java using Open JDK Java enterprise version 1.6.020.

5.3 Detailed Explanation of the Approach Used to Determine the Performance of the Mutual Exclusion Algorithms

As discussed in Section 4.5, the performance of mutual exclusion algorithms is measured in isolation and in data structures (implemented based on different mutual exclusion algorithms). For measuring the performance in the above scenarios, locks are made re-entrant. A re-entrant lock allows a process to acquire the same lock that it is holding without causing deadlock. Consider a case in which process p gets the lock and executes the Critical Section which contains another `lock()` method call. In this case, if the locks are not made re-entrant then p would block itself as it would try to get the lock which it already holds. For making the locks re-entrant, each process has a local Integer (L) which is intialized to zero. When a process calls the `lock()` method, it increments L. It then checks the value of L. If L greater than 1, then the process returns from the `lock()` method (i.e. the process already has the lock) otherwise it executes the remaining `lock()` method (i.e. the process is trying to acquire the lock for the first time). In the `release()` method, each process decrements L and if the value of L is greater than zero, then it returns, otherwise it executes the `release()` method.

5.3.1 Experimental Setup

Recall that the mutual exclusion algorithms which are implemented in order to compare their performance are the Test and Set lock (TS) [4], the Test and Test and Set lock (TTS) [4], the MCS lock [26], the CLH lock [22], the randomized tree based mutual exclusion algorithm with $O(\log n / \log \log n)$ RMR complexity (RMX) [13], the deterministic mutual exclusion algorithm based on the arbitration tree (Tree lock) [1], Java re-entrant lock, and the randomized mutual exclusion algorithm against oblivious adversary (Backpack) [16]. As explained earlier, these algorithms were tested on the R910 server. The experiments were performed with up to 64 processes because adding more processes would not increase parallelism.

In isolation, time taken by each process to get the lock and release the lock is recorded seperately. The Critical Section is very small and each process takes approximately 0.0001 ms to execute it. The total time taken by a process is computed by adding time taken by it to get the lock and time taken to release the lock. The average and standard deviation of the total time is calculated.

Note that the time taken by processes to execute the lock is the time taken by it to get the lock and release the lock.

The mutual exclusion algorithms were executed between 10^5 and 13 x 10^5 times. Figure 5.1 shows the average time taken by 8 processes to excecute different locks k x 10^5 times, where k is some parameter. The x-axis in the graph shows the number of times the lock() and release() methods are called i.e., k x 10^5 and the y-axis shows the average time in milli-seconds (ms). The results show that when 8 processes execute different locks, the average time taken by processes to execute a particular lock when k is 10^5 is more than the average time taken by processes to execute the same lock when k, is greater than 10^5. The average time gradually decreases as k increases. The reason for this can be that, intially when the processes start to execute the lock, they take more time because each process has to intialize

the objects and variables used in the lock. This leads to a higher average time when k is small.

As k is further increased from 9 x 10^5 to 13 x 10^5, not much difference is seen in the average time. The tests were performed on up to 64 processes and the results were similar to the 8 process case (shown in Figure 5.1).

As discussed earlier, the average time taken by processes to execute different algorithms is higher when k is small and the reason for it is the time taken by processes to intialize objects and variables used in the algorithms. So, in an another test case, the top and bottom 10 percent of the recorded time was omitted. Figure 5.2 shows the average time taken by 8 processes to execute different locks when the top and bottom 10 percent of the values are omitted. The results show that 8 processes take almost the same time as when executing different locks k x 10^5 times.

Due to the reasons explained above, the number of iterations for which mutual exclusion algorithms were executed in isolation and in data structures was fixed to 1 million and the top and bottom 10 percent values of the distribution list are omitted. For all the graphs, the x-axis represents the total number of processes executing the mutual exclusion algorithm and the y-axis represents the elapsed time in milli-seconds.

5.3.2 General Hypothesis for Different Locking Strategies

In this Section, we describe the results we expect to see based on the previous work done in this area. As stated by Anderson [4], as the number of processes executing the TS and the TTS locks increases, a sudden increase is seen in the time taken by the processes to execute these locks. Trigonakis, David and Guerraoui presented a paper [12] in which they explained that the MCS and the CLH locks are most resilient to contention in comparison to other mutual exclusion algorithms such as the Array-based locks, the TS lock, the TTs lock and the HCLH lock. So, we expect to see similar behavior by the MCS and the CLH locks in our experiments.

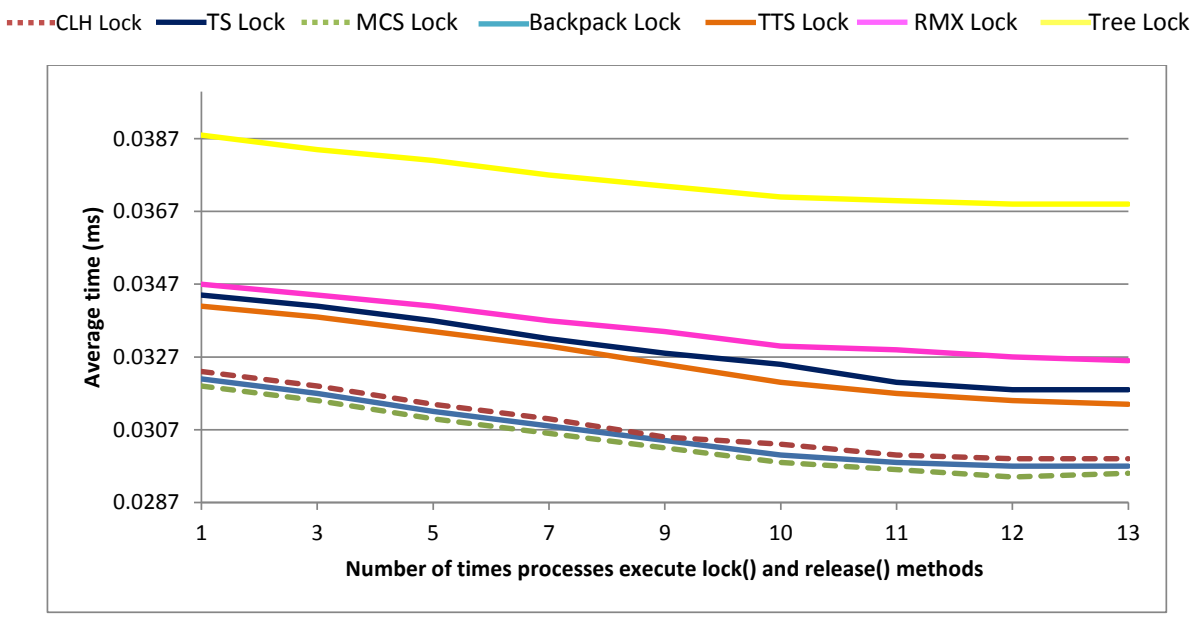


Figure 5.1: 8 Processes executing different Mutual Exclusion Algorithms k x 10^5 times

In the RMX lock, deterministic and randomized promotion strategies are used due to which processes may not always have to climb up to the root node to get the lock, while in the Tree lock, each process has to climb up to the root node to get the lock. So, we expect to see difference in the performance of the RMX lock and the Tree lock.

Figure 5.3 shows the number of times processes get promoted at different levels of the arbiration tree used in the RMX lock when executing the lock 10 x 10^5 times. For the Backpack lock, we expect to see constant RMR complexity but it is hard to predict without conducting experiments. It also shows the total number of levels of the tree used in the RMX lock and the Tree lock. In the result, we see that almost 65 percent of the processes gets promoted at different levels of the arbiration tree in the RMX lock.

5.3.3 Performance Analysis of the Mutual Exclusion Algorithms in Isolation

Figure 5.4 shows the performance of different mutual exclusion algorithms in isolation. The results in the Figure shows that the Java re-entrant lock shows best performance among all other locks. The TS and the TTS locks show good performance (after the Java re-entrant

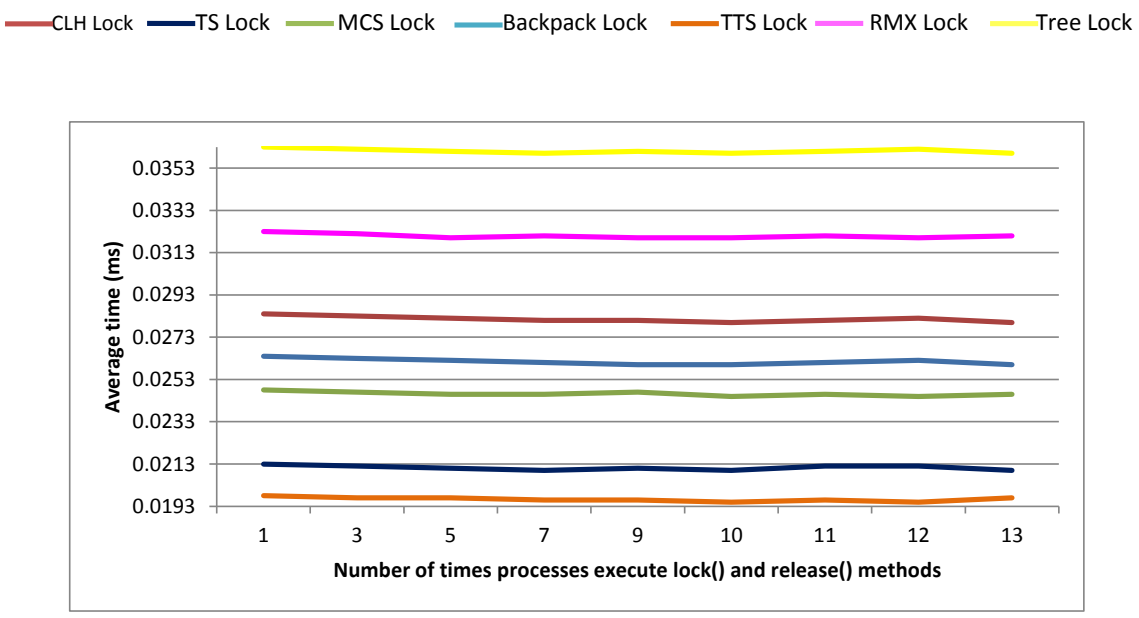


Figure 5.2: 8 Processes executing different Mutual Exclusion Algorithms k x 10^5 times when the top and bottom 10 percent of values are omitted

Number Of Processes	Number Of Levels in the Tree used in the RMX lock	Number Of Levels in the Tree used in the Tree lock	Number of promotions at Level 3	Number of promotions at Level 2
64	3	6	375572	313549
56	3	6	350483	327435
48	3	6	352891	313954
40	3	6	348963	319960

Figure 5.3: The number of times processes get promoted before reaching the root node in RMX lock when executing it 10 x 10^5 times

lock) upto 52 processes execute these locks. The MCS lock, the CLH lock and the Backpack lock follows the TS lock and the processes take almost the same time to execute these locks. Time taken by processes to execute the RMX lock and the Tree lock is more than other locks because in these locks, processes may have to climb to the root node to get the lock.

The reason of poor performance of the TS and the TTS lock when more than 52 processes execute them is the contention in the quickpath interconnect. In the TS lock, a process, busy waiting in the `lock()` method (see Figure 3.6), performs a `test&Set` operation on a shared variable. This `test&Set` operation forces all processes to discard their cached copies

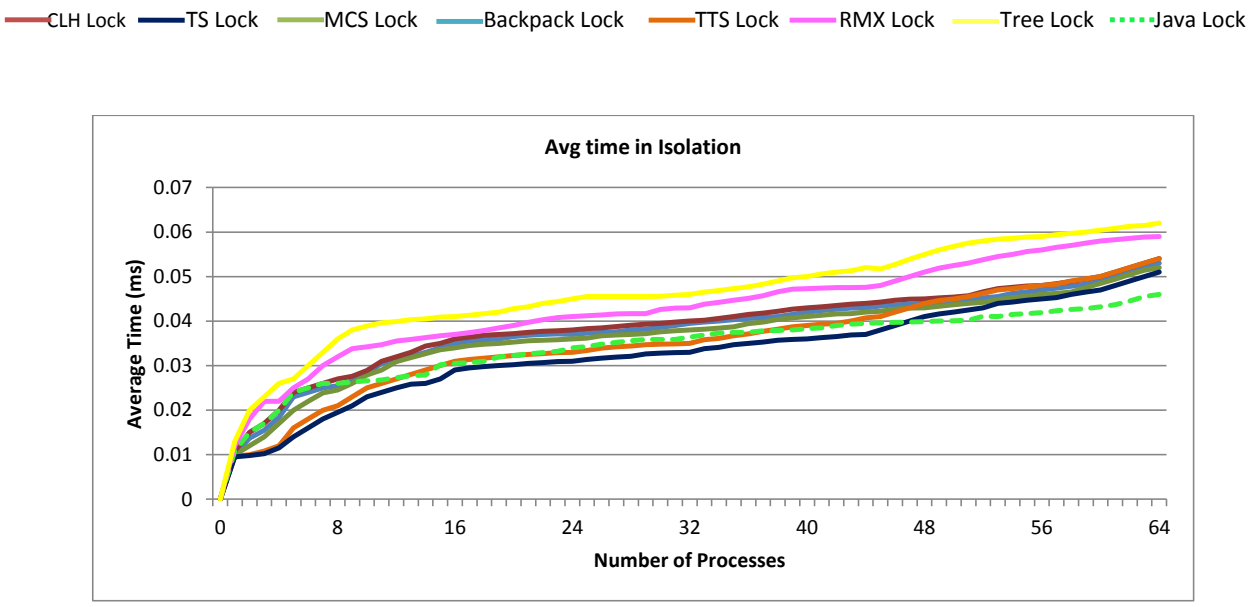


Figure 5.4: Processes executing different Mutual Exclusion Algorithms in Isolation

of the shared variable. Due to this, each process uses the quickpath interconnect to fetch the new value of the shared variable when it incurs a cache miss. This leads to contention of the quickpath interconnect. If the number of processes executing the lock is further increased, then the performance of the TS lock further deteriorates. The performance of the TTS lock is slightly better than the TS lock because in the TTS lock, each process, while busy waiting for the lock, re-reads the value of the cached copy of the shared variable (see Figure 3.7). Each process accesses the interconnect only when it incurs a "cache miss". So, the processes do not incur a "cache miss" every time (as in the case of the TS lock). In the TTS lock, all processes trying to acquire the lock must have incurred a cache miss almost at the same time, which may have caused a storm of traffic on the quickpath interconnect, hence leading to poor performance of the TTS lock.

In the Java re-entrant locks, when the contention is low, processes take more time to execute the lock method but as the contention increases, a steady increase is seen in the average time taken by processes to execute the lock.

The graph in the Figure 5.4 shows that there is only a small difference in performance of the

CLH lock and the MCS lock and that may be due to the system on which the experiments are conducted. As the R910 is a CC-NUMA machine, processes executing the CLH lock may have their predecessor node in the cache of some other processor. Each process will get the cached copy of the *locked* field of its predecessor node (for details see the algorithm in Figure 3.3) but it will cost an RMR. For instance, consider two processes p and q, where p is q's predecessor. Suppose that p and q are executing the mutual exclusion algorithm on two different cores. Now if q has to get the cached copy of the *locked* field of p's node, it has to access shared memory, hence causing an RMR. While in the MCS lock, each process spins on the *locked* field of its own node. So, the processes incur fewer RMRs in comparison to the CLH lock. Per passage, in the CLH lock, each process incurs 4 RMRs while in the MCS lock each process incurs only 3 RMRs. This may be the reason for the small difference in the performance of the CLH lock and the MCS lock.

The RMX lock and the Tree lock show better performance than the TS and the TTS lock because in the TS and the TTS lock, processes incur unbounded RMRs which results in more average time while in the RMX and the Tree lock, the number of RMRs incurred by processes are bounded. Performance of these locks is not as good as the MCS lock, the CLH lock and the Backpack lock and the reason behind this may be the number of RMRs incurred by processes. Each process incurs O(1) RMRs in the MCS and the CLH lock which are less than the RMRs incurred by each process in the RMX lock and the Tree lock. The graph shows that the RMX lock shows better performance in comparison to the Tree lock. This may be due to the deterministic and randomized promotion techniques (see the algorithm in the Figure 3.9) used in the RMX lock. In the RMX lock, processes may get promoted and may not have to climb all the way up to the root node to get the lock, while in the Tree lock, each process has to win all the locks on the path from its leaf node to the root node. This may be the reason why, the time taken by the processes to execute this lock is larger than the RMX lock. As explained earlier, Figure 5.3 shows the size of the tree in the

RMX lock and the tree lock and the number of promotions happening at different levels of the arbitration tree in the RMX lock (for details see Section 5.3.2).

In the Backpack lock, most of the processes either get the weak lock in line 3 of Figure 3.11 or enter some other process's backpack in line 10 of Figure 3.12. Therefore, processes execute only a single iteration of the repeat loop in line 2 of Figure 3.12 either get the weak lock or enter someone's back pack as seen in Figure 5.5. A more detailed explanation is provided in the next section.

5.3.4 How Adding Delay affects the Performance of Mutual Exclusion Algorithms

In this section, the length of the Critical Section is varied by adding a delay of d milli-seconds. This is done to see how varying the length of the Critical Section affects the performance of different mutual exclusion algorithms.

5.3.4.1 How Adding Delay affects the Performance of the Backpack Lock

In the Backpack lock, roughly half of the processes should either get the weak lock or enter some other process back pack in a single iteration. If no process is in the Critical Section of the Backpack lock, then a process completes the weak lock method call immediately. But if the Critical Section is large, then processes may have to execute the repeat loop in line 2 of Figure 3.12 more often to either get the weak lock or to enter some other process's back pack. To determine that almost half of the processes should get the weak lock or enter some other process back pack in each iteration, experiments were conducted with the Critical Section of different lengths. Each process, after acquiring the lock, sleeps for d milli-seconds before releasing the lock, where d is a parameter.

Figure 5.5 shows the number of iterations taken (the number of times the repeat loop in line 2 is executed) by processes to get the weak lock or to enter someone's back pack as a fuction of the delay. In the figure, x-axis represents delay added and the y-axis represents number of times the `lock()` and `release()` is executed. The vertical bars in the Figure represents

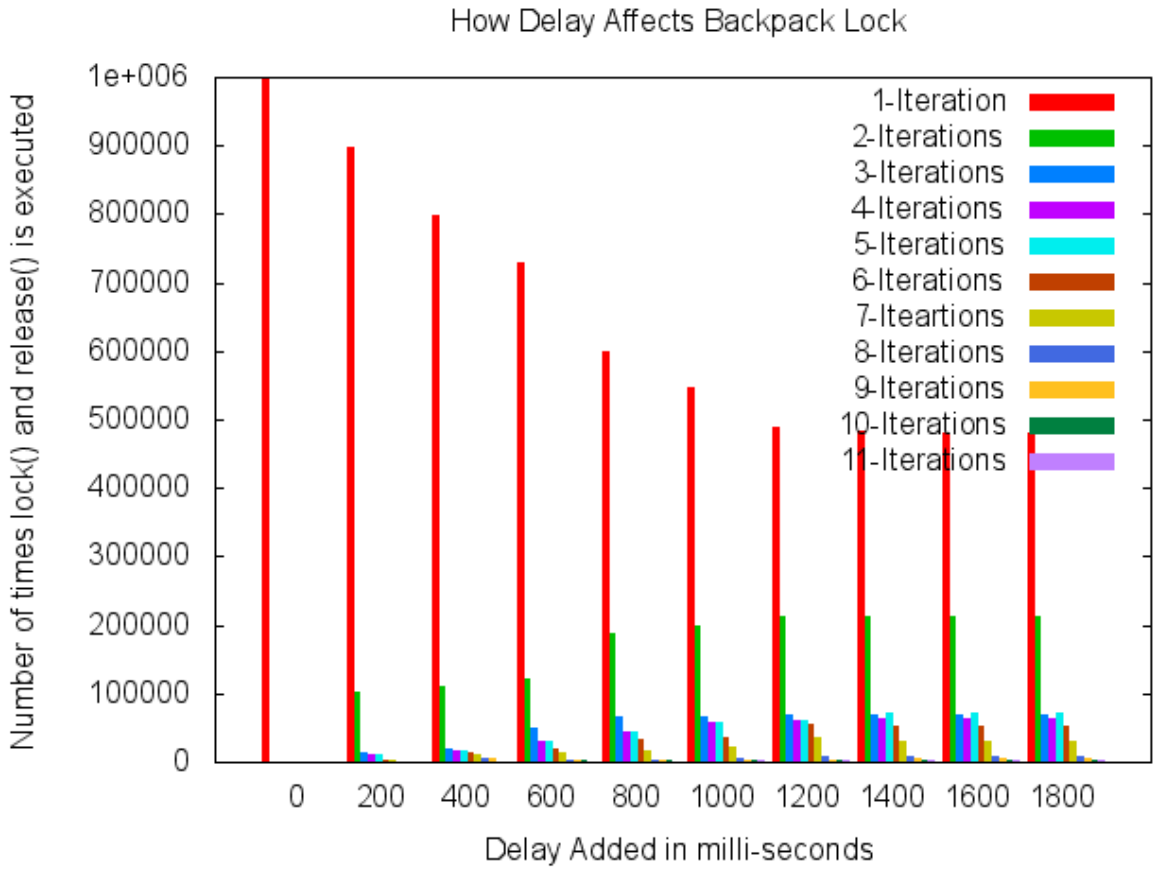


Figure 5.5: Performance of Randomized Mutual Exclusion against Oblivious Adversary when Delay is added

the number of iterations in which the processes either get the weak lock or enter someone's back pack. Figure 5.6 shows the number of iterations taken by 8 processes to either get the weaklock or enter someone's back pack in the tabular form. The number of processes that execute the algorithm is up to 64 and not much diffrence was seen in the results when compared to 8 process test case.

In the Backpack lock, when processes do not sleep in the Critical Section (i.e. the value of d is 0), almost all the processes either get the *weak lock* or enter someone's *back pack* in a single iteration of the repeat loop (in line 2 in Figure 3.12). As the value of d increases the number of processes that get the weak lock or enter someone's back pack in a each iteration decreases. Due to the delay inserted in the Critical Section, the processes have to wait either

Sleep Time in ms	1	2	3	4	5	6	7	8	9	10	11
0	999994	6	0	0	0	0	0	0	0	0	0
200	899138	101438	12697	10790	9785	3000	3100	52	0	0	0
400	798509	111189	20212	16198	15672	14299	10062	6754	4746	1300	1059
600	729824	121189	48963	30000	29599	19141	13050	3124	2025	2026	1059
800	600756	187628	65473	44766	43187	34066	16287	2405	2329	1791	1308
1000	546788	200154	66099	56823	57233	36891	21189	6239	3947	3069	1568
1200	488061	212000	68878	61789	60055	55751	35258	9508	4000	3100	1600
1400	482067	212300	69878	63789	72060	51551	31147	7908	4200	3250	1850
1600	481022	212345	69978	63839	72160	51751	31197	7958	4450	3350	1950
1800	480992	212345	69978	63839	72160	51751	31197	7958	4450	3350	1960

Figure 5.6: Number of Iterations in which the Processes either get the weak lock or enter someone's backpack when executing the Backpack lock 10^6 times

to get the weak lock or enter someone's back pack. This leads to an increase in the number of iterations. As the delay is further increased, almost half of the total number of processes get the weak lock or enter someone's back pack in a single iteration, and the remaining processes iterate more to get the weak lock or to enter some other processes back pack. When the delay is further increased from 1500ms to 2000ms, there is a slight decrease in the number of processes getting the weak lock or entering someone's back pack in single iteration while when the delay is increased further then the processes getting the lock in single iteration remains unchanged.

In the Backpack lock, when the processes have to perform the random coin flip in line 4 of Figure 3.12, almost half of the processes should write their IDs to the shared register in line 6. The remaining processes will read the ID of a process from that shared register and will try to enter that process's back pack in single iteration of the repeat loop in line 2. In the second iteration, half of the processes who wrote to the shared register earlier would perform the coin flip and again half of the processes i.e. 1/4 of the processes would try to enter some other process's backpack whose ID it reads from shared register.

5.3.4.2 How Adding Delay affects the Performance Analysis of other Mutual Exclusion Algorithms

In this test case, now we will consider how the performance of the other mutual exclusion algorithms varies when we add delay in the Critical Section. Each process, after getting the lock, sleeps in the Critical Section for d milli-seconds and then releases the lock, where d is a parameter. For this test case, a delay of 500 to 1500ms is added so that the performance of these algorithms can be compared to the Backpack lock. As seen in Figure 5.6, when the delay of 500ms is added in the Backpack lock, almost 70 percent of the processes get the weak lock or enter someone's back pack in single iterations and as we keep on increasing the delay, processes iterate more to do the same. So, it would be interesting to know the performance of the Backpack lock among all other locks.

Figure 5.7 to 5.9 shows the performance of the mutual exclusion algorithms when the delay of 500ms, 1000ms and 1500ms is added in the Critical Section. The results in Figure 5.7 show that the TS lock and the TTS lock show the best performance among all locks when up to 3 processes execute these locks. But as the contention increases the performance of the TS and the TTS lock deteriorates. The reason behind this may be that as the processes waiting for the TS lock repeatedly performing `test&Set` on the shared variable increase, this causes an RMR every time, thus increasing the contention in the quickpath interconnect. Due to contention in the interconnect, the average time taken by processes to complete `lock()` and `release()` method call increase which leads to bad performance of the TS lock. The performance of the TTS lock is a little bit better than that of the TS lock. This is because in the TTS lock, processes repeatedly read the value of a cached copy of the shared variable and only incur RMRs when they incur cache misses. Adding delay in the Critical Section does not have much impact on the performance of the CLH and the MCS locks.

The Java re-entrant lock has highest average time when up to 16 processes execute this lock but as the contention increases then a steady increase in the average time has been seen.

As the delay increases from 1000ms to 1500ms (see Figures 5.8 and 5.9), the performance of the TS and the TTS locks further deteriorates because processes while waiting for the lock incur more RMRs, hence further increasing the contention in the interconnect.

The difference in the average time taken by processes to execute the RMX lock and the Tree lock decreases when the delay of 1000ms and 1500ms is added. This may be because the randomized and deterministic promotion techniques used in the RMX lock are done by the process while releasing the lock. So, may be no process is promoted while the process holding the lock is asleep in the Critical Section and all the processes try to win the lock on the nodes while moving up towards the root node. The Difference between the two locks further decreases as the delay is increased to 1500ms.

In the Java re-entrant lock, when the delay is further increased to 1000ms and 1500ms, the threshold up to which this lock has highest average time is small. For instance, when delay of 1000 ms is added, the Java re-entrant locks has highest average time when up to 12 processes execute it and this threshold value drops down to 10 when delay is further increased to 1500ms. When contention is high, this lock has low average time.

When the delay of 1500ms is added in the Critical Section, processes executing the Backpack lock takes slightly more time than the CLH lock and the MCS lock. The reason behind this may be that almost half of the processes get the weak lock or enter someone's back pack in a single iteration while remaining processes iterate more which results in slightly higher average time.

5.3.5 Performance of Locks used in Data Structures

As discussed earlier, for the performance analysis of different mutual exclusion algorithms used in data structures, coarse-grained and fine-grained locking techniques are considered. For coarse-grained locking, AVL trees are used. For fine-grained locking, the concurrent linked list is considered as it is simpler to implement in comparison to the concurrent AVL trees with fine-grained locking.

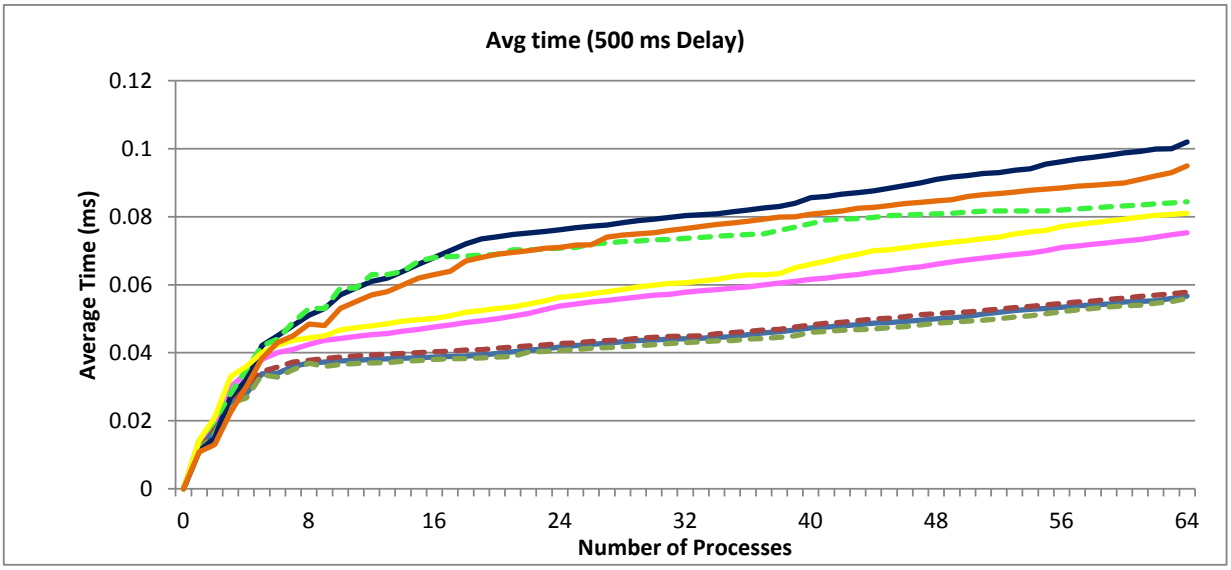
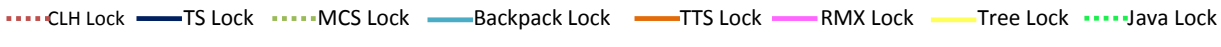


Figure 5.7: Processes executing different Mutual Exclusion Algorithms when Delay of 500ms is added in the Critical Section



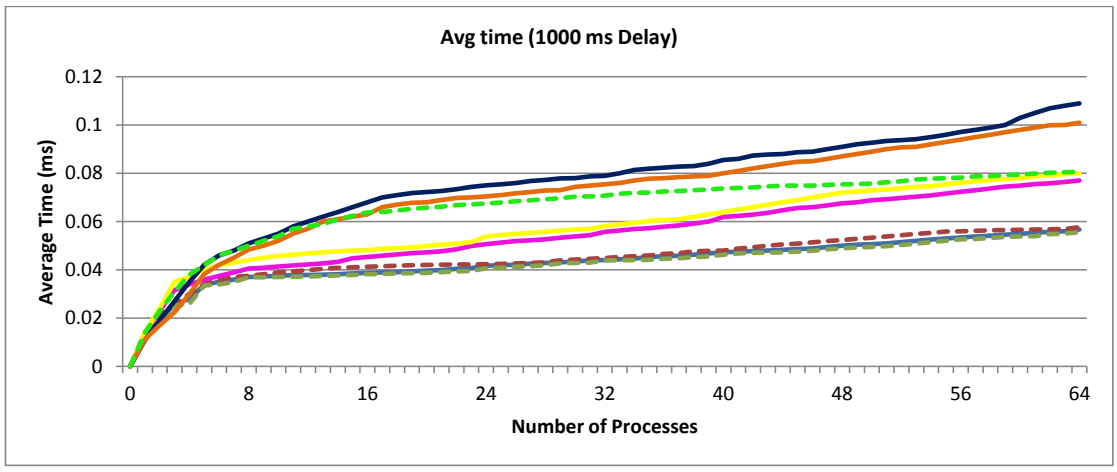


Figure 5.8: Processes executing different Mutual Exclusion Algorithms when Delay of 1000ms is added in the Critical Section

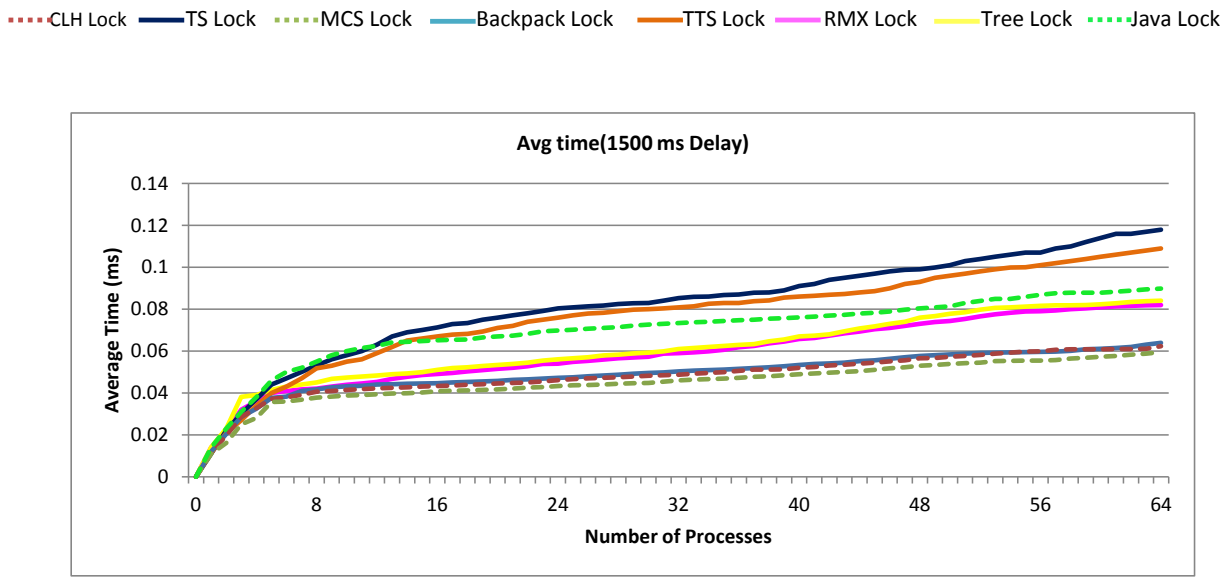


Figure 5.9: Processes executing different Mutual Exclusion Algorithms when Delay of 1500ms is added in the Critical Section

The mutual exclusion algorithms in the AVL tree were executed between 10^5 and 13 x 10^5 times. The average time taken by processes to excecute different locks k x 10^5 times, where k, is some parameter was recorded and the top and bottom 10 percent of the values of the distribution list were omitted to get more precise results. The Figure 5.10 shows the average time taken by processes to execute different locks when the top and bottom 10 percent values are omitted. In the Figure, the x-axis shows the number of times the lock() and release() methods are called i.e., k x 10^5 and the y-axis shows the average time in ms. The top and bottom 10 percent of the values of the distribution list were omitted to get more precise results. Earlier the performance of different mutual exclusion algorithms was analyzed without omitting the top and bottom 10 percent of values and the results showed that when k is lower, the average time taken by processes to execute different locks is higher. This may be because when the processes start to execute the lock, they have to intialize the objects and variables used in the lock.

So, in the another test the top and bottom 10 percent of values in the distribution list were omitted. The results in the Figure 5.10 shows that 8 processes takes almost the same time

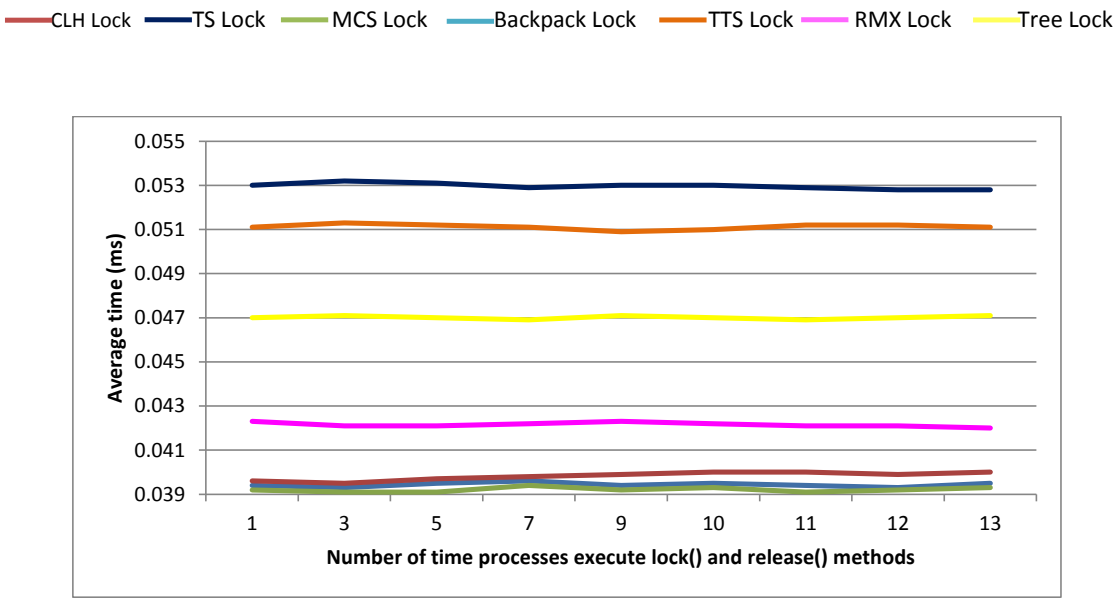


Figure 5.10: 8 Processes executing different Mutual Exclusion Algorithms in the AVL Tree k x 10^5 times when the top and bottom 10 percent of values are omitted

when executing different locks k x 10^5 times in the AVL tree.

The tests were performed on up to 64 processes and the results were similar to the 8 process case (shown in Figure 5.10). Due to this reason, the number of iterations for which mutual exclusion algorithms in the AVL tree was fixed to 1 million and the top and bottom 10 percent values of the distribution list are omitted.

For all the graphs, the x-axis represents the total number of processes executing the mutual exclusion algorithm and the y-axis represents the elapsed time in milli-seconds.

As discussed in the previous chapter, in coarse-grained locking, a process locks the entire data structure, performs operations, and then releases the lock. For this test case, the AVL tree is implemented by using the *Lock* class which implements a mutual exclusion algorithm. The AVL tree used in the experiments is implemented by Brian Cheng (student at Virginia Tech) [10].

Test Case	P_i	P_d	P_l	Figure on Page
1	100%	0	0	78
2	80%	0	20%	79
3	50%	0	50%	80
4	0	80%	20%	81
5	40%	40%	20%	82
6	25%	25%	50%	83
7	10%	10%	80%	84

Figure 5.11: Different probabilities with which processes perform insertions, deletions and look-ups in the AVL tree

The AVL tree is first created with 1,000,000 nodes (the size of the data structure is approximately 28 MB). Recall that in coarse-grained locking, each process, before performing any operation on the AVL tree, first locks the whole tree and then performs the respective operation, followed by releasing the lock on the tree. Each process can perform insertions, deletions and look-ups on the AVL tree. The value which is to be inserted or deleted or which is to be looked-up is chosen uniformly at random in $\{1, \ldots, 10,000,000\}$. For generating the random integer values uniformly, the Random class in Java is used. The total time is recorded for each process to perform the respective operation on the AVL tree which includes the time taken to lock the entire tree, performing the insertion or deletion or look-up and then releasing the lock. Then the average time per operation is calculated. Different insert, delete, and look-up combinations are considered.

Each process performs insert, delete and look-ups depending on the different parameters, p_i, p_d, p_l, where p_i is the parameter representing the probability with which processes insert, p_d is the is the parameter representing probability with which processes delete and p_l parameter for look-ups. Figure 5.11 shows different test cases considered for coarse-grained locking. Each test case is instantiated by above mentioned parameters. Processes at random decide which operation they have to perform and the probability distribution of operations depends on p_i, p_d and p_l.

From these test cases, we hope to gain some insight about how the performance of

different mutual exclusion algorithms varies with the contention.

5.3.5.1 Performance Results using Coarse-Grained Locking

In the coarse-grained tests, the time taken by processes to perform deletions, insertions and look-ups is measured. In the Figures 5.12 - 5.18 on pages from 78 to 84, the results show that the time taken by processes to perform insertions or deletions on the AVL tree is much longer than the time taken to perform look-ups. The MCS lock, the Backpack lock and the CLH lock show the best performances among all the locks. The performance of the RMX lock and the Tree lock is not as good as the locks mentioned above but still their performance is better than the TS lock and the TTS lock. When the contention is small, the TS lock and the TTS lock show the best performances among all the locks but as the contention increases, the performances of these locks degrade. For example, as seen in Figure 5.14, the TS lock and the TTS lock have shortest average time per operation among all other locks when up to 16 processes execute them, while in test case where all processes insert (i.e. $p_i = 100$ %), the TS and the TTS lock show best performances when up to 8 processes execute them. In the other test cases, the threshold may be different but still the performance of the TS lock and the TTS lock is better than all other locks when contention is small (see Figures 5.13 - 5.15). The reason behind the poor performance of the TS and the TTS locks when many processes execute these locks is contention in the interconnect (for details see Section 5.3.3). The Java re-entrant lock has highest average time among all other processes when the contention is small but as the contention increases, it shows same behavior as explained in the Section 5.3.3.

The time taken by processes to perform look-ups (with different locks) is smaller compared to the time taken by them to perform insertions and deletions as in the case of look-ups, processes hold the lock for short time in comparison to insertions and deletions. The test cases in which p_l is longer than p_i and p_d, the threshold for which the TS and the TTS lock performs better is higher. In this case, the threshold for which the Java re-entrant lock

peforms worst is also higher. For example, in Figures 5.17 and 5.18, the TS and the TTS lock show the best performances for up to 48 processes. In the above mentioned figures, the Java re-entrant lock show worst performance for up to 30 processes (approx). So, higher the probability of look-ups better the TS and the TTS lock perform. The reason for this may be that when p_l is higher, the time for which processes hold the lock is small in comparison to when processes hold the lock for performing insertions and deletions. As the time for which processes hold the lock is small, so, this leads to less contention in the interconnect, hence, leading to good performances of the TS and the TTS locks.

5.3.5.2 Experiments using Fine-Grained Locking

A linked list is first created with 1,000,000 nodes (the size of the linked list is approximately 28 MB). Each process performs insertions, deletions or look-ups on the list. Look-ups in this case are lock-free, i.e. processes do not get the lock for performing look-ups. The value to be inserted, deleted or to be looked-up is decided uniformly at random from the domain of 1 to 10,000,000.

Each process decides at random which operation it has to perform and the probability distribution of operation to be performed depends on the parameters, p_i, p_d, p_l, where p_i is the probability of insertion, p_d is the probability of deletion and p_l is the probability of a look-up. Figure 5.19 shows different test cases considered for fine-grained locking. Each test case is instantiated by above mentioned parameters and the operation which each process performs is decided by these parameters.

The total time is recorded for each operation on the linked list, including the time to lock the node, performing the insertion or deletion and then releasing the lock. Then the average time per operation, insertion, look-up and deletion is calculated. The mutual exclusion algorithms in the linked list were also executed between 10^5 and 13 x 10^5 times. The average time taken by 8 processes to execute different locks k x 10^5 times, where k is some parameter was recorded and the top and bottom 10 percent of the values of the distribution list were

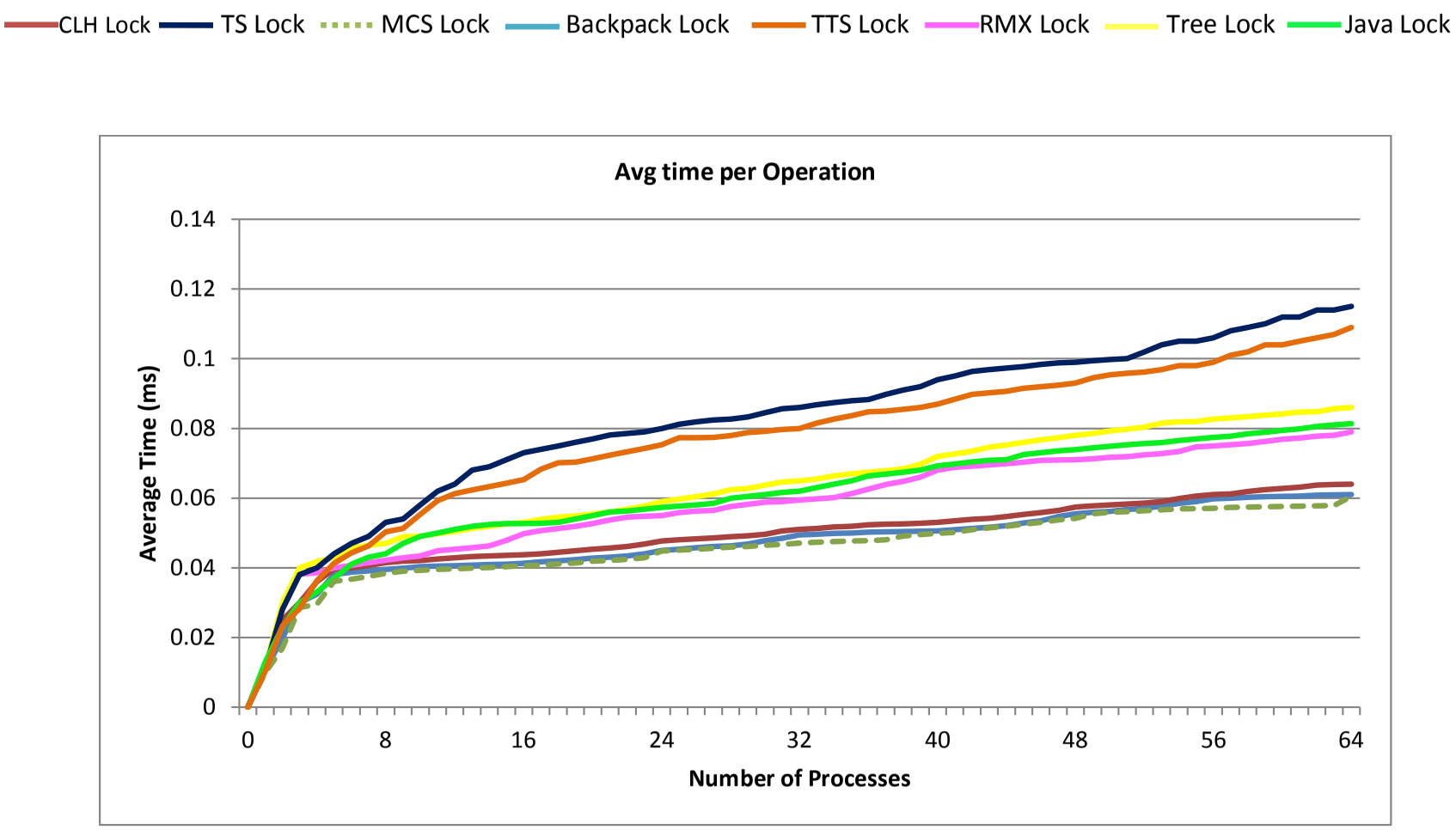


Figure 5.12: $p_i = 100\%$

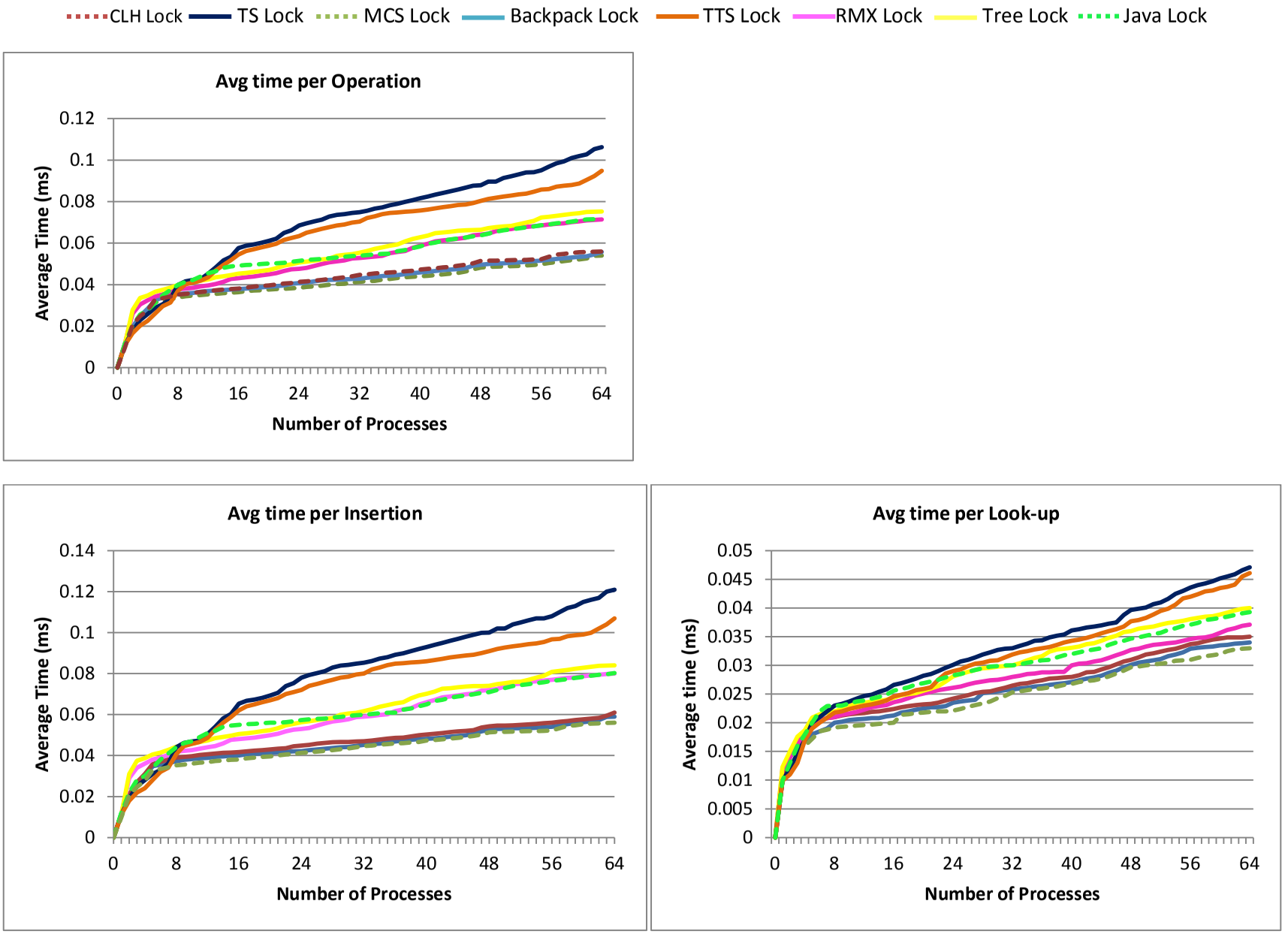


Figure 5.13: $p_i = 80\%$ and $p_l = 20\%$

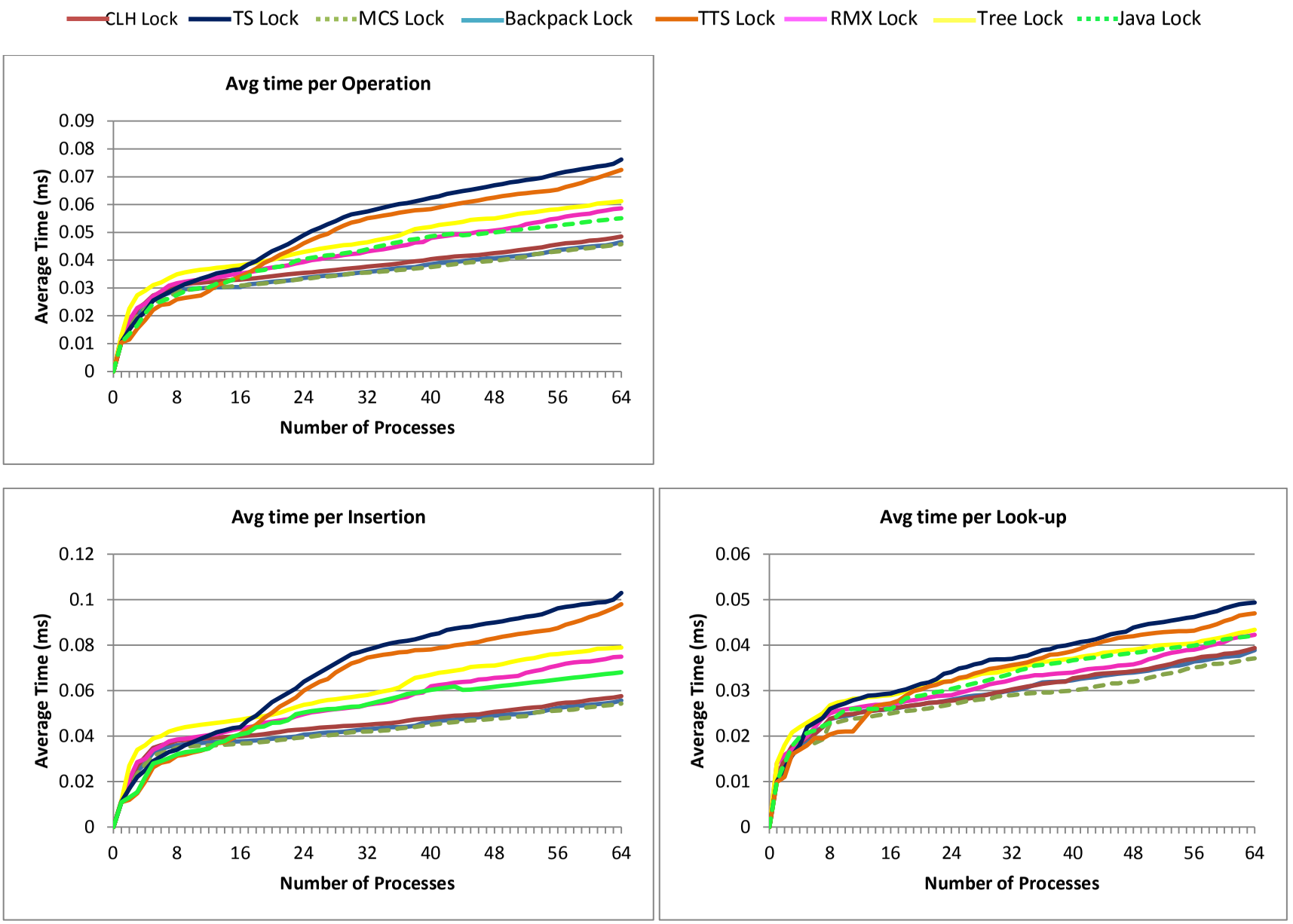


Figure 5.14: $p_i = 50\%$ and $p_l = 50\%$

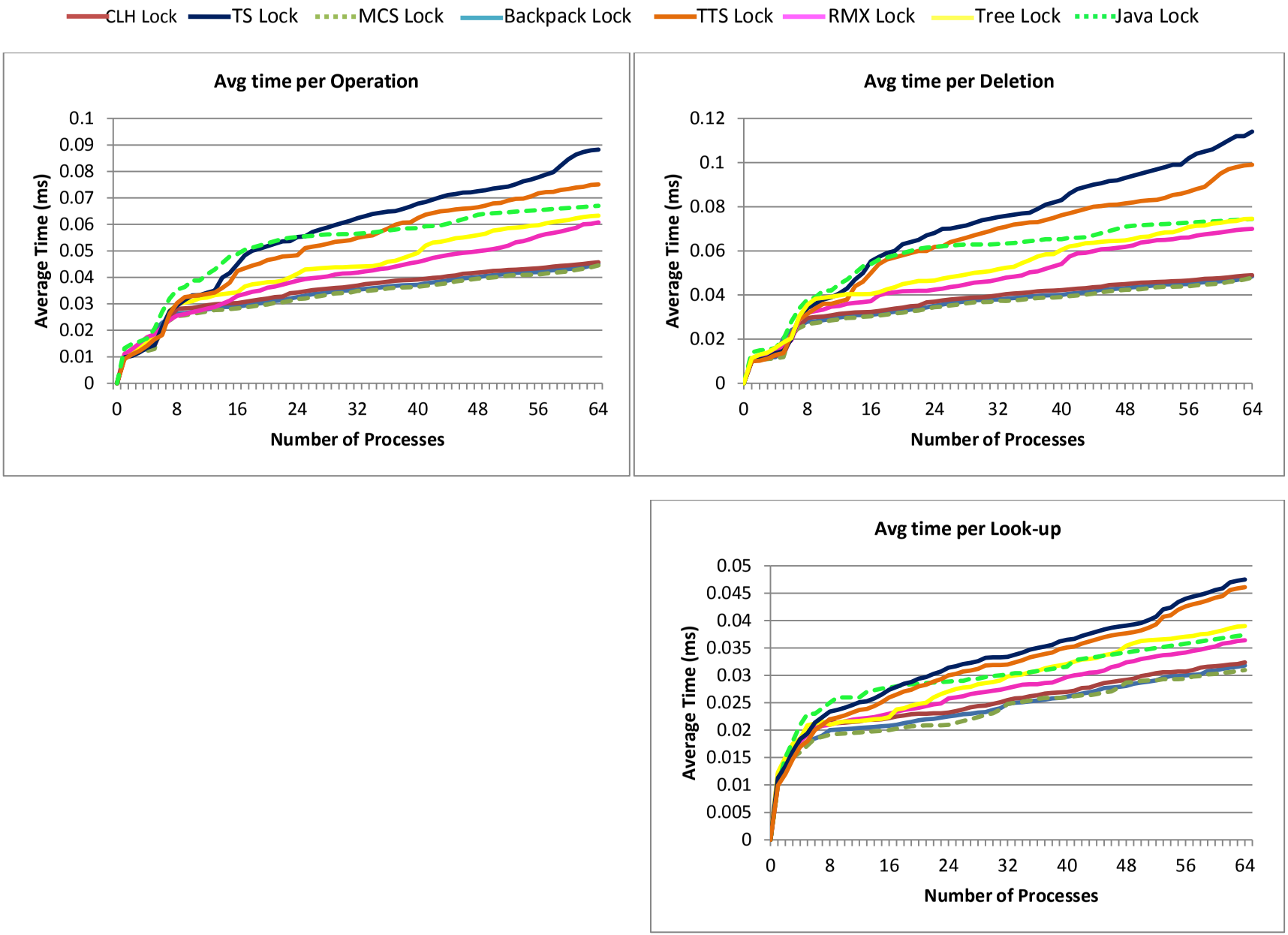


Figure 5.15: $p_d = 80\%$ and $p_l = 20\%$

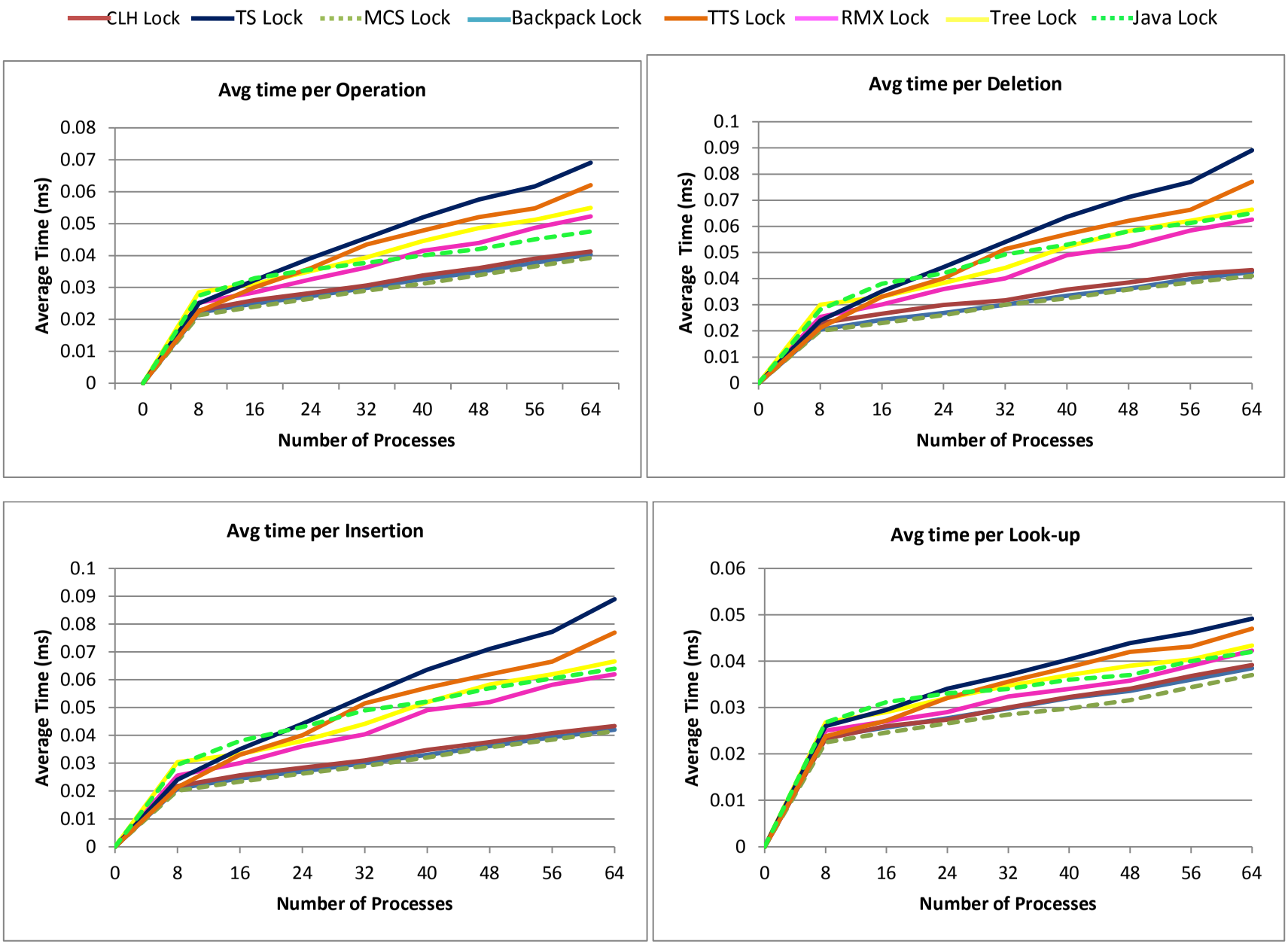


Figure 5.16: $p_i = 25\%$, $p_d = 25\%$ and $p_l = 50\%$

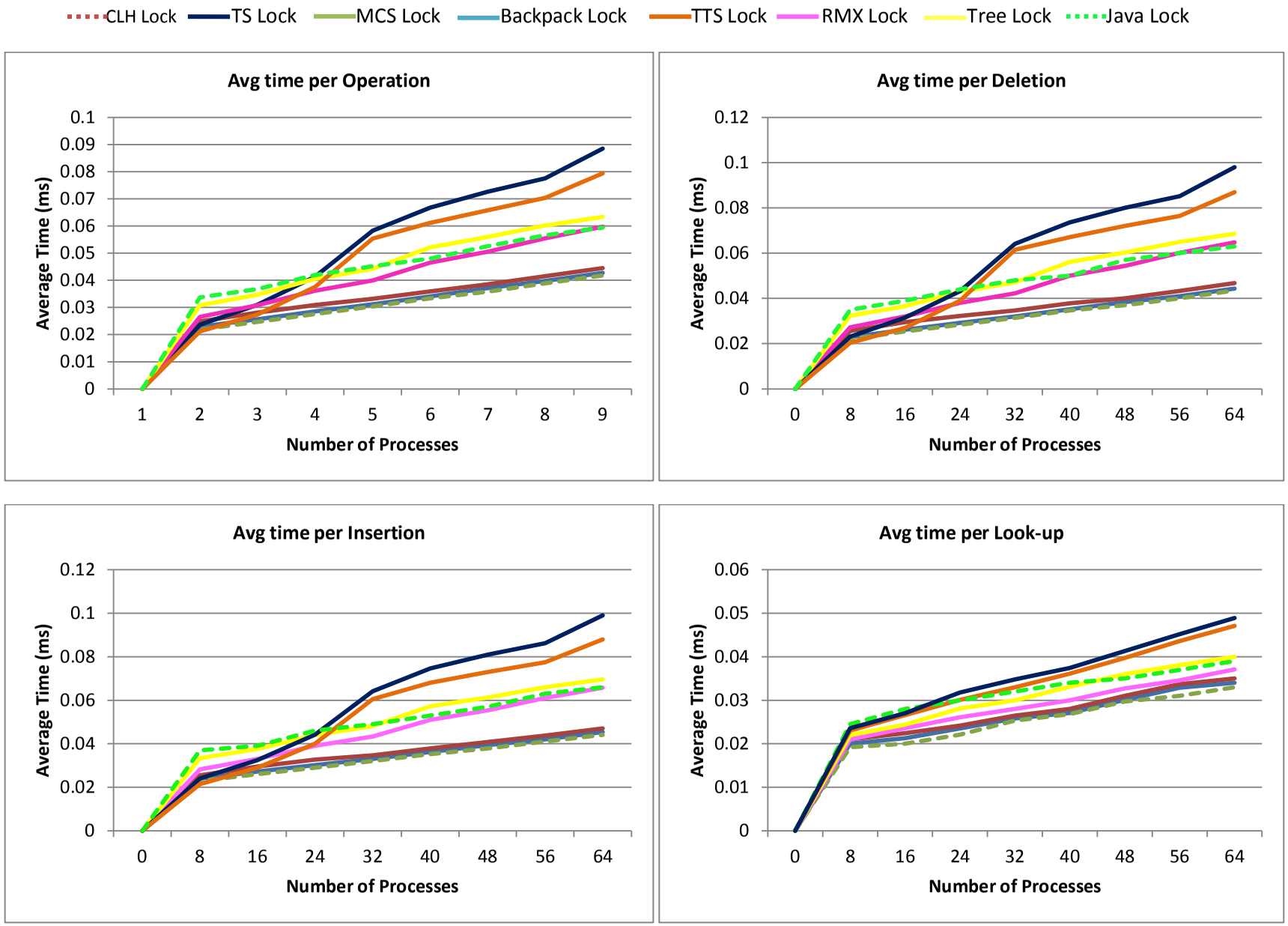


Figure 5.17: $p_i = 40\%$, $p_d = 40\%$ and $p_l = 20\%$

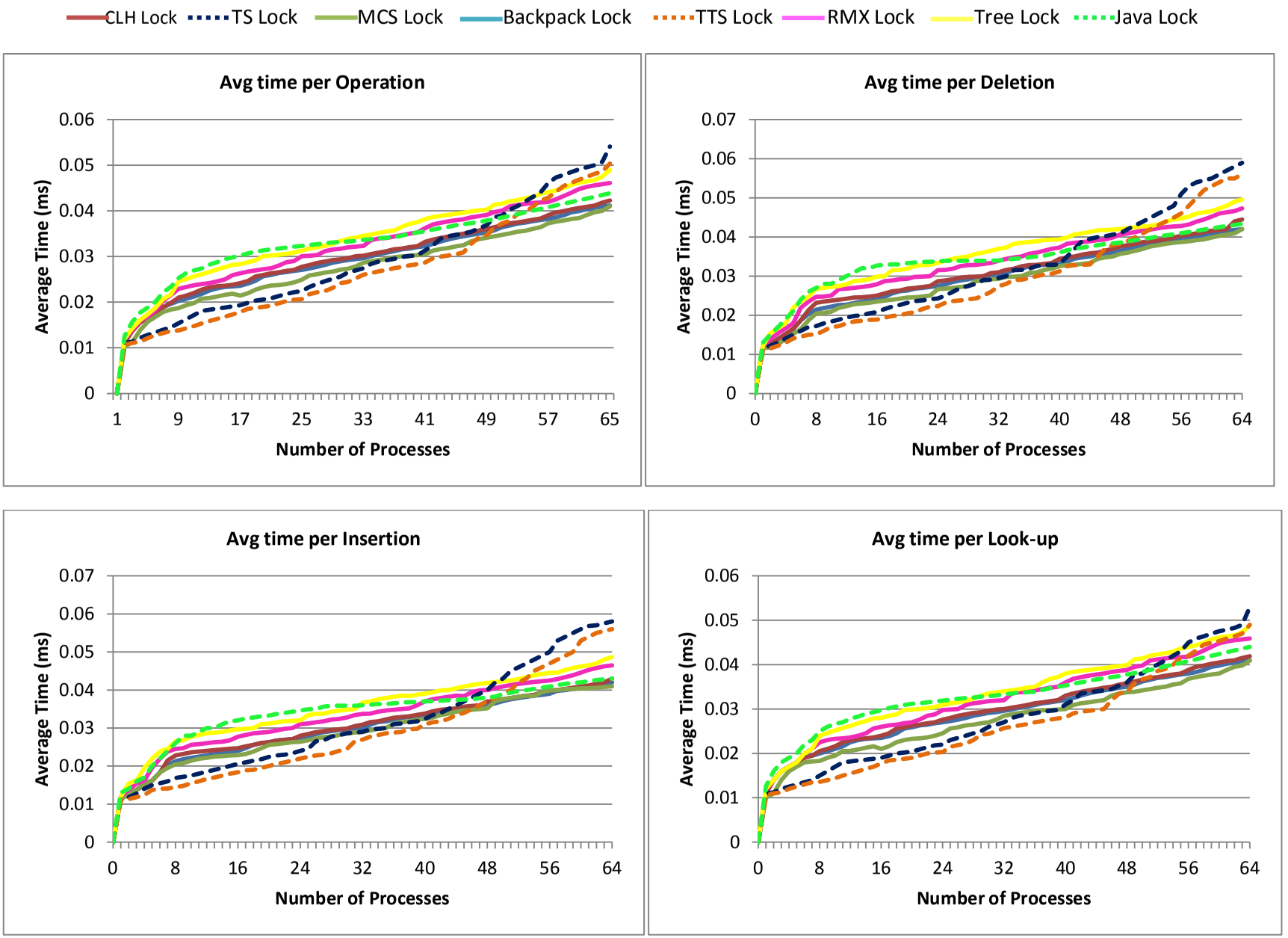


Figure 5.18: $p_i = 10\%$, $p_d = 10\%$ and $p_l = 80\%$

Test Case	P_i	P_d	P_l	Figure on Page
1	100%	0	0	88
2	80%	0	20%	89
3	50%	0	50%	90
4	0	80%	20%	91
5	40%	40%	20%	92
6	25%	25%	50%	93
7	10%	10%	80%	94

Figure 5.19: Different probabilities with which processes perform insertions, deletions and look-ups in the linked list

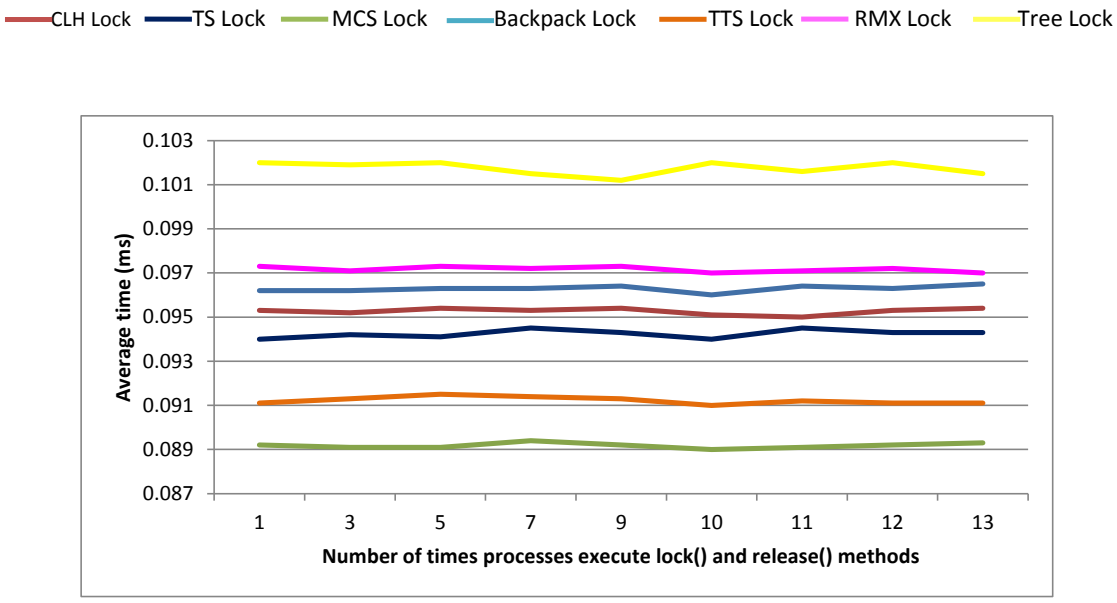


Figure 5.20: 8 Processes executing different Mutual Exclusion Algorithms in the linked list k x 10^5 times when the top and bottom 10 percent of values are omitted

omitted to get more precise results. The Figure 5.20 shows the average time taken by processes to execute different locks when the top and bottom 10 percent values are omitted. In the Figure, the x-axis shows the number of times the lock() and release() methods are called i.e., k x 10^5 and the y-axis shows the average time in ms. The results in the Figure 5.20 shows that 8 processes takes almost the same time when executing different locks k x 10^5 times in the list.

The tests were performed on up to 64 processes and the results were similar to the 8 process case (shown in Figure 5.20). Due to this reason, the number of iterations for which mutual exclusion algorithms in the list was fixed to 1 million and the top and bottom 10 percent values of the distribution list are omitted.

5.3.6 Measurement Results using Fine-Grained Locking

Figures 5.21–5.27 on pages 88–94 show that the MCS lock, the Backpack lock and the CLH lock have the best performances among all the locks in most of the test cases. When the probability with which processes perfrom look-up is high, the TS and the TTS locks benefit from low contention and the Java re-entrant lock suffers in low contention. In the test case where processes insert and delete with 10% probability and perform look-ups with 80% probability, the TS and the TTS lock have the shortest average time per operation. This is because look-ups are lock-free and few processes access the lock to perform insertions and deletions. In the other cases, when the contention is small, the TS lock and the TTS lock show the best performances among all the locks but as the contention increases, the performances of these locks degrades. For example, as seen in Figure 5.23, the TS lock and the TTS lock have the shortest average time per operation among all other locks when up to 16 processes execute them, while in the test case where all processes insert (i.e. $p_i = 100$ %), the TS and the TTS lock show the best performances when up to 8 processes execute them. In the other test cases, the threshold may be different but still the performance of the TS lock and the TTS lock is better than all other locks when contention is small (see

Figures 5.22 - 5.24). The reason behind the poor performance of the TS and the TTS locks when many processes execute these locks is contention in the interconnect (for details see Section 5.3.3).

The performance of the RMX lock and the Tree lock is not as good as the locks mentioned above but still their performance is better than the TS lock and the TTS lock (when the contention is large). The test cases in which p_l is longer than p_i and p_d, the threshold for which the TS and the TTS lock performs better is higher. For example, in Figure 5.26, the TS and the TTS lock show the best performances among all other locks. So, the higher the probability of look-ups, the better the TS and the TTS lock perform. The reason for this may be that when p_l is higher, the number of processes which try to get the lock are fewer which leads to low contention in the interconnect, and hence good performances of the TS and the TTS locks.

The Java re-entrant lock has longest average time among all other processes when the contention is small but as the contention increases a steady increase in the average time is seen. The test cases in which p_l is longer than p_i and p_d, the Java re-entrant locks show poor performance as few processes access lock (i.e contention is small). This lock small average time only if contention is high.

All the Figures show that the processes take less time in the fine-grained locking to perform insertions or deletions in comparison to the coarse-grained locking. The difference in the time take in the fine-grained and coarse-grained locking is not only due to the locking strategies used but also may be due to the different data strutures used in both of the locking strategies.

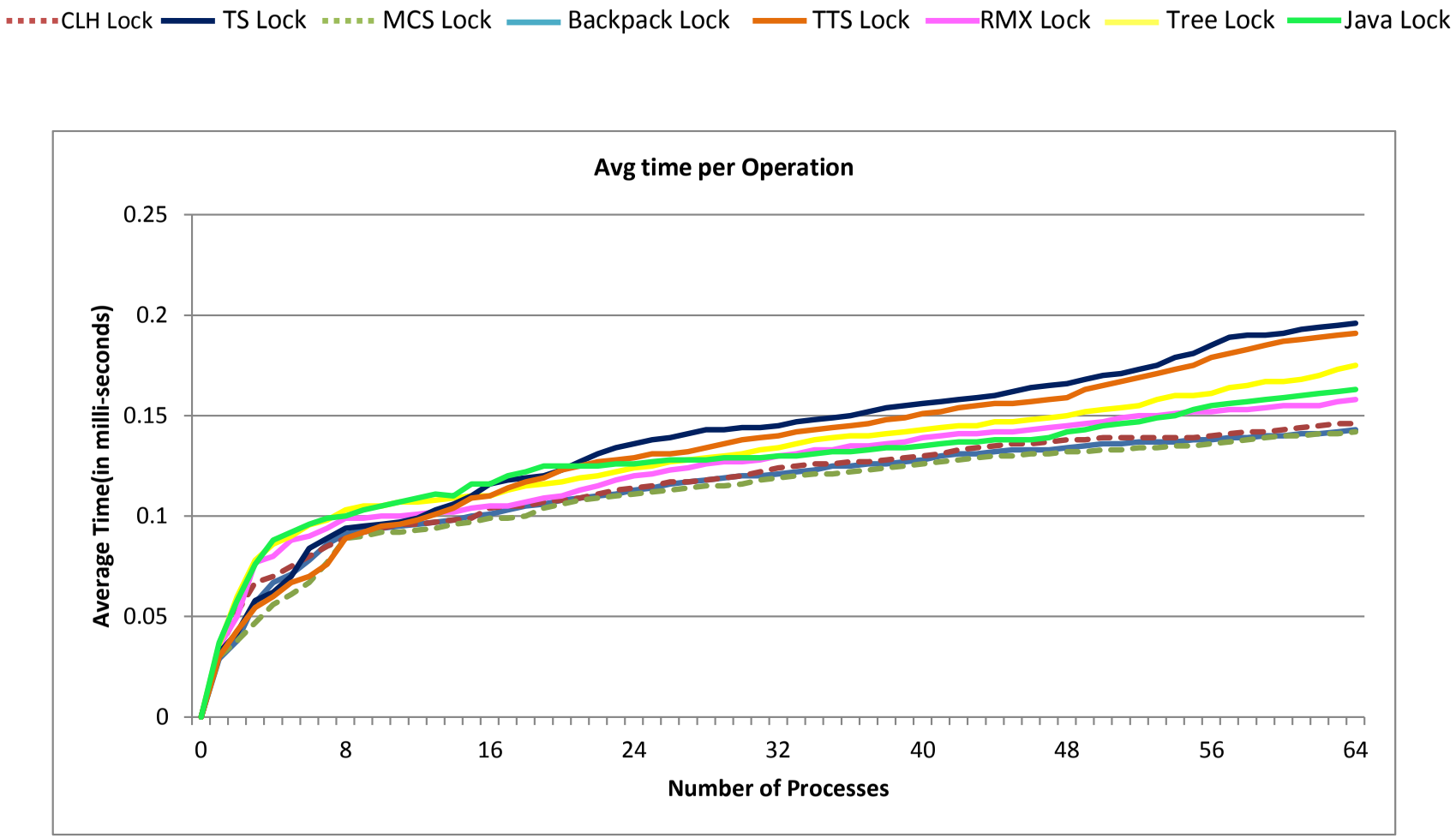


Figure 5.21: $p_i = 100\%$

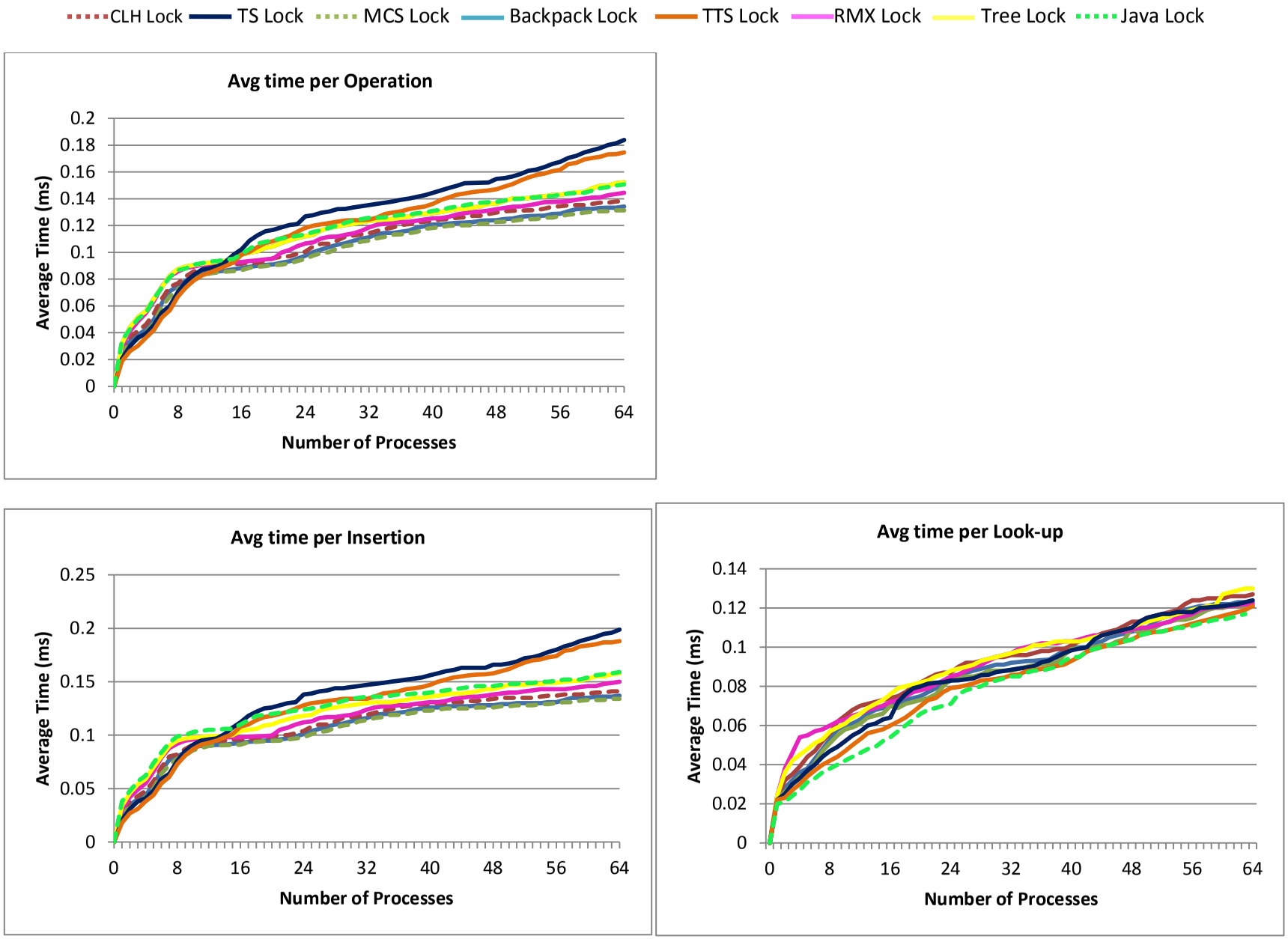


Figure 5.22: $p_i = 80\%$ and $p_l = 20\%$

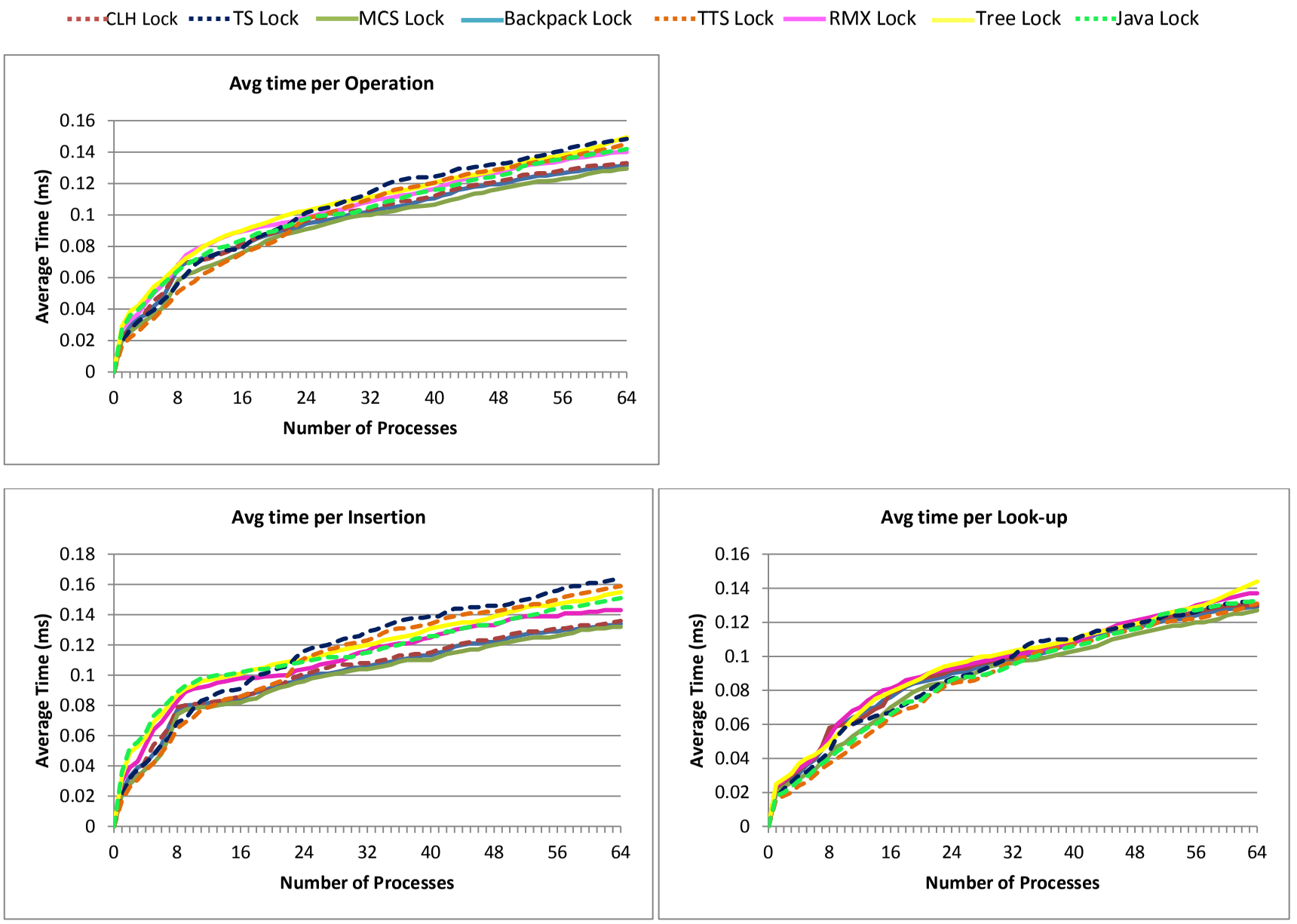


Figure 5.23: $p_i = 50\%$ and $p_l = 50\%$

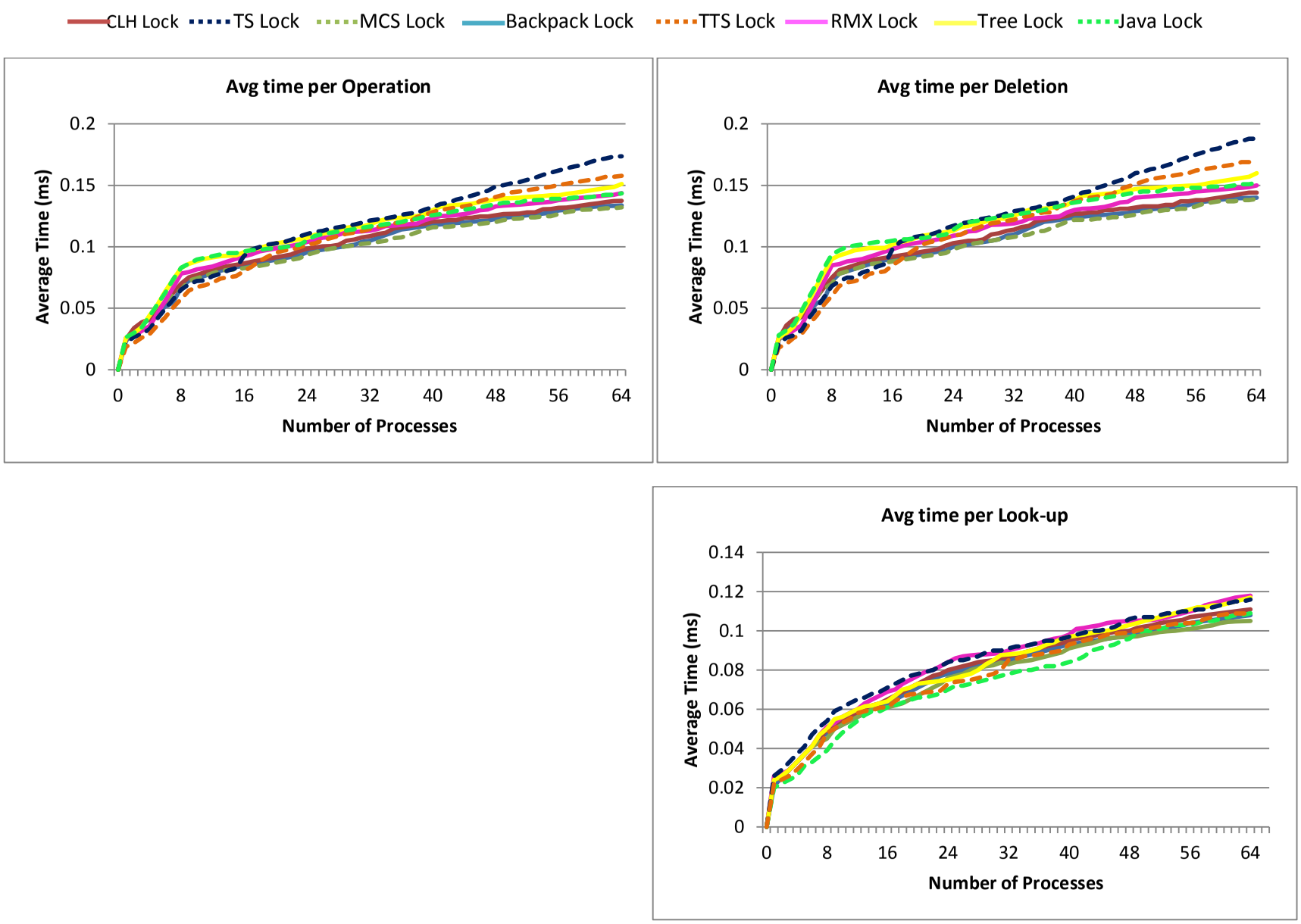


Figure 5.24: $p_d = 80\%$ and $p_l = 20\%$

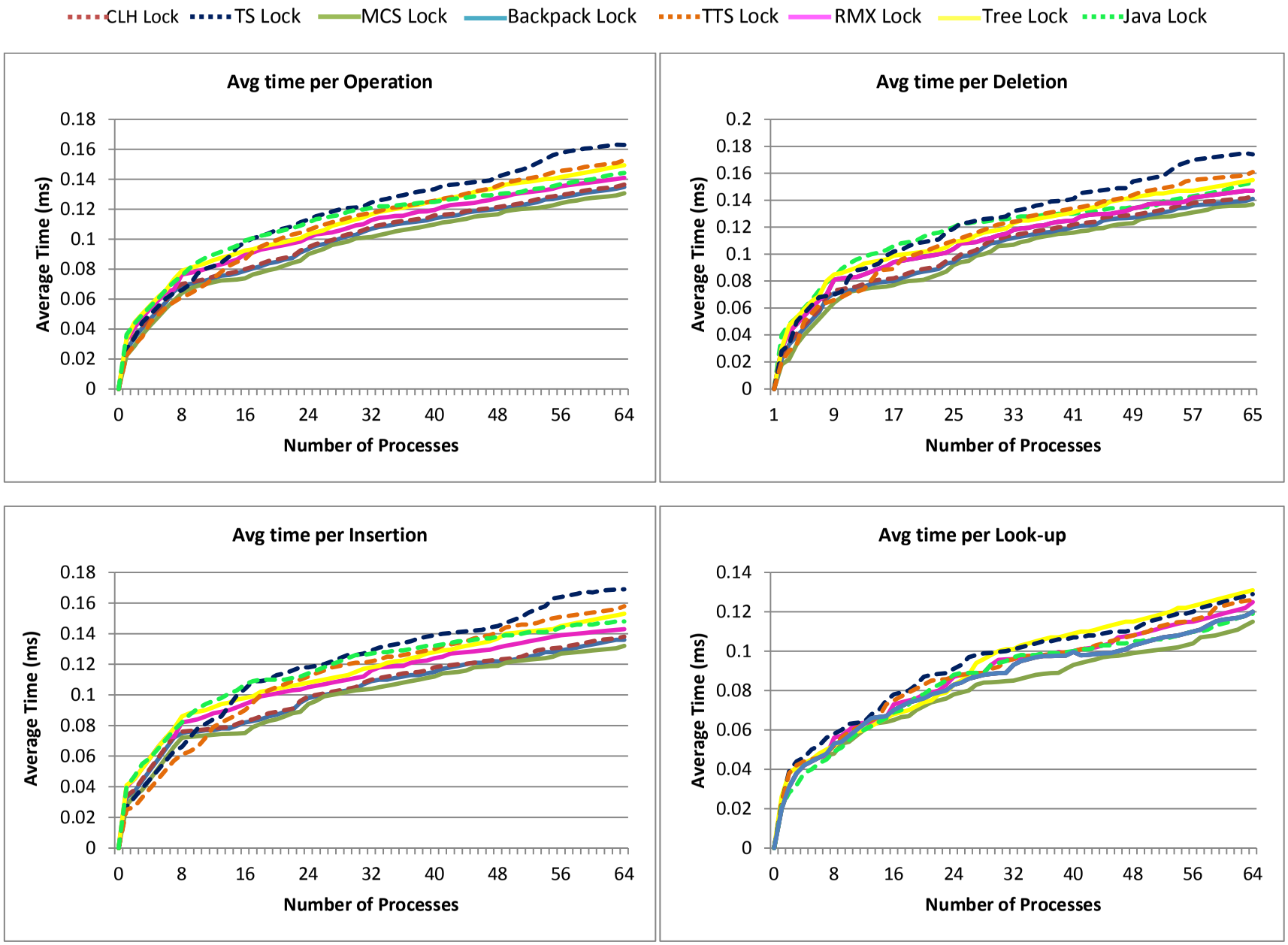


Figure 5.25: $p_i = 40\%$, $p_d = 40\%$ and $p_l = 20\%$

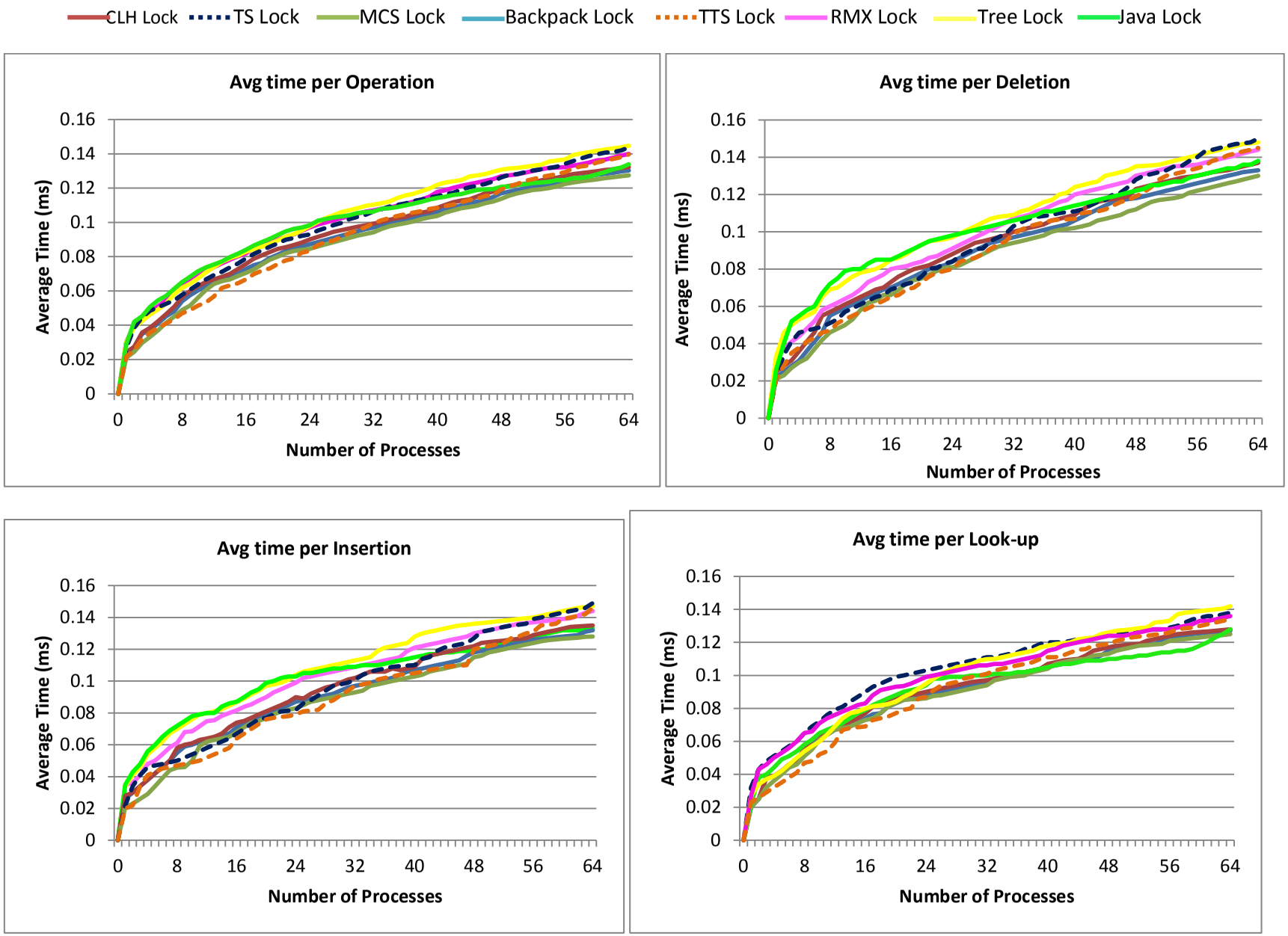


Figure 5.26: $p_i = 25\%$, $p_d = 25\%$ and $p_l = 50\%$

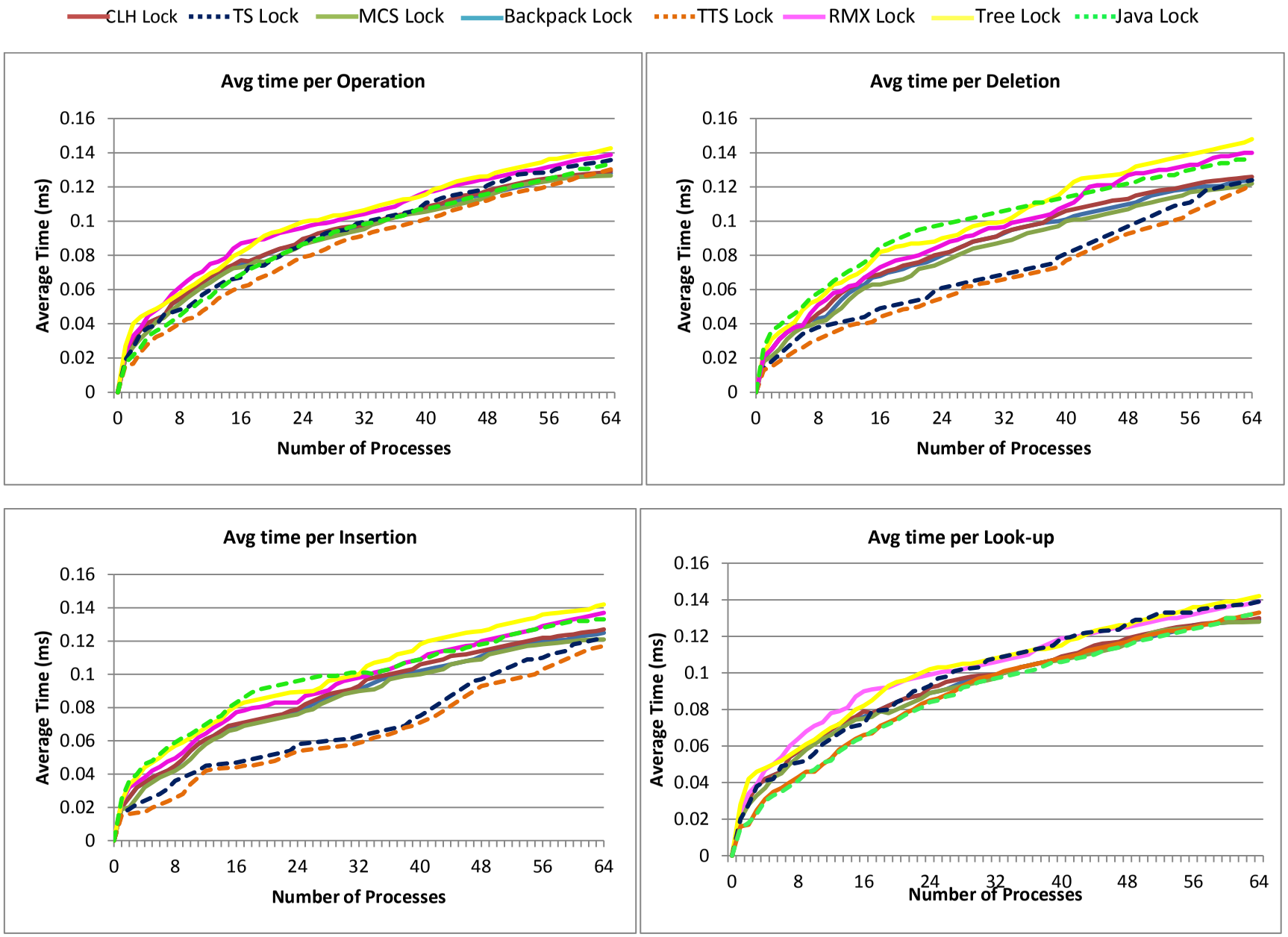


Figure 5.27: $p_i = 10\%$, $p_d = 10\%$ and $p_l = 80\%$

Chapter 6

Conclusion

6.1 Introduction

This thesis presents a performance comparison of various mutual exclusion algorithms. The main focus of the thesis is on measuring the time taken by processes for executing different mutual exclusion algorithms in isolation and also in different data structures. While measuring the performance of mutual exclusion algorithms in isolation and in data structures, processes perform insertions, deletions or look-ups with varying probability. The performance of algorithms is also evaluated by adding a delay of k ms in the Critical Section, where k is some parameter. In all the above experiments, the average time per operation, insertion, deletion and look-up is recorded to gain some insight about how the performance of different mutual exclusion algorithms varies with the contention and the delay.

6.2 Thesis Contribution

Mutual exclusion is a very important problem in distributed computing. It prevents two processes from accessing some shared resource at the same time. It is a standard building block for shared memory algorithms. Mutual exclusion algorithms used by shared memory algorithms should be efficient as operating systems and shared memory algorithms frequently make use of them. If the mutual exclusion algorithm is not efficient, then the efficieny of shared memory algorithms may suffer.

In this thesis, the performance of mutual exclusion algorithms in isolation shows that the MCS, the CLH and the Backpack locks are most resilient to contention, followed by the RMX and the Tree locks in all test cases. The Backpack lock scales in almost the same

way as the MCS and the CLH lock. The TS and the TTS locks are most affected by the contention but they benefit from low contention. For instance, in the experiments using locks in data structures (where the look-ups are lock-free), the performances of these locks is the best among all other locks (see section 5.3.6).

For concurrent programming, when the contention is low, the TS or the TTS locks are to be preferred. These locks only need a `test-and-set` object, which makes them simple to implement. In these locks, processes have to execute only few operations if there is no or little contention (as seen in Section 5.3.6 on Page 86). The TTS lock should be preferred over the TS lock because in the TTS lock, processes incur less RMRs in comparison to the TS lock as seen in results in Chapter 5.

When the contention is high, the MCS, the CLH, or the Backpack lock should be preferred. The advantage of the Backpack lock over the MCS and the CLH locks is that in the MCS and the CLH locks, strong primitives such as `getAndSet()` are used which may not be supported by certain systems. Even if they are supported by hardware, their efficiency is not known and may vary on different architectures. Therefore, it is better not to rely on their efficiency. David, Guerraoui and Trigonakis [12] presented a paper in which they conducted similar performance tests of several mutual exclusion algorithms on processor architectures such as Niagara, Tilera, Xeon and Opteron. They considered the MCS lock, the CLH lock, the TS lock, the TTS lock and the Array based locks. The experiments were conducted for up to 80 processes.

In our thesis, the Array based lock was not implemented as this algorithm is not space efficient. We have considered several randomized mutual exclusion algorithms to compare their performances with deterministic mutual exclusion algorithms. All the randomized and deterministic locks in our thesis were tested under different scenarios such as delay being added in the Critical Section, in data structures and in isolation.

The results of the paper by David, Guerraoui and Trigonakis show that the MCS and the

CLH lock are most resilient to contention while the TS and the TTS locks show the worst performance among all locks (when contention is high). This matches our results. Our results provide evidence that the Backpack lock (a randomized mutual exclusion algorithm) can compete with other established locks using stronger primitives (i.e. the MCS and the CLH locks).

6.3 Possible Extensions of this Thesis

In future, the performance of these locks can also be tested on the different shared memory multiprocessors such as Niagara, Opteron, and Tilera to see how does the shared memory multiprocessors impact the performance of these locks under low and high contentions. We can implement the locks using different programming language and see whether this impacts the performance of locks. As each programming language has its own compiler and code optmiziming techniques so, implmenting these locks in different programming language may impact their performance.

Bibliography

[1] James H. Anderson and Yong-Jik Kim. Fast and scalable mutual exclusion. In *Distributed Computing, 13th International Symposium*, pages 180–194, 1999.

[2] James H. Anderson and Yong-Jik Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43, 2000.

[3] James H. Anderson and Yong-Jik Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, 2001.

[4] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel Distributed Systems*, 1(1):6–16, 1990.

[5] James K. Archibald and Jean-Loup Baer. An economical solution to the cache coherence problem. In *Proceedings of the 11th Annual Symposium on Computer Architecture, ISCA*, pages 355–362, 1984.

[6] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[7] Hagit Attiya and Vita Bortnikov. Adaptive and efficient mutual exclusion (extended abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 91–100, 2000.

[8] Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight rmr lower bounds for mutual exclusion and other problems. In *Proceedings of 40th ACM Symposium on Theory of Computing, STOC*, page 447, 2008.

[9] Michael A. Bender and Seth Gilbert. Mutual exclusion with $o(\log^2 \log n)$ amortized

work. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS*, pages 728–737, 2011.

[10] Brian Cheng. AVL tree Implementation. https://code.google.com/p/ece5510-concurrent-avl-tree/source/browse/trunk/src/edu/vt/ece/main/?r=35/, 2011.

[11] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Parallel Distributed Systems*, 27(12), 1978.

[12] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, 2013.

[13] D.Hendler and P.Woelfel. Adaptive randomized mutual exclusion in sub-logarithmic expected time. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 141–150, 2010.

[14] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[15] S.J. Frank. Tightly coupled multiprocessor systems speed memory access times. *Electronics*, 57:164–169.

[16] G.Giakkoupis and P.Woelfel. A tight rmr lower bound for randomized mutual exclusion. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC*, pages 983–1002, 2012.

[17] Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-rmr implementations of cas and other synchronization primitives using read and write operations. In *Proceedings of the 26th Annual Symposium on Principles of Distributed Computing*, pages 3–12, 2007.

[18] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture, ISCA*, pages 124–131, 1983.

[19] M. Herlihy and N. Shavit. *Array Based Locks*, page 150. Morgan Kaufmann, New York.

[20] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*, page 154. Morgan Kaufmann, 2008.

[21] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*, page 27. Morgan Kaufmann, 2008.

[22] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*, page 151. Morgan Kaufmann, 2008.

[23] J. Kessels. Arbitration without comman modifiable variables. 1982.

[24] Yong-Jik Kim and James H. Anderson. Adaptive mutual exclusion with local spinning. *Distributed Computing*, 19(3):197–236, 2007.

[25] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 378–391, 2005.

[26] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[27] Oracle. Atomic Integer Class. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/AtomicInteger.html/, 2011.

[28] Oracle. Number Class. http://docs.oracle.com/javase/6/docs/api/java/lang/Number.html/, 2011.

[29] Abhijeet Pareek. Rmr efficient randomized abortable mutual exclusion. Master's thesis, Department of Computer Science, University Of Calgary, 2012.

[30] Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In *In Proceedings of 9th Annual ACM Symposium on Theory of computing, STOC*, pages 91–97, 1977.

[31] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *American Federation of Information Processing Societies: National Computer Conference, AFIPS*, pages 749–753, 1976.

[32] Gadi Taubenfeld. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and fifo algorithms. In *Proceedings of the 18th International Symposium on Distributed Computing*, pages 56–70, 2004.

[33] Wikipedia. Power Edge R910. http://www.dell.com/downloads/global/products/pedge/en/poweredge-server-11gen-whitepaper-en.pdf/, 2010.

[34] Wikipedia. Comapre and Swap. http://en.wikipedia.org/wiki/Compare-and-swap/, 2013.

[35] Wikipedia. DDR3 Memory. http://en.wikipedia.org/wiki/DDR3_SDRAM/, 2013.

[36] Wikipedia. Intel Quickpath Interconnect and Hyper-Transport. http://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect/, 2013.

[37] Wikipedia. Sun Naigara. http://en.wikipedia.org/wiki/UltraSPARC_T1,http://en.wikipedia.org/wiki/UltraSPARC_T2,http://en.wikipedia.org/wiki/SPARC_T3/, 2013.

[38] Wikipedia. Test and Set. http://en.wikipedia.org/wiki/Test-and-set/, 2013.