A Framework to Study the Performance of the Group Mutual Exclusion Algorithms

by

Darshik Bharat Shirodaria

B.E, University of Mumbai, India, 2012

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

THE UNIVERSITY OF NORTHERN BRITISH COLUMBIA

Aug, 2017

Abstract

Group mutual exclusion problem generalizes the classical mutual exclusion problem, a fundamental problem in concurrent programming. It arises in applications involving sharing resources such as memory and data. In group mutual exclusion, a process requests for a "forum"; processes requesting the same forum may access the critical section simultaneously. Several algorithms have been proposed for the group mutual exclusion problem, but very few studies have been conducted to compare the performances of these algorithms by means of execution on actual machines. Besides the studies conducted have been a mere one-on-one comparison. Also, there exists no testing environment that accommodates multiple algorithms and compare their executions.

This work aims at testing the performance of group mutual exclusion algorithms extensively by executing multiple such algorithms in a test framework. We propose to build an automated test framework to execute these algorithms, both individually and collectively under various experimental setups and observe their performances graphically using several performance metrics. Our experiments would constitute several collective comparison studies of algorithms along with replicating a few one-on-one comparison experiments from the literature.

To use the algorithms into our framework, we intend to translate them from pseudo codes to source codes. The aim is to eventually creating a repository of these source codes such that they could be used for other applications besides our framework.

Dedicated to my grandfather...

Late. Liladhar Dharamshi Shirodaria

Acknowledgements

Throughout my time as a graduate student at UNBC, I came across many people who have encouraged me in one way or the other. I would like to thank them all. Among them, some deserve special thanks.

First of all, I would like to thank my supervisor, Dr. Alex Aravind for his continuous support, encouragement and mentoring. His timely inputs and directions along with his contributions with availing the right resources for my research played a significant role in my entire masters journey. I thoroughly enjoyed working with him. I take this opportunity to acknowledge my committee members, Dr. Waqar Haque and Dr. Balbinder Deo for their valuable time and suggestions for my thesis. I would like to thank Dr. Pranesh Kumar for agreeing to serve as my external examiner and Dr. Oye Abioye for being the chair of my defence.

I would like to thank all my peers and friends who supported me during my masters. I want to thank Rafael Roman Otero and Mehul Solanki, two of my peers who always made sure to be by my side throughout my masters. I would also like to thank my lab mates Rahim, Conan, Shanthini, and Gurpreet for their support. A special mention for Chris Botha, our former system administrator, for all his help, right from arranging server machines to extending support for any OS related issues. I would also like to extend my gratitude to Dr. Mahi Aravind for the amazing dinners that she hosted and for the banana cakes that she would send to lab on all special occasions. Lastly, a sincere thank you to my friends, Deeshen, Sonal, Emily, my well wishers Binita Parekh, Deven Choksi and my relatives Atul and Nutan Suchak and my cousin Salony and all other other important people in my life for all their support. Without the support of all these people my masters wouldn't have been a successful journey.

Finally, I would like to thank my parents for helping me pursue my masters in first place. Without their contributions and sacrifices, I wouldn't have been able to accomplish to dream of pursuing my masters.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Over the past couple of decades, we witnessed a paradigm shift in computer hardware technology. Multicore processors have taken modern-day computing by storm as almost all the machines now come equipped with a chipset consisting of two or more processors capable of executing many programs simultaneously. With the rise of multicore computers, to exploit the capabilities of multicore processors to the fullest, concurrent programming is becoming a mainstream activity [14].

In concurrent programming environments, processes sharing resources such as data structures and databases is a frequent activity where a **process** is an instance of a program running. As resource sharing is becoming increasingly common, coordinating access to shared resources among the processes in the system is a challenge faced by several applications. One such example is that of a network printer shared by multiple computers. Here, it is essential to ensure that at any given instance only one of the multiple printing requests arising from the machines trying to print copies is processed. This scenario is a well defined in concurrent programming as the mutual exclusion problem. The problem states that in a system consisting of n competing processes trying to access a shared resource, at any given instance, only one of the

competing processes should be able to access the shared resource [3].

One of the generalizations of the mutual exclusion problem is group mutual exclusion (GME) problem, introduced and solved initially by Joung in 1999 [1]. GME concerns systems or applications where processes with same interests can access the shared resource simultaneously, while processes with conflicting interests wait for their turn to access the shared resource. GME problem could be observed in several applications. For example, in a video conferencing system with an electronic whiteboard, any of the participating users in the video conference can use the whiteboard to post information they wish to share and all other participants should be able to view the contents of the whiteboard simultaneously. Another example is that of a CD jukebox described by Joung [1]. In this example, a CD jukebox is a resource shared by a set of processes. Multiple processes interested in the same CD can simultaneously access it while the processes interested in accessing some other CD must wait until that particular CD is loaded into the jukebox. In this example, CDs are the forums. In the group mutual exclusion problem processes trying to access shared resources request for a *forum*. A *forum* denotes the interest of a process. Processes requesting the same forum are allowed to access the shared resources simultaneously.

GME could also be considered as a generalization of the readers-writers problem, where all the readers can request for the same forum while the writers can execute themselves exclusively in separate individual forums [14]. Also, concerning concurrency, this problem is similar to k-exclusion problem [21].

1.2 Motivation

Several algorithms have been proposed to provide solutions for the GME problem [1,2,4–6,8–10,14,17–19]. Most of these algorithms are innovative in their ways. While innovation aspect is theoretically appealing, the most important metric for them to be practically attractive is their performance under various conditions suitable for specific applications.

Until now, only two performance studies for the GME algorithms were conducted [2,10]. Each of these studies aimed at one-on-one comparison of two algorithms using a set of experiments. A comprehensive performance study involving multiple group mutual exclusion algorithms has not performed until now.

1.3 Research Challenge and Inspiration for the Work

An immediate question would be why such a study has not been performed yet. There are several reasons, though the primary ones are the technical challenge and the availability of hardware system to conduct such a study. Conducting performance study of concurrent algorithms in the shared memory system is complicated as it involves deeper understanding of complex algorithms to be implemented and machine level implementation details.

Until recently, systems that can execute programs truly parallel was not available for general software development community. Shared memory based parallel computers having such capability were quite expensive and were usually owned only by high-performance computing groups. Also, these parallel computer systems primarily facilitate synchronized parallelism for a limited set of highly regular parallel applications using higher level libraries such as OpenMP. Any implementation of fine-grained asynchronous parallelism using OpenMP-like support is inefficient. The arrival of multicore systems as a mainstream platform has addressed the later issue.

Although mutual exclusion problem dates back to early 60's, only recently Buhr et al. [15] have studied the performance of the mutual exclusion algorithms. The inspiration for the research presented in this thesis primarily comes from this work.

This thesis mainly focussed providing a testing environment in the form of a framework where all the algorithms for the GME problem could be tested for their performances on multicore machines, both collectively and individually. Thus, this thesis proposes to design a testing framework to conduct performance studies of the group mutual exclusion algorithms. The objective of designing and developing of a

testing software framework alone was quite ambitious. However, without actual performance studies conducted using this framework, it would be challenging to demonstrate its use. So, we decided to implement several GME algorithms and conduct a few performance studies. My supervisor, Dr. Aravind, is an expert in designing and proving concurrent algorithms. With his expert advice and the timely availability of a multicore server (capable of executing maximum 40 threads) from the Department of Computer Science, we were able to complete the implementation and performance studies of GME algorithms.

The main steps involved in this research are:

1. Literature survey.
2. Design and development of the test framework to study group mutual exclusion algorithms.
3. Identification and implementation of performance metrics.
4. Implementation of several group mutual exclusion algorithms
5. Conducting performance experiments on implemented algorithms and illustrating their performance results

1.4 Contributions

This thesis has several contributions. The main contributions are:

1. Design and implementation of a test framework to study the performance of the group mutual exclusion algorithms in the shared memory context. The performance of group mutual exclusion algorithms can be evaluated under different experimental settings using this framework.
2. A limited performance study of several group mutual exclusion algorithms.

3. A repository of the group mutual exclusion algorithms in the form of C programs. These program files are portable, and therefore could be useful for designing real-world applications or systems. They could also be used as a reference for implementing other group mutual exclusion algorithms into sources codes. Insights for designing a new algorithm or an improvement to an existing algorithm can be derived by trial and error modifications and from the observations of the behavior of existing algorithms.

1.5 Organization of Thesis

In Chapter 2, we present the design choices and challenges along with research questions concerning our framework. In chapter 3, we formally introduce the group mutual exclusion problem and its properties. In Chapter 4, we lay out the architecture of automated test framework that we have designed for executing the algorithms for the group mutual exclusion. We also pin down details of each of the components of our framework and define a few terminologies used in the framework. Besides, we present the workflow and system primitives used within the system or while translating algorithms into code bases. These primitives play a vital role in optimizing the performance of algorithms in our framework. In Chapter 5, we present the performance metrics that would be used to evaluate and compare the performance of the algorithms. We also present all our performance experiments conducted using our framework, and their respective results followed by analyzing and summarizing these results. In Chapter 6, we conclude and summarize our thesis, and outline the future directions to extend the work done in this thesis.

Chapter 2

Framework: Design Choices and Challenges

2.1 Research Questions

The primary contribution of this thesis work is the design and development of a software framework to study the group mutual exclusion algorithms. Let us start with the main research questions we need to address to achieve this goal.

1. What would be the primary objective of the framework?
2. What are the main features of the framework?
3. What are the main components of the framework?
4. How to model and design these components?
5. What software technologies should we use so that the software portable and have a wider use?
6. To demonstrate the working of the framework, which group mutual exclusion algorithms should be implemented and studied?

7. What performance experiments would we like to conduct?

These questions are not straightforward to answer. Finding suitable solutions for them require reading literature on not only group mutual exclusion, but also software design and modeling and simulation of concurrent systems. They also lead us to make some critical decision choices.

2.2 Design Choices

When we talk about choices involving the design and development of the proposed framework, there are two main choices must be made. What kind of computer systems that we are planning to adopt - real or a simulation system, and what set of performance metrics need to be supported. To get an idea to make these choices, we briefly review the existing work in that direction for the group mutual exclusion from the literature. As we indicated in the introduction, there are two earlier works on performance study, one by Keane and Moir [2] and the other by Blelloch, Cheng, and Gibbons [10].

Keane and Moir conducted their performance using the Augmint simulator [22], developed at the University of Illinois, Urbana-Champaign. Augmint [22] simulates a cache-coherent multiprocessor by switching between multiple threads of execution and runs on a single machine. Blelloch, Cheng, and Gibbons performed their study using an actual machine, a Sun UltraEnterprise 10000 with 64 250 MHz UltraSparc-II processors. So we have two choices, and apparently, the preferred one is using real computers that can capture the performance in real life implementations. Therefore, we decided to go with the choice of using real machines. Since we don't have Sun system in our lab, our choice here is Intel processor-based system. Specifically, we use Dell PowerEdge FC630 rectified blade server controlled using Integrated Dell Remote Access Controller (iDRAC8). The machine comes equipped with two chipsets of Intel(R) Xenon(R) CPU E5-2650 v3, each comprising of 10 cores (20 threads) and operating at the speed of 2.30GHz. We have Ubuntu 14.04 LTS as the operating

system running on our machine.

Some of the key advantages of using real systems are:

- The framework could run on multiple machines, each with a different processor architecture.
- Most of the multicore machines (or the compiler within) support or provide a built-in instruction set while simulation would require simulating many higher level instructions.
- The total lines of code to be written are lesser as the code base for creating a simulation environment is being eliminated.

For the choice of performance metrics, we again look at the works of Keane and Moir, and Blelloch, Cheng, and Gibbons. Keane and Moir conducted five experiments to test and compare their algorithm's performance with Joung's algorithm:

i Mutual exclusion - concurrency level is reduced to 1;

ii Contention-free - only one process exists in the system that executes CS 1000 times without performing any work inside CS;

iii Variable concurrency - involves a set of 32 competing processes, each of them requesting a forum number from range *1..M*. The experiment runs until all the processes execute their CS code 1000 times;

iv Non-empty CS - here the processes perform some work when inside CS unlike other three experiments where no work is done inside CS; and

v Threshold - used a set threshold for potential concurrency attainable using the algorithms.

Blelloch, Cheng, and Gibbons performed similar sets of experiments, and key change is they varied demand for various forums (setting a fixed probability for each forum). These algorithms were run for 1000 rounds (that each thread executes).

We decided to provide options to conduct these experiments. The main difference is that our framework allows the user to set the time duration (in seconds) for the simulation, instead of a fixed number of rounds. Also, our framework allows computing the number of forum entries, fairness, delays in forum entries, and forum switches. An attractive feature of our framework is automating the workload generation, simulation process, result collection, and performance visualization.

2.3 Design and Implementation Challenges

Recently, Buhr et al. [15] designed a test environment [16] to evaluate and compare the algorithms for the mutual exclusion problem. This work is the first to study of comparing performances of multiple algorithms for mutual exclusion problem simultaneously under one testing environment for shared memory systems. While we draw primary inspiration from this work, it's functionality is very limited, and it is command-line based. We would like to fully automated framework having started to finish functionality - automatic load generation to performance graph visualization integrated. Designing such a comprehensive test framework is a challenging task, and we adopt a modular approach for designing our framework. We break down the entire process into several stages with each of them resulting in the formation of a module.

Apart from the requirements specified above, the framework needs to be relatively simple and easy to use. It should consist of all the components that would be essential for performing end to end experiments. Concurrent Locking framework [16] provided a good test environment for mutual exclusion algorithms. However, their system lacked a few useful components which when included within the system would make it complete. We aim to address those limitations by providing those key components that were missing in their work.

Along with conducting performance studies, designing a test framework that can be used for comparing performance complexities for all the possible algorithms would just be an ideal testing environment for algorithms proposed for the group mutual exclusion problem. It would be even better if this framework could be scalable such that incorporating new experimentation settings into it could be done with ease along with accommodating newer algorithms designed in future. Having an automated test framework that is scalable helps setup first standard platform for studying performance complexities of algorithms in group mutual exclusion.

The first component is the Graphical User Interface (GUI) for ease of usage. Since their system is designed to run on UNIX based operating systems, in the absence of a GUI, it is often tedious for users to operate a software or a framework in console mode operation. Having GUI provides ease of usage for the users. The second key component that we would want to include in our system which wasn't available in Concurrent Locking [16] work is that of graphically displaying performance results. Having the test framework plot results generated from performance experiments in graphical form within the system would certainly be useful as it would eliminate the burden of porting results to plotting tools for generating graphical results.

By implementing the algorithms into programs codes, we could help create a repository of all the existing algorithms under one roof and that too, in the form of program files. This repository could be a useful resource for the implemented algorithm codes could be used for various applications and because of the portable nature of these source files, it could be distributed to larger audiences.

First, based on the literature review, an automated test framework with components that can be useful in executing any group mutual exclusion algorithm is designed and implemented. Second, the components necessary for studying the performances of these algorithms are determined and added to the framework. Third, we work to implement algorithms into the code base which is a daunting task. We overcome this by focusing on thoroughly understanding the logic behind each of the algorithms

implemented which helped us in retaining their core essence even after the implementation. Finally, the functionality of the proposed framework is demonstrated by executing all the implemented algorithms in the framework and then illustrating their performances through performance metrics.

2.4 Summary

In this chapter, we discussed the comparison studies conducted for evaluating the performances of algorithms for the group mutual exclusion. Also, we derived motivation for this thesis by investigating the related work and identifying open problems that could be addressed. In the process, we specified our design approach and the research methodology we used for achieving our proposed goals and highlighted the contributions of this thesis. We discuss these contributions extensively in the following chapters.

Chapter 3

Group Mutual Exclusion Problem

3.1 System Overview

The system contains a set of n independent cyclic processes that are interested in accessing a shared resource. These processes communicate only via shared variables. Each process interested in accessing the shared resource, at any given time, either perform some local computation or communicates with other processes through shared memory, using variables or data structures as shown in Figure 3.1.

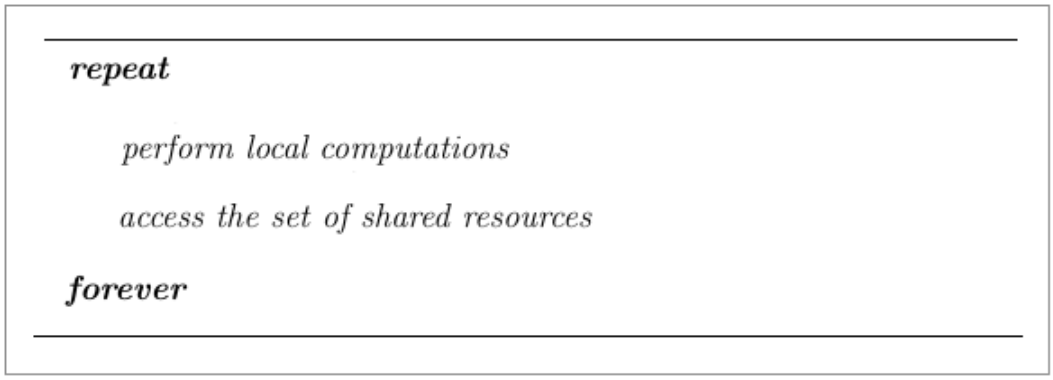

Figure 3.1: Tasks of a cyclic process in shared memory system

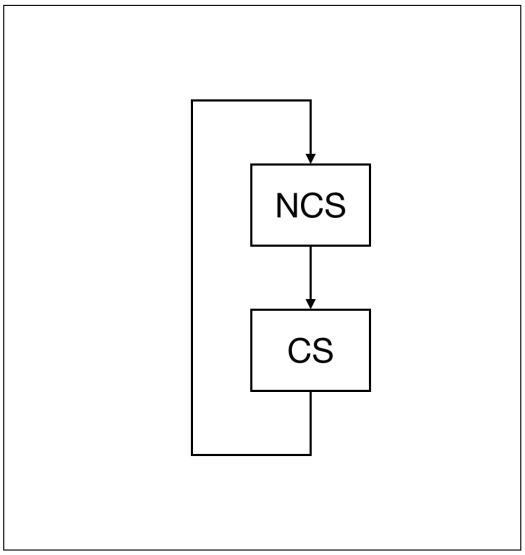

Figure 3.2: Process execution in shared memory system

If processes get the access to the shared resources, they can write the shared variables while the other processes read those shared variables, simultaneously. Upon accessing the shared resources, the process notifies other processes through these same shared variables of the availability of the shared resources. Each process in such systems loops continuously to perform the actions as shown in Figure 3.1. The code segment of processes in the (group) mutual exclusion could be categorized into two parts: Critical Section (CS) code which is the code segment that is responsible for accessing the shared resource and Non-Critical Section (NCS) which consists of all the remaining code. With the terminologies defined above, Figure 3.1 could be modified as shown in Figure 3.2. The following assumptions are made for the given system:

1. The execution speed of processes is arbitrary but finite.

2. Time to execute CS code for each process may be arbitrary but finite.

3. No process can fail while executing its CS code.

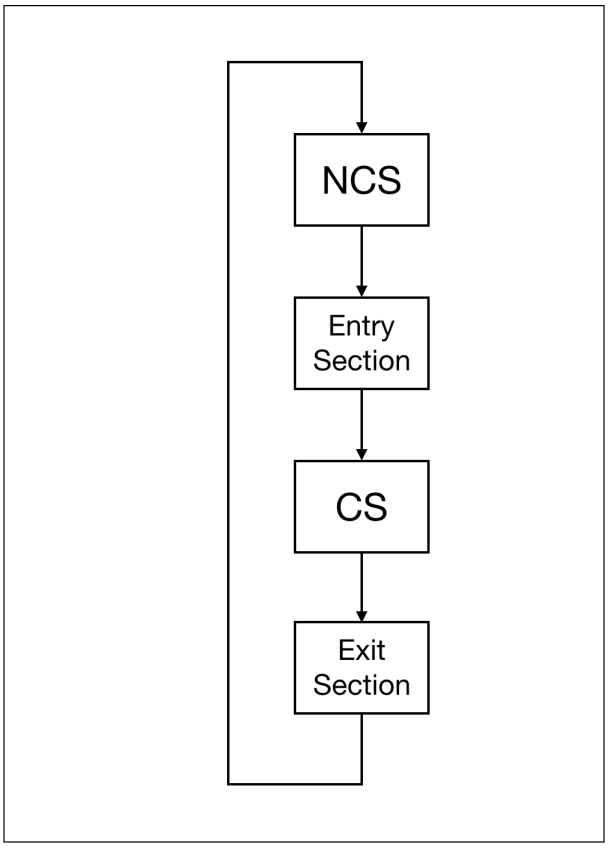

Figure 3.3: Solution for the group mutual exclusion problem

The system would be relatively simple to implement if each process had its resources for its execution. However, with the set of resources being shared among processes for their execution, there arises inconsistency in accessing the resources with multiple processes trying to access them.

At any given time, there could be multiple processes requesting access to the shared resource. In this instance, the communication between the processes plays a vital role in the efficient functioning of the system. The concurrent algorithms provide a solution to this problem of inconsistency by enabling proper communication and coordination between processes before and after accessing the shared resource. The section of code that is responsible for communication between processes before they access the shared resources is called as Entry Section and the section that is

responsible for communication by processes after accessing the shared resources is called as Exit Section. Thus, Entry Section and Exit Section when inserted before CS and after CS respectively, form the desired solution which is shown in Figure 3.3.

In the Entry Section, the processes *choose* a forum. This is followed by the *doorway*, which is a non-waiting code section, followed by a **waiting section** and an optional **opening section** section. It is executed by a process if it is the first process *initiating* the *request* for a forum, say *f1*, or when no other process is *requesting* access to a forum when the process is in it's *waiting section*. Executing this section ensures that the process does wait for entering it's CS code. The request is *acknowledged* when the process enters it's CS code. A forum *f1* is *opened* when there are processes interested in it that can enter their CS code in a bounded number of its steps and is *closed* when there are no processes interested in it that can enter their CS code. A forum *f1* is said to be *active* when the processes interested in *f1* are executing their CS code [14].

There are two primary memory models for which the existing solutions for the group mutual exclusion problem are based on shared memory context. They are, the **Distributed Shared Memory** (DSM) model and the **Cache Coherent** (CC) model. The primary difference between the two models is where the variables are stored and how the processes access those variables. In the DSM model, every variable is associated with exactly one process [18]. Trying to access a variable associated with some other process results in the process making a **Remote Memory Reference** (RMR). In CC model, the variables are shared, and global as well as are not related to any processes. The consistency in CC model is ensured using cache coherent protocols which instantaneously update all the copies.

3.2 Properties and Preliminaries

In [1], Joung introduced two preliminary logical components that are essential for designing any algorithm for the group mutual exclusion. These components include:

- Mutual Exclusion component to achieve Mutual Exclusion among forums
- Concurrent Entering component to facilitate concurrent access to processes interested in the same forum

In the following section, we will discuss the properties that have been presented from the existing algorithms proposed thus far for the group mutual exclusion problem in shared memory context.

3.2.1 Properties under the mutual exclusion component

In any group mutual exclusion algorithm, the properties of the mutual exclusion component ensure exclusiveness among forums to be active at any given instant, thereby facilitating processes interested in the same forum to execute their CS together. The following are the properties of the mutual exclusion component:

P1. **Safety:** At any given point of time, only one forum must be set to active. That is if a process p requesting for forum f is executing it's CS code, then no other process requesting a forum other than f should not be able to execute it's CS code simultaneously.

P2. **Lockout Freedom:** A process p requesting for a forum f should eventually be able to access it.

P3. **Bounded Exit:** If a process enters the Exit Section code, then it should be able to reach NCS in a finite number of steps.

3.2.2 Properties under concurrency

Joung introduced the property of *concurrent entering* [1] to facilitate maximum forum utilization.

P4. **Concurrent Entering:** ensures that the processes requesting the same forum are able to access it concurrently. Which means, if a process p requests for a

forum *f1* and no other process requests a different forum, then process p enters forum *f1* in bounded number of it's own steps. However, to ensure Lockout Freedom, the given forum should be *active* only for a finite period of time.

However, his definition was not precise, and several papers following that attempted to give more formal and precise definition of concurrent entering which eventually leads to introducing new properties. Here are the properties under concurrency:

3.2.2.1 Concurrent Entering vs Concurrent Occupancy

Several algorithms presented for the group mutual exclusion have attempted to give **Concurrent Entering** a stronger and precise definition. As argued in [4], Hadzilacos points out that Keane and Moir in [2] gave a precise, but weaker definition as compared to Joung which was vague. According to Keane and Moir, if a process p request for a forum $p.t$ and no process q such that $q.t$ $p.t$ is executing outside its NCS, then p eventually executes its CS code even if no other process accessing the same forum comes out of it is defined as *concurrent entering*. This definition was termed as **Concurrent Occupancy (P5)** by Hadzilacos. According to Hadzilaocs, using Concurrent Occupancy, a process's execution of CS may be delayed which would not be the case with Concurrent Entering in the absence of contention.

Some of the papers in the literature argue that concurrent entering is effective only when there are no conflicting requests. In fact, few of the algorithms proposed for the group mutual exclusion do not satisfy the concurrent entering property [2, 5, 10]. Without a proper concurrency property, in scenarios of conflicting requests, the group mutual exclusion problem tends to act as a simple mutual exclusion problem.

The property of **No Late Entry** introduced in [10], is effective even in scenarios of conflicting requests. However, this property is restrictive as it prohibits concurrency after a particular moment during the time when forum f is *active*.

P6. **No Late Entry:** No process p is allowed to enter an *active* forum f if p initiated

its request after forum f was set to *active*.

In one of their recent studies [14], Aravind, and Hesselink compared various versions of *concurrent entering* property introduced in the past. They proposed a new concurrency property called **Simultaneous Acceptance** that would be effective to handle conflicting request and assure a reasonable level of concurrency.

P7. **Simultaneous Acceptance:** Let S be the set of processes that have completed their requests for a forum f, when a process p with interest f enters it's entry section. Then, when p completes its entry section, every process in S enters f within a bounded number of it's own steps.

3.2.3 Fairness Properties

One important area that has been addressed by several works done for this problem is the **Fairness** property. Hadzillacos [4] and Jayanti [6] introduced **First Come First Served (FCFS)** and **First In First Enabled (FIFE)** properties respectively for the group mutual exclusion followed by Aravind and Hesselink [14] introducing the **Forum First Come First Served (F-FCFS)** properties which are as follows:

P8. **FCFS:** If a process p requests a forum before another process q requests for a different forum, then process p will execute it's CS before process q [4].

P9. **FIFE:** If a process p requests a forum before another process q requests for the same forum, and process q enters forum before process p, then process p will enter the forum within bounded number of it's own steps [6].

P10. **F-FCFS:** If forum *f1* is requested before any request for forum *f2* is initiated, then forum *f1* is opened before forum *f2* [14].

In this chapter, we look at the work done in this area for the comparison of existing algorithms. Incorporating insights from a few open areas to be addressed for

this problem, we draw our motivation from work done for this thesis and present the contributions of this thesis.

Chapter 4

Framework - Design and Implementation

In this chapter, we present the first contribution of this thesis, the framework, which is used to evaluate and compare the performance of the group mutual exclusion algorithms in shared memory context. We present the architecture of this framework along with its components. We also define a few terminologies and introduce the parameters that we use in this framework. Finally, we explain the working of the framework and present system primitives.

4.1 Essential features

For an efficient design of our framework, it would be nice to have the following features being supported by our framework:

- Graphical User Interface for ease of usage
- Plotting graphical results extensively from the results generated
- Easy and faster workload generation for executions
- Automated flow control to minimize human efforts

- Modular design to optimize development process and expand scope of using multiple technologies
- scalable enough to accommodate multiple algorithms and performance metrics and add future algorithms and newer metrics

Concurrent Locking [16], which serves as the basis of our framework design, is the environment designed to conduct performance comparison studies for mutual exclusion algorithms. Buhr et al. in 2014 conducted a comparison study of mutual exclusion algorithms [15] by implementing those algorithms into programming codes and running them on multicore machines using this test environment. They even created a repository of algorithms by implementing all the algorithms into program codes. From the features mentioned above, several features like automated flow control and scalability have already been introduced in Concurrent Locking. Having these features as reference considerably reduced our efforts while we worked on including those features into our framework.

4.2 Concurrent Locking

Before we discuss our framework's architecture, we briefly look at Concurrent Locking test framework [16].

4.2.1 System Overview

Concurrent locking is designed to run on the UNIX-based operating system. The codebase of almost the entire environment, including the algorithms, is written in C language. It consists of a centralized controller that manages the flow of execution. This controller channelizes every executing component in the system. It accepts a given number of processes and execution time as parameters along with the algorithm name. Each execution comprises of 5 Runs, each of which lasts for, a time specified. During this period, the generated processes execute the given algorithm's code base in cycles. The traces of execution are stored and updated at the completion of each

process cycle. Computations are performed on these trace values at the end of each run and results are generated as performance metrics. This environment is designed to be used via console mode operation only. It allows a user to run an algorithm individually using a single workload as well as in batches of multiple workloads. It also provides the functionality of automating the entire cycle using shell scripts. Along with individual executions, comparison experiments could also be performed using this automated system controlled by scripts. These scripts when executed, generate workloads, and each algorithm is executed using the generated workloads.

4.2.2 Dependencies

The framework uses a few assembly level instructions like *fetch and add* and *test and swap*. P-threads(POSIX threads) are used in the system for creating a set of competing processes. Also, this system uses C99 standards. Hence, there are a few dependencies on the compiler to support these essentials. We use this execution framework as the basis of our scalable and automated test framework for the group mutual exclusion problem. We have modified this framework to use it for executing algorithms for the group mutual exclusion problem. We will now look at the architecture of our framework.

4.3 Architecture and Components

Our framework is based on the principles of layered architecture, a commonly used pattern for constructing complex software architectures where the overall software is segmented into modules (sections). This approach helps in reducing the complexity of the system along with proportionately distributing functionalities across each module and is a useful technique as changes can be made in one module without affecting the others.

The architecture of our automated test framework, in its modular form, is presented in Figure 4.1. There exists three main section in our framework:

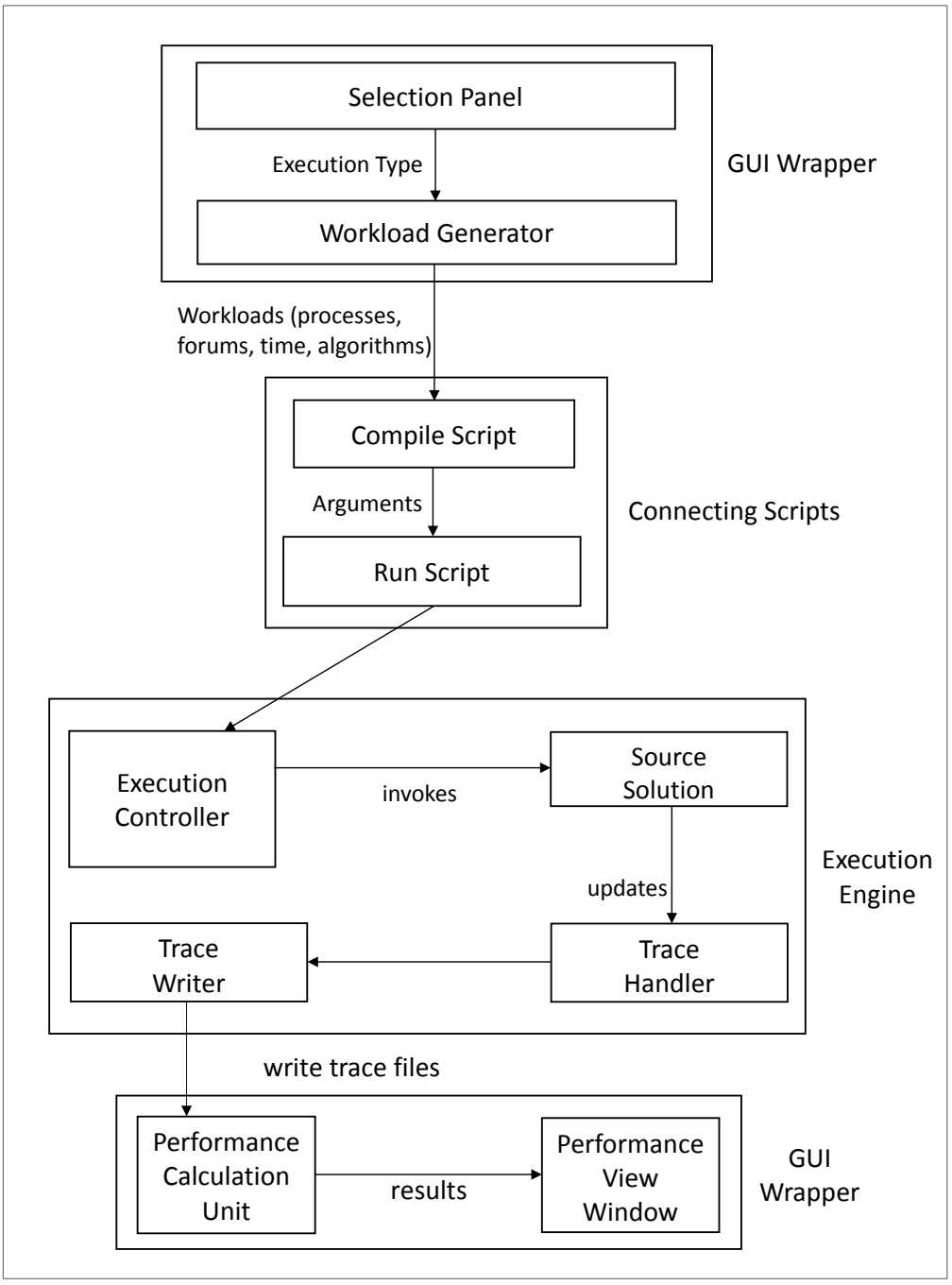

Figure 4.1: Architecture of Scalable and Automated Test Framework

- Execution Engine
- Connecting scripts
- GUI wrapper

Each of these sections contains a few logical components. These logical components control the overall execution of the system. Due to their modular nature, these sections are independent of each other for their respective execution. However, for the overall progress of the system, they are invoked coherently. Understanding the purpose of each section requires us to understand each of its logical components.

4.3.1 Execution Engine

The execution engine the core, or the heart, of our framework. As the name suggests, this part is responsible for the execution of algorithms for group mutual exclusion. All the components of this part are written in C language. This part consists of 4 main logical components: the execution controller, solution source, trace handler and trace Writer.

4.3.1.1 Execution Controller

The execution controller is the most significant component of execution engine as it controls the other three components, Solution Source, Trace Handler and Trace Writer, as it channelizes workflow across all the other components. It keeps track of the overall progress of the system. The following are the key responsibilities of the execution controller:

- define and initialize system primitives like system variables, macros, and functions
- define and initialize thread handler data structures
- invoke the relevant solution source

- initialize the shared variables and data structures in solution source
- assign processor/core to the processes (P-threads) and assign them starting positions
- invoke the trace writer

4.3.1.2 Source Solution

The source solution is the component that holds source code of algorithms implemented into programs, individually into a separate C file. It is in this component where the execution of the algorithm takes place, and the processes cycle through the Entry Section, Critical Section, Exit Section and Non-Critical Section of codes. All the shared variables and data structures and local variables and data structures are defined in this component. In our framework, *five* execution Runs are performed to compare and contrast consistencies and thus, generating better results.

4.3.1.3 Trace Handler

This component stores the traces locally during each run. It primarily comprises of data structures to record values for parameters for performance calculation. Upon each process cycle during each Run (execution), the data structures in the trace handler are written, and the relevant values are recorded. The values for these data structures are recorded before using variables which are strategically placed before and after the CS execution code in each of the source files which helps us achieve standardization for recording the results and ensuring that any overheads resulting from these operations have identical effects on all the algorithms.

Most of the data structures in this component make use of the *Fetch And Add* primitive to update the values. The data structures used here are defined in the execution controller, but their usage is within Source Solution.

4.3.1.4 Trace Writer

The last component of Execution Engine is responsible for writing traces into text files. It fetches the values from the trace handler's data structures and writes those values into the trace files, which are in text format. For each source solution, it maintains a separate trace file for each performance metric used for performance evaluation by using a separate file pointer in C. After this component is successfully executed, the control of execution shifts out of execution engine and transfers to the performance calculation unit in the GUI Wrapper part of our framework.

4.3.2 Connecting scripts

Connecting scripts act as a connection point between the GUI Wrapper and execution engine. This component consists of two Bourne Shell scripts that help automate the process of execution. Bourne Shells are widely used as interactive command interpreters and, thus, are handy to trigger the operation of the execution engine which is written completely in C language.

4.3.2.1 Compile Script

One of the two Bourne Shell scripts is the compile script. This script is invoked by workload generator, a component of GUI-Wrapper. Upon running this script, the number of processes, number of forums, time for each execution, and the algorithm name are the parameters passed to the Compile script.These parameters are recorded by the variables in Compile Script. It is responsible for compiling the file that holds the execution controller and also passes the solution source file name to invoke the right algorithm code. Upon compiling the Execution Controller and Solution Source, it then invokes the Run Script and passes the number of processes, number of forums, and time for each execution as parameters.

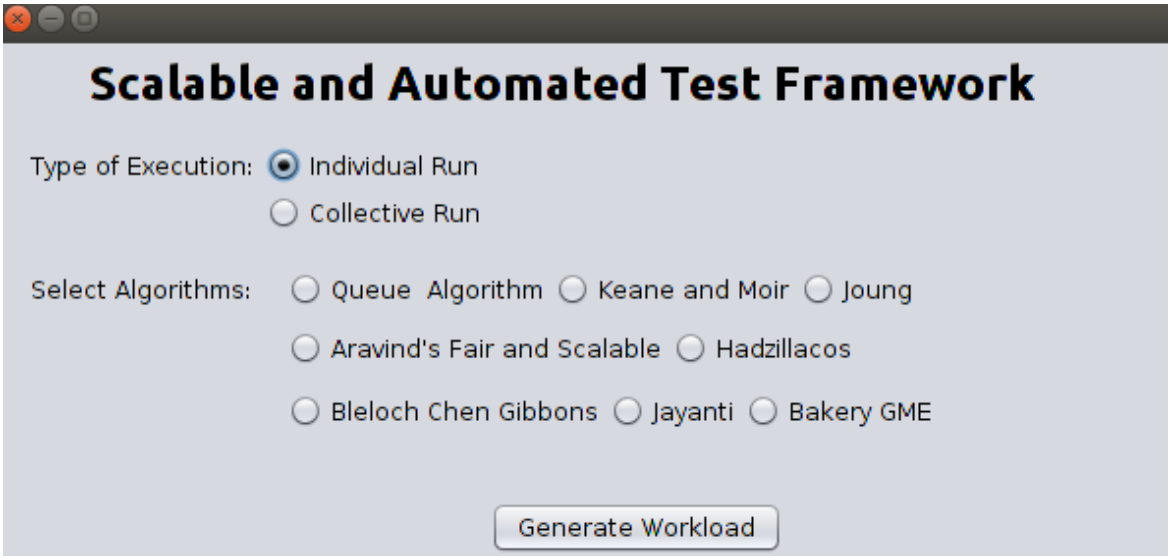

Figure 4.2: Selection Panel window

4.3.2.2 Run Script

Run script is the second Bourne Shell script used in the framework. When the run script is executed, it runs the file that holds the execution handler and the control shifts to Execution Handler component of the execution engine. It passes all the parameters it received, when invoked by the compile script, to the execution controller.

4.3.3 GUI Wrapper

Today, users find it easier to use a software using the Graphical User Interface (GUI) instead of Command Line Interface (CLI). As mentioned earlier, Concurrent Locking framework forms the basis of our framework. Since Concurrent Locking is operated using CLI, we thought of initially adding the GUI wrapper for the ease of use. Eventually, we realized that the GUI Wrapper could be extended to incorporate even the components for performance calculation as well as for viewing results. Thus, we have four components within our GUI. Each of the components is explained below.

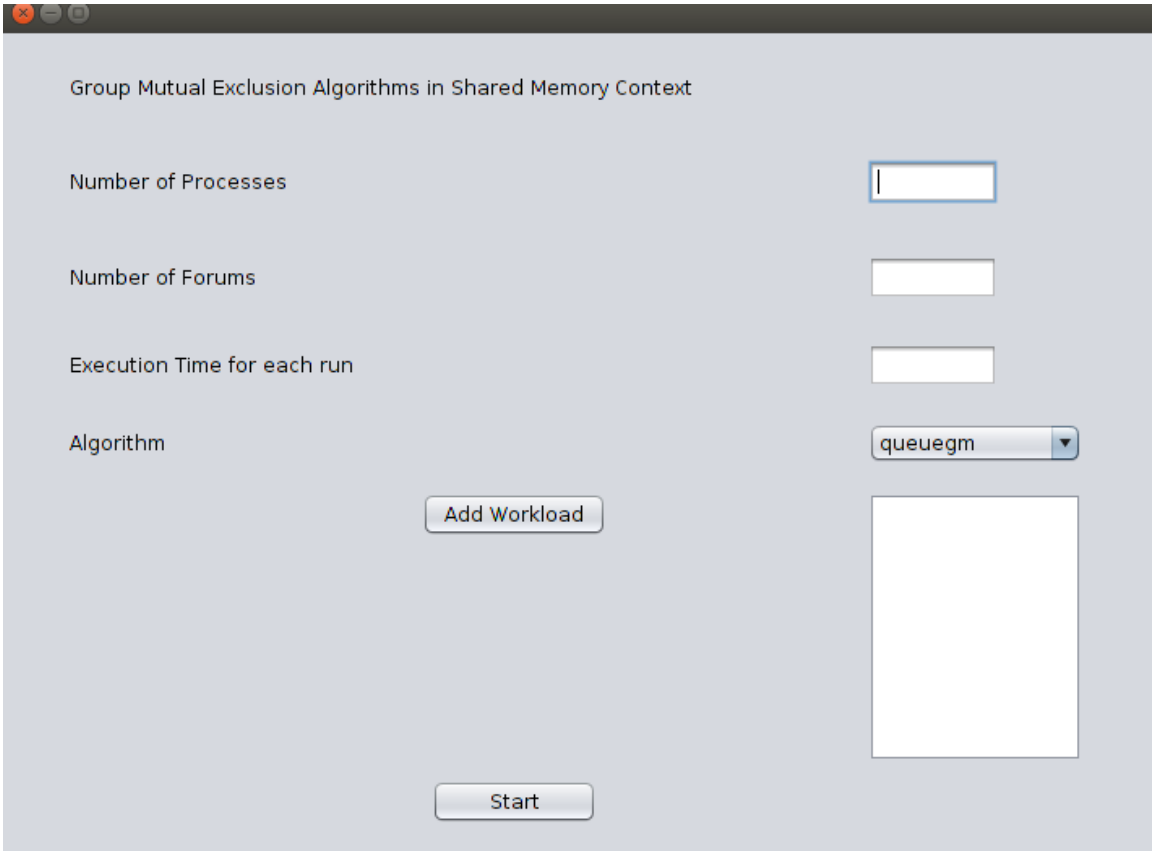

Figure 4.3: Workload Generator Window

4.3.3.1 Selection Panel

In our framework, we provide several execution settings. In the first setting, the algorithms are tested and executed individually, while in the second setting, we can have a comparison of performances of multiple algorithms. In either of the settings, we select the algorithm(s) for execution. This selection is made using the Selection Panel as shown in Figure 4.2.

Upon selecting the choice of setting, we must select the desired algorithm(s). Clicking the *Generate Workload* button navigates us to the Workload Generator window.

4.3.3.2 Workload Generator

When the control reaches the workload generator window, we can generate N workloads consisting of a given set of parameters that are essential for the execution of algorithms. The parameters are the **number of processes, the number of forums and the time for each execution** as seen in Figure 4.3. These parameters are defined in the terminologies mentioned above. Clicking the **'Add Workload'** button creates a new workload consisting the set of parameters. Clicking **'Start'** button initiates Java Processes which are used to invoke the compile script for each workload. The parameters of the workload are passed to the compile script as arguments. The control then shifts to the connecting scripts, which in turn triggers the execution engine. Once the execution is over, the control comes back to the GUI Wrapper and shifts to the performance calculation unit.

4.3.3.3 Performance Calculation Unit

Performance Calculation Unit reads the trace files written by Trace Writer and performs calculations for the performance metrics used for evaluating complexities of the algorithms using different experimentations. After calculating the values for each metric for the algorithm, results are written in Java Array Lists. These lists are used by the Performance View Window to read and present the results. The performance metrics used in our framework and the calculations performed are detailed extensively in the next chapter, and the results of our experimentations and their analysis are presented in Chapter 5.

4.3.3.4 Performance View Window

The performance view window, shown in Figure 4.4, displays the results of the executions in graphical format. If the individual execution setting is selected for execution, then we plot individual graphs for each workload for five Runs(executions) for the algorithm along with plotting mean of all the workloads for each of the per-

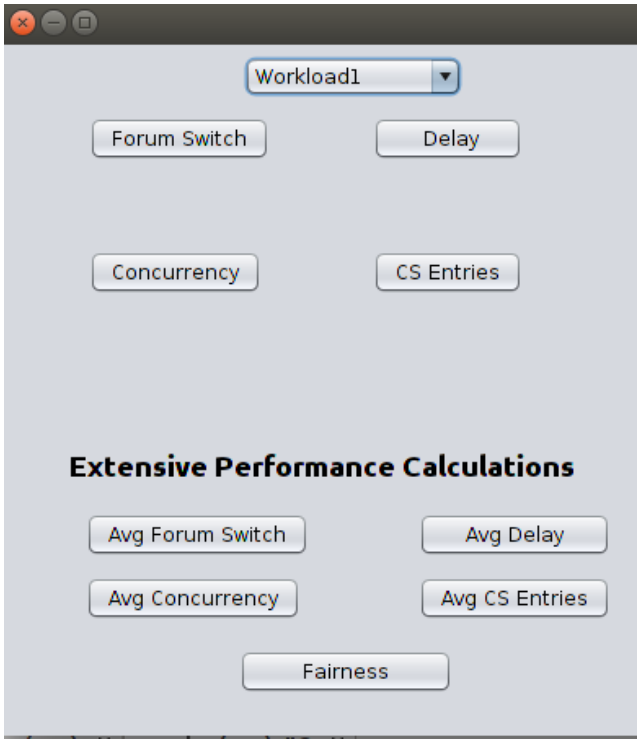

Figure 4.4: Performance View Window

formance metrics. If the comparison of execution setting is selected, then we plot individual graphs of each workload for each algorithm as in the previous case. Along with that, we also plot the mean of all the workloads for each performance metric for all the algorithms selected.

4.4 Framework Design - A unique approach

Our test framework is designed or written, using three different programming languages. Each part discussed above is written in a different language. The GUI Wrapper and its components are written in Java, the Execution Engine, and its components in C, and the connecting scripts are written using shell scripting. We describe our rationale behind doing that.

4.4.1 Advantages and limitations of C

C is one of the most popular programming languages in computer science. It is best to work with when using a lot of lower level instructions, for example, assembly level instructions. Also, when dealing directly with the hardware or accessing machine cores, (as in our case of using multiple cores of the machine), C provides extensive support for many such operations. Thus, rightly so, we retained most aspects of the structure of Concurrent Locking [16], for our Execution Engine. Besides, many of the algorithms make use of higher level operations such as Compare and Swap(CAS), Fetch and Add(FAA), Test and Set(TAS) and other locking and unlocking operations. The C99 compiler used in this framework has these operations predefined which could be used directly.

4.4.2 Automation

To automate the system, we need to compile the execution controller file, which is written in C, for every execution. This automation is easily achieved by using Bourne Shell Scripts for compiling and running C files. However, to have a GUI wrapper for the ease of use, it would require invoking the script using GUI for each execution. That lead to our quest for a higher level programming language where using invoking scripts and having GUI wrapper could be achieved with ease.

4.4.3 Support by java

One of the most widely used programming languages, Java was our answer to the problem mentioned above. Designing GUI platforms using Java is not a tedious task. Efficient GUI frames and a plethora of libraries at our disposal, for plotting clean graphs certainly helps Java gain an edge over the other languages or platforms.

Java Processes offer more support than what we assumed. Using Java Processes (not to confuse it for Threads in Java), we can compile, run and access almost all the possible files. Also, we discovered one of the easiest ways to invoke and run

shell scripts using Java and in turn compile and run C code using shell scripts. This approach led to us formalizing a new approach of designing cross-platform frameworks eliminating language dependencies.

4.5 System Primitives

Our framework has a lot of system primitives retained from the Concurrent Locking framework for mutual exclusion algorithms [16]. In this section, we discuss the important primitives in our framework. Some of the primitives are critical to the state change of our framework. On the other hand, other primitives help to ensure the correctness of the algorithms and replicating close to exact if not the exact behavior of the algorithm to preserve the essence they hold. The key system primitives of our framework are as follows:

1. **Self-checking Critical Section:** It is tedious to check for the correctness of an algorithm programmatically and to verify whether all the properties that are claimed to be true are satisfied in the true sense. Having a self-checking critical section for group mutual exclusion Algorithms assures of checking for the *mutual exclusion* property where if there exist processes executing their CS while having requested for different forums, exit(0) will be executed immediately, and system run will be terminated instantly.

   The shared variables *CurrTid* and *CurrFid* in our system hold the id of the thread and the form number requested by it. CurrFid is initialized at the beginning to active forum's id. A thread then loops 100 times pretending to perform the CS, but at the end of each iteration, the thread compares its forum id with the one in CurrFid for any change. If there is a change, mutual exclusion principle of the algorithm has been violated, and the program is stopped.

2. **volatile:** All algorithms for group mutual exclusion have one or more wait sections in their Entry Section which are executed when competing for accessing

the shared resource (executing CS) or waiting for a thread to exit the CS. These wait sections are infinite loops reading one or more shared variables, and a process spins in these loops until the other competing process stops competing for CS. Concurrency allows the value of that shared variable to be changed by another process.Due to the compile time reordering, the process sometimes is unable to see the value of the shared variable updated by some other process. To make this waiting loop work, the value of the shared variable must be updated on each iteration. The technique to achieve this explicit loading in C is to qualify a variable's type with volatile. All shared variables used in the Entry Section and Exit Section code for the algorithms are declared with the volatile qualifier to prevent this problem.

3. **Fence():** In cases of compile time reordering, the processor sometimes reorders the load operations before the store operations. Hence, write operations may be performed before the read operations. For example, for two competing processes, if they both perform the write operation before the read operation then they both might end up executing their CS simultaneously. Such activity will violate the property of mutual exclusion if the two processes executing their CS codes have requested for different forums. Hence, we insert Fence() between read-write operations to prevent the compiler in the processor to reorder those operations. Also, since the volatile qualifier in the C99 standard that we use in our framework is not as robust as Java, manually inserting memory fences are essential.

   In a sequential program, this optimization is benign because of the single thread; in a concurrent program, this optimization must be precluded for correctness. For all the algorithms in our work, the minimal number of store/load fence instructions are inserted to ensure correctness but allow maximum performance. Platforms with weaker memory models such as ARM or power-PC may need additional fences (we use x86 standard architecture based machine for our work). While the fence instructions prevent reordering of access to lock data, additional

fence instructions may be required to prevent reordering of application data into or outside of the protecting mutual exclusion component of the algorithm.

4. **Pause():** The waiting section in algorithms can hamper their performance as processes keep looping around checking for the change in the value of the shared variable(s). As this loop executes, the processor pipeline fills with load/compare instructions, which takes resources (CPU), space (instruction cache), and power (heat). To alleviate such occurrences, we use a pause instruction specifically for this situation. A Pause() instruction is primarily used to provide the processor with a temporary delay, which in turn could be used to flush the pipeline of compare instructions. Their usage is vital, especially for smaller waiting loops.

5. **Atomic Operations**

   All group mutual exclusion algorithms make use of at least one atomic operation. Apart from the few initial algorithms, most of the newer algorithms (deduced in past decade or so) make use of higher level atomic operations. These instructions are used to perform read-write operations, primarily essential to communicate a process's intent to other processes within the system. Each of the instructions used could be formulated manually. However, the GCC compiler provides those instructions which could be used to interact directly with the hardware.

   These instructions guarantee a read, an arbitrary action, and writes occur atomically. For example, a test-and-set instruction reads a value, writes a marker value, and provides the original value read; no interruption can occur during the read/write sequence. A fetch-and-increment instruction reads a value, increments the value, writes the incremented value, and provides the original value read; no interruption can occur during the three actions. For atomic instructions, the hardware controls execution order, that is, precluding interruption between the read and write, which is impossible with software solutions. While it is interesting that these instructions can be created without any atomic assistance from the hardware, memory instructions are still needed to handle races on shared variables.

4.6 Summary

In this chapter, we presented our scalable and automated test framework for evaluating and comparing performance complexities of algorithms for the group mutual exclusion problem in the shared memory context. We explained the architecture of our framework and described all its components. We also discussed and described our approach in designing the framework using multiple languages. In the next chapter, we present the performance metrics used for the performance studies. We also list the algorithms that we implemented and the updates we made while implementing them as source codes to overcome systems limitations and improve the performance.

Chapter 5

Experimentation Results

In this chapter, we discuss our experiments conducted for performance studies with their results. We also present the experiments that we repeated from the performance experiments from the literature in order to compare the outcomes of those experiments in our framework. Lastly, we provide the results of those experiments along with the similarities and differences observed in the original work.

For our performance experiments, we use Dell PowerEdge FC630 rectified blade server controlled using Integrated Dell Remote Access Controller (iDRAC8). The machine comes equipped with two chipsets of Intel(R) Xenon(R) CPU E5-2650 v3, each comprising of 10 cores (20 threads) and operating at the speed of 2.30GHz. We have Ubuntu 14.04 LTS operating system running on our machine.

5.1 Past Experimentations

In this section, we present an overview of experimentations performed by Keane and Moir [2] and Blelloch, Cheng, and Gibbons [10] in their respective studies. We observe these experiments in the later part of this chapter and present results of a few of these experiments conducted using our framework.

5.1.1 Keane and Moir

Keane and Moir performed **five** performance experiments to compare the performance of their algorithm with Joung's [1] algorithm. They performed experiments with varying the underlying Mutual Exclusion Algorithms using Young and Anderson [20] and MCS [12]. Their experiment aimed at examining the performance of algorithms acting as mutual exclusion algorithms and evaluating the performance of each algorithm based on the cycles required for them to attain 1000 CS Entries for each process. Their second experiment involved having only one active process from a set of 32 processes where they tested for the algorithm based on the simulation cycles taken to attain 1000 CS Entries. Their third experiment examined performance under varying levels of concurrency where each process selected a forum number ranging from 1...M. They used 32 processes for the experiment with each process cycling 1000 times to attain the CS. Their fourth experiment involved processes performing some task inside of the CS, unlike previous experiments where processes solely entered and exited CS. For this experiment they had all processes requesting the same forum, hence eliminating any waiting time for all processes. Their final experiment tested the performance of their algorithm when modified to define a threshold, which is the amount of potential concurrency attainable by their algorithm. For this experiment, they used 32 processes, each of which randomly selects one of two available forums.

5.1.2 Blelloch, Cheng, and Gibbons

Blelloch, Cheng, and Gibbons [10] performed two sets of experiments. In the first set, the experiments conducted compared the performance of their algorithm with Keane and Moir's algorithm [2]. Their second set of experiments gave timings for an implementation of a shared work stack using their algorithm. We are interested in observing their first set of experiments.

They implemented Keane and Moir's algorithm using MCS locks [11] as suggested by Keane and Moir for their experiments. The experiments involved the varying

number of processes, the amount of work done inside and outside of CS and the ratio of requests for the two forums that they used for their experiment. For all their experiments they use 1-32 processes. They performed experiments using low load and high load settings for experiments involving varying amounts of work done by processes. In the low load setting, the processes solely entered and exited CS while accessing it and they performed no work when inside the CS. For high load setting, the processes performed significant work, roughly equal work when inside CS as when executing outside of the CS. The work done inside the room was selected based on Gaussian distribution with the mean equal to work done outside of CS. For varying the ratio of selection of one of the two forums, they achieved that with two settings. In the first setting, the probability of selecting each forum was p = 0.5 while for the second setting, one of the forums was given a probability of p=0.1 which meant the other forum had a probability of p=0.9 for being selected. For all the experiments in the first set, they use n =1000 rounds as a measure of duration for each experiment to be performed.

5.2 Our Approach

Since our experiments are performed to evaluate performances for algorithms proposed for the group mutual exclusion problem, we exemplify a few experiment techniques from Keane and Moir and Blelloch, Cheng, and Gibbons to design our experiments. However, as our framework is based on the mutual exclusion test package designed by Buhr et al. in [16], we draw significant inspiration from their ways to calculate performance results.

We perform several experiments with varying the number of processes, forums and time for each execution. Unlike Keane and Moir and Blelloch, Cheng, and Gibbons, whose majority of the experiments involved experiments duration to be equal to the time elapsed for each execution so as to ensure that each process in the system attains 1000 CS Entries, our experiments are based on the time specified in a given workload as a measure to run an experiment. We rely on a few performance metrics

for evaluating the performances of algorithms in our experiments.

5.2.1 Performance Metrics

For results, we use CS Entries attained collectively by all processes in the system as the basis of our calculations and evaluate performances of algorithms based on several performance metrics. We do so, as Buhr et al. [16] perform their calculations and deduct results following the same approach. Buhr et al. evaluated the performance of mutual exclusion algorithms based on two metrics:

1. **CS Entries** which is the aggregate CS Entries attained by all processes collectively in a given workload during execution.

2. **Fairness** which is the relative standard deviation from the mean of CS Entries achieved by all the processes collectively from all the executions. (As mentioned in Chapter 4, we use **five** Runs for each experiment execution.) It is denoted as a percentage of the coefficient of variation, which is a normalized measure of dispersion of fairness for each algorithm. The more the CS Entry counts differ for each run for a given workload, the higher is the percentage of unfairness.

To provide a considerable evaluation, we use two more metrics to calculate complexities of algorithms.

3. **Delay** which is the maximum number of CS entries that can occur while a process is waiting to enter the CS

4 **Forum switches** which is the maximum number of times that forums can change while a process is waiting to enter the CS.

Joung and Hadzilacos briefly discussed these metrics in [1] and [4] respectively. Hadzilacos proposed **delay** to be measured using the number of CS entries that occur before a process's request to access CS is granted and forum switches as the total number of forums that are activated before a forum requested by a given process

is set to active. To put **delay and forum switches** to use as performance metrics, we calculate them based on the actual entries accounted. We retain the technique proposed by Hadzillacos to calculate forum switches for our work. However, we believe, that delay for a given process could be best recorded by calculating the actual wait time for that process before it accesses CS. Hence, we record wait time in milliseconds for each process cycle. All our records generated are aggregated before plotting the results as we have multiple runs for each execution and multiple cycles for each run.

5.2.2 Implementation of algorithms

For our performance studies, we implemented nine algorithms from the available group mutual exclusion algorithms. Our algorithms primarily are divided into two categories, FCFS based algorithms and non-FCFS based algorithms. As discussed in Chapter 3, fairness has been a key factor for newer algorithms that have been proposed for the group mutual exclusion problem. FCFS based algorithms ensure that the order of request for executing CS by the processes is retained when the requesting processes execute the CS code. On the other hand, non-FCFS based algorithms aim to achieve maximum performance irrespective of the order of execution of CS by the requesting processes.

We have implemented Hadzilacos's [4], Aravind's bakery group mutual exclusion [9], Jayanti et al.'s [6] and Aravind's fair and scalable (2 variants) [14] algorithms under the category of FCFS based algorithms. For non-FCFS based algorithms, we have implemented Joung's [1], Keane and Moir's [2], Aravind's queue [17] and Blelloch, Cheng, and Gibbons's room synchronization [10] algorithms.

5.3 Performance Experiments

Our performance study involves **four** experiments to be executed on our automated test framework. In each of these experiments, we generate workloads by varying one or more parameters. Each algorithm executes for **five** runs for a given workload.

No.	Experiment	Static Parameter(s)	Resulting Graphs
1.	Varying Processes (Processes: 1 - 32)	Time = 2 secs, Forums = 2	1. Average CS Entries
2.	Varying Forum (Forums: 1 - 32)	Processes = 32, Time = 2 secs	2. Average Forum Switches
3.	Varying Exec. Time (Time: 1 - 5 secs)	Processes = 32, Forums = 2	3. Average Delay
4.	Mutual Exclusion (Processes: 1 - 32, Forums: 1 - 32)	Time = 2 secs	4. Fairness

Table 5.1: Summary of performance experiments

All the algorithms execute sequentially using each workload generated in isolation to utilize the machine capacity to the fullest. Traces are recorded, and the results are calculated, and subsequently plotted graphically from these traces.

We present each of our study experiments with their results and identify the trends observed while executing the algorithms. Our results show performances of the algorithms for each of our performance calculation metric for each experiment. Because we are dealing with the concurrent system, the traces generated are dependent on the performance of the machine, and hence some variations are encountered with each new execution. For our calculations, we use the raw data recorded as traces without processing them. Also, we do not use any smoothening tools for our graphical results. We aim to plot the actual values in their raw form resulting in occasional spikes in our results. Such spikes are observed even in the performance experiments conducted by Buhr et al. for mutual exclusion algorithms [16]. Besides, some of the algorithms demonstrate a rise in performance after a certain number of executions during several of 100s of our executions for each experiment. These rises are nothing but downward spikes encountered in previous execution and hence must not be mistaken for

an anomaly in the performance or behavior of the algorithms.

5.3.1 Varying Processes

For this experiment set, we vary the number of processes from 2 to 32, keeping the number of forums constant at 2 and execution time for each run at 2 seconds. Hence, 31 workloads are generated with increasing number of processes, and the domain axis represents an increasing number of processes for each graph in this experiment.

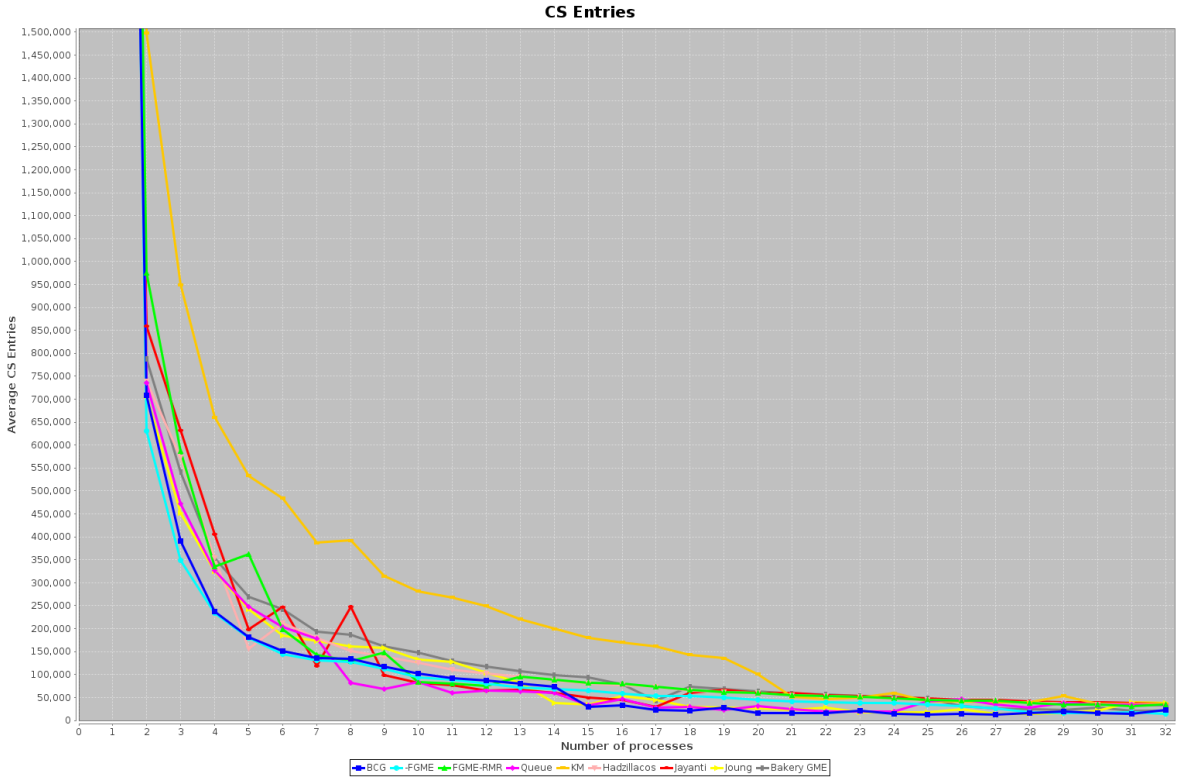

Figure 5.1: Critical Section Entries for varying processes experiment

With an increase in the number of processes, there is a rise in contention among them, and thus, the number of CS entries attained by processes decrease with each subsequent workload. Hence, we witness exponentially decreasing graphs for each of the algorithms. It is observed that for workloads with the lower number of processes (less than 16) algorithms like Keane and Moir and Joung with simple read-write

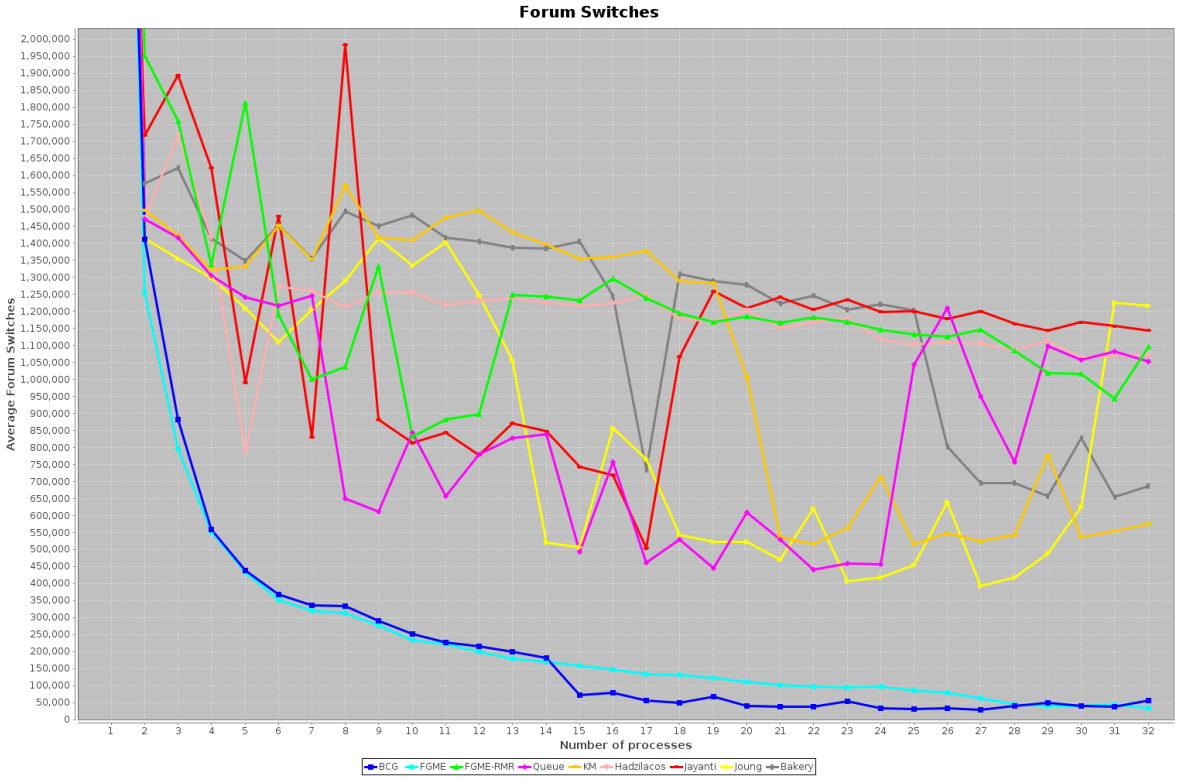

Figure 5.2: Forum switches for varying processes experiment

operations tend to achieve a higher number of critical section entries in comparison to algorithms with higher level read write operations (most of the FCFS based algorithms).

This observation contradicts the theoretical time complexities for each of the algorithms. We assume such behavior is observed because the algorithms using higher level read write operations like Compare And Swap, Fetch And Add and Test And Set which are expensive operations, take more steps in achieving CS entries that algorithms using simple locking mechanisms to perform read-write operations. However, with an increase in contention, algorithms with higher level read write operations, which are mostly the FCFS based algorithms, tend to perform better than algorithms with simple read-write operations. Interestingly, similar observations were made in [16] when Buhr et al. performed performance experiment for comparing mutual exclusion algorithms. For their experiments, many of the software solutions performed better than the hardware solutions as they initiated their work believing

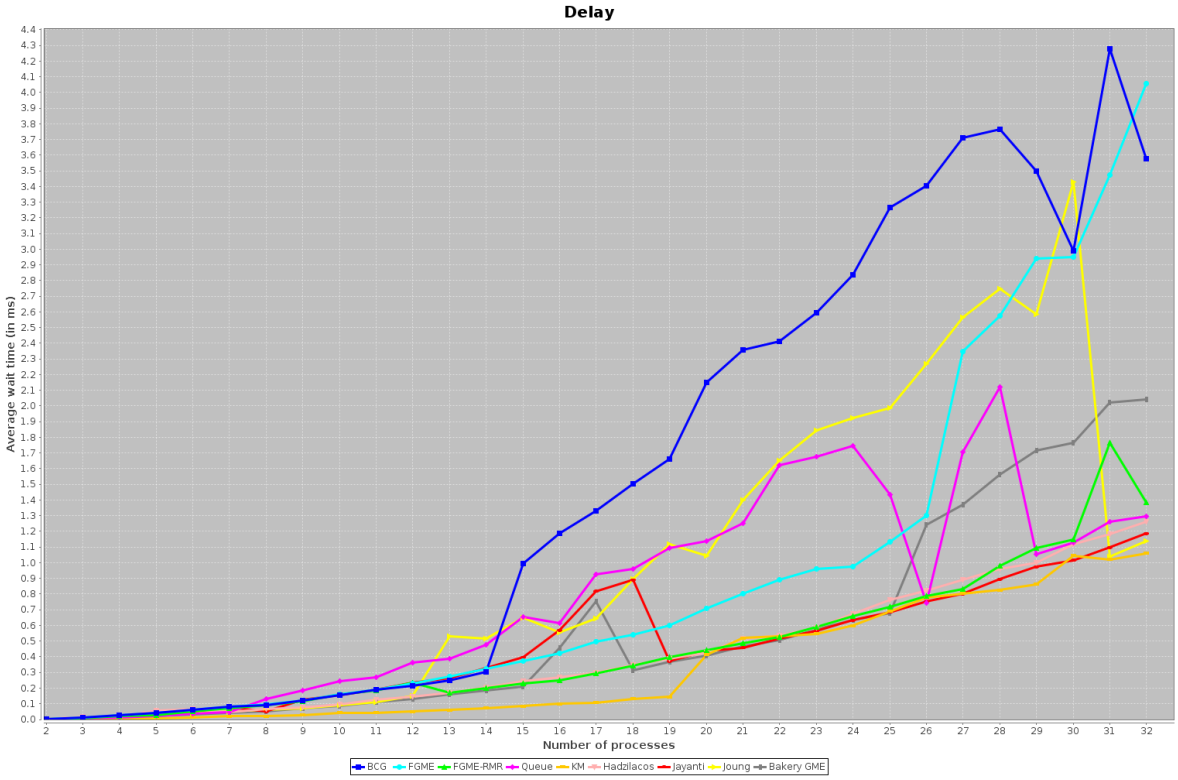

Figure 5.3: Delay for varying processes experiment

that software solutions would be an order of magnitude slower than the hardware solutions.

Forum switches for this experiment show a decreasing exponential curve for all the algorithms in most of the executions we performed. With an increase in contention among processes, CS entries decrease and the wait time increases. Hence, fewer forum switches are observed with increase in the number of processes. However, we do not witness a steep decrease in forum switches for most of the algorithms like that of CS entries. It is because, forum switches is a collective calculation for the processes where a single forum switch could account for one or more processes in an active forum, unlike CS entries where each of the entries is individually calculated for each of the processes, irrespective of the forum they choose.

While observing the delay which is the average time each process waits for its CS entry, the trend witnessed compliments the results of the number of CS entries.

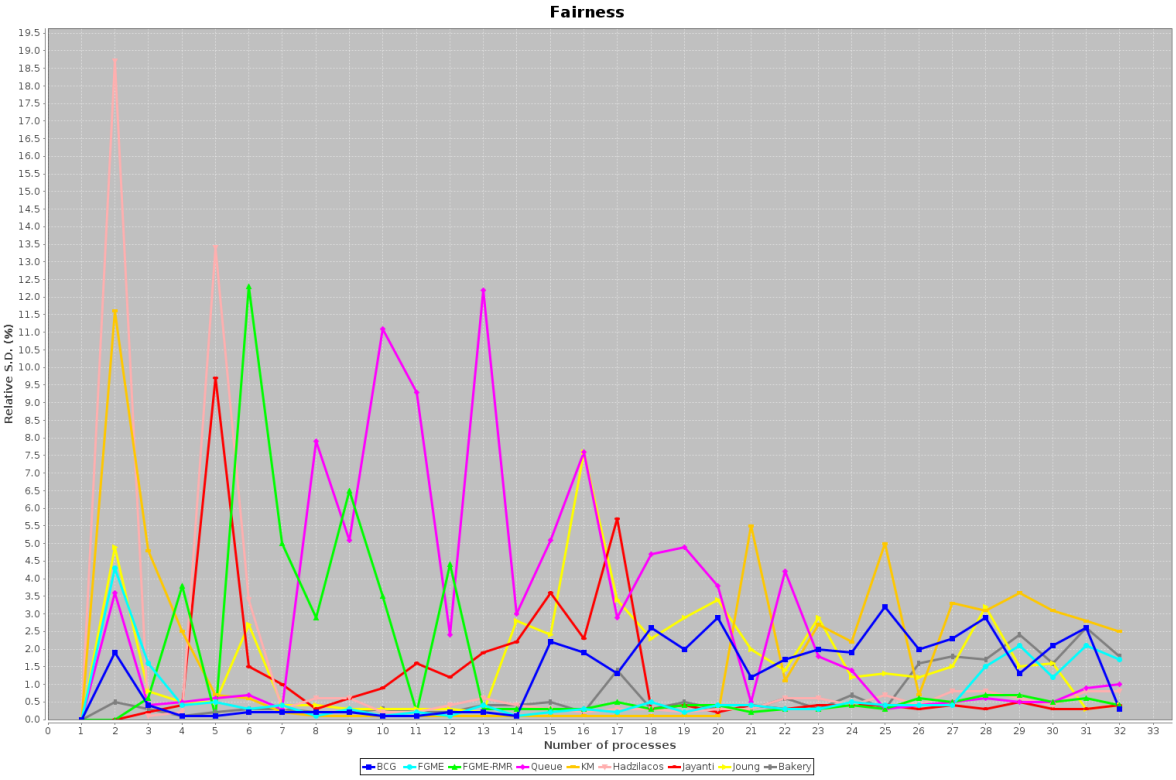

Figure 5.4: Fairness for varying processes experiment

With an increase in the number of processes and subsequent rise in contention the wait time for each process keeps increasing. Hence we witness increasing exponential graphs for the wait times for all the algorithms. For this experiment, algorithms with simple read-write operations, which are non-FCFS based algorithms tend to achieve less wait time than the FCFS based group mutual exclusion algorithms.

For this experiment, it has been observed that non-FCFS based algorithms using simple read-write operations show better performance than the other algorithms. However, FCFS based algorithms show better performance for this metric. These algorithms attain much lesser spikes and achieve fewer deviations which result in better fairness for them with many of them achieving deviation under **2%** for most of the workloads.

5.3.2 Varying Forums

For this experiment set, we vary the number of forums from 1 to 32, keeping the number of processes constant at 32 and execution time for each run at 2 seconds. Hence, workloads are generated with increasing number of forums, and the domain axis represents an increasing number of forums for each graph in this experiment. Processes experience maximum contention for accessing CS in this experiment. Though there is no rise in contention among processes, there is an increase in distribution of processes. With each new workload, they get to choose from a bigger number of forums. Thus, the number of CS entries attained by processes decrease with each subsequent workload as it takes longer for a forum to turn active with each new workload. Hence, we witness exponentially decreasing graphs of CS entries for each of the algorithms.

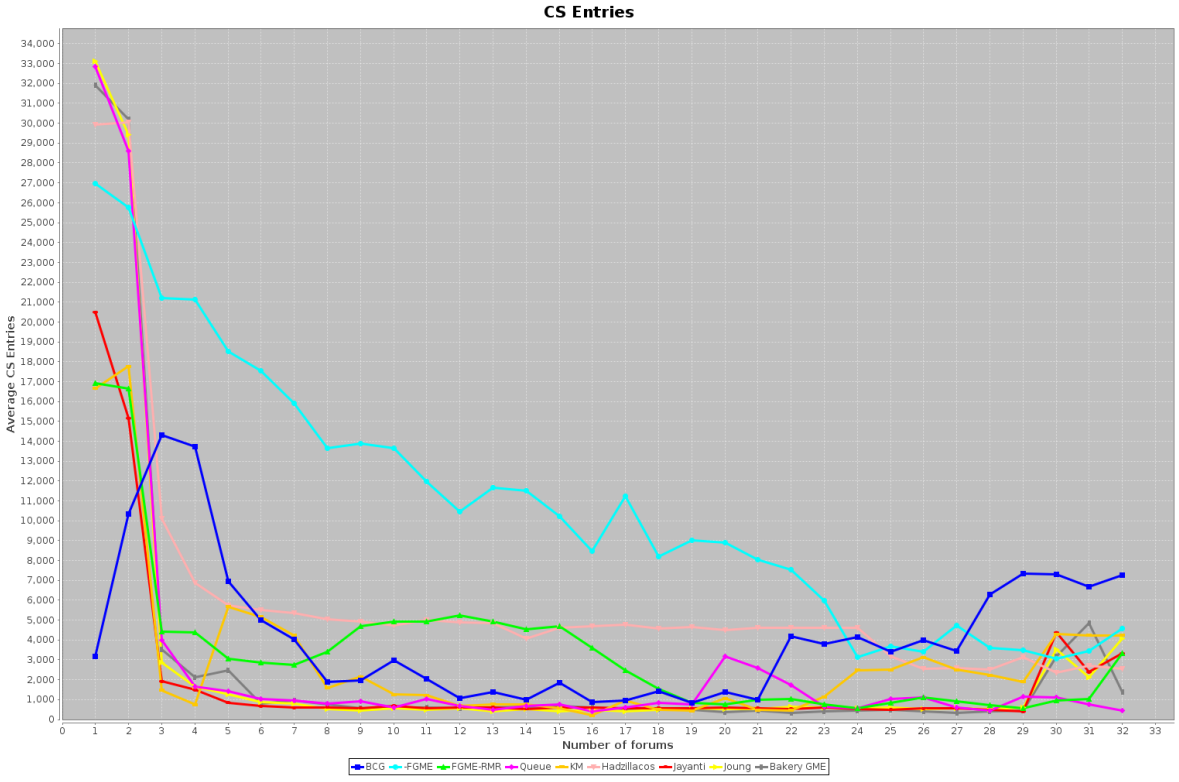

Figure 5.5: Critical Section entries for varying forums experiment

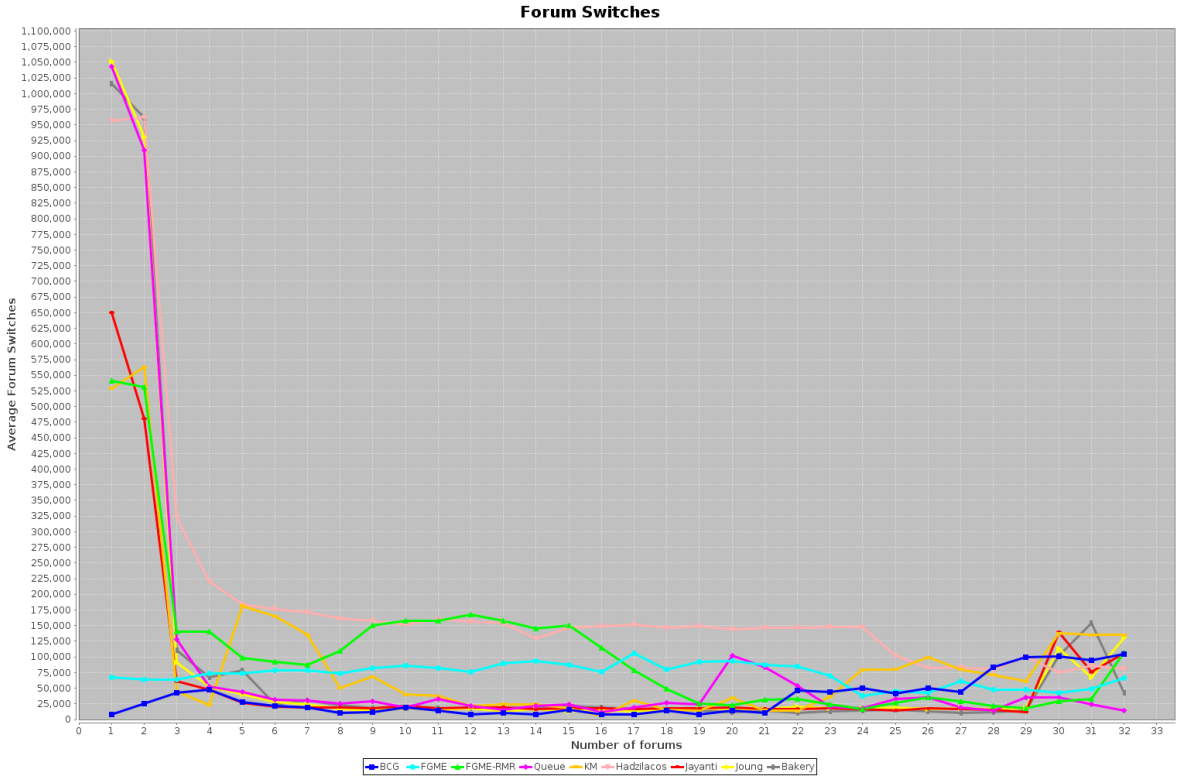

Figure 5.6: Forum switches for varying forums experiment

Most of the algorithms tend to achieve higher CS entries for workloads with less than three forums. However, it is observed that throughout the experiment, algorithms using higher level read write operations tend to achieve more CS entries despite high contention. This observation strengthens the claims made by several FCFS based algorithms to produce better performance with a higher number of processes. We assume such behavior is observed because, under non-FCFS based algorithms, some of the processes might have to wait longer, thus reducing the total number of CS entries attained collectively by them.

An interesting observation that was made throughout our experiments was that Blelloch, Cheng, and Gibbons's algorithm tend to achieve a higher number of CS entries while Aravind's Fair Group Mutual Exclusion Algorithm with constant RMR shows a decline in performance with the higher number of forums (usually for forums greater than 16). We believe the increase in performance for Blelloch, Cheng, and Gibbons's algorithm is because of the underlying round robin nature of the algo-

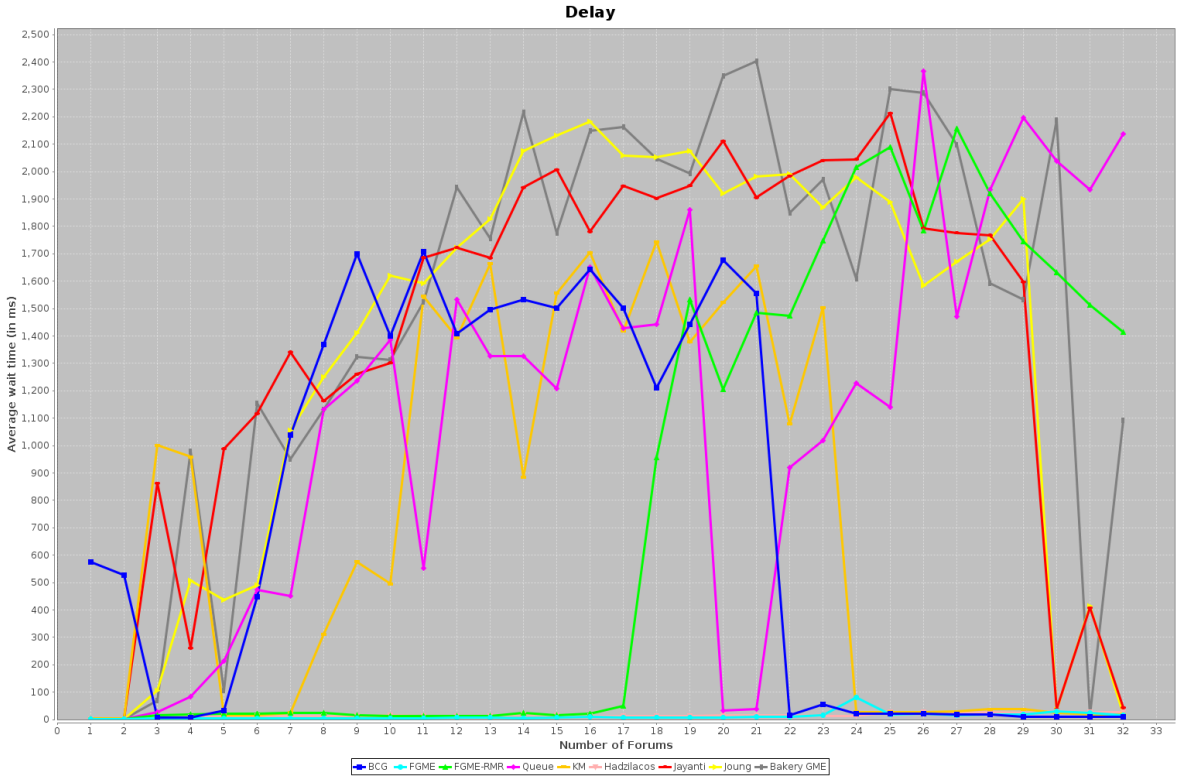

Figure 5.7: Delay for varying forums experiment

rithm. On the other hand, the change in forums has minimal impact on Hadzilacos's algorithm as it barely reflects deviation in the CS entries attained by the processes. Aravind's Fair and Scalable Group Mutual Exclusion Algorithm demonstrates an exponential decrease in CS entries with an increase in the number of forums yet achieves more CS entries than all the other algorithms for most of the workloads.

For workloads with more than two forums, almost all the algorithms show minimal deviation in the number of forum switches attained. The only algorithms that show substantial changes are Aravind's Fair Group Mutual Exclusion Algorithm with constant RMR and Blelloch, Cheng, and Gibbon's algorithm. Aravind's algorithm with constant RMR shows a dip in the number of forum switches while Blelloch, Cheng, and Gibbons's algorithms achieve more forum switches with a higher number of forums. Aravind's Fair Group Mutual Exclusion Algorithm achieves one of the highest forum switches and with minimal deviation. Similar to the CS entries, non-FCFS algorithms using simple read-write operations achieve lesser forum switches than the

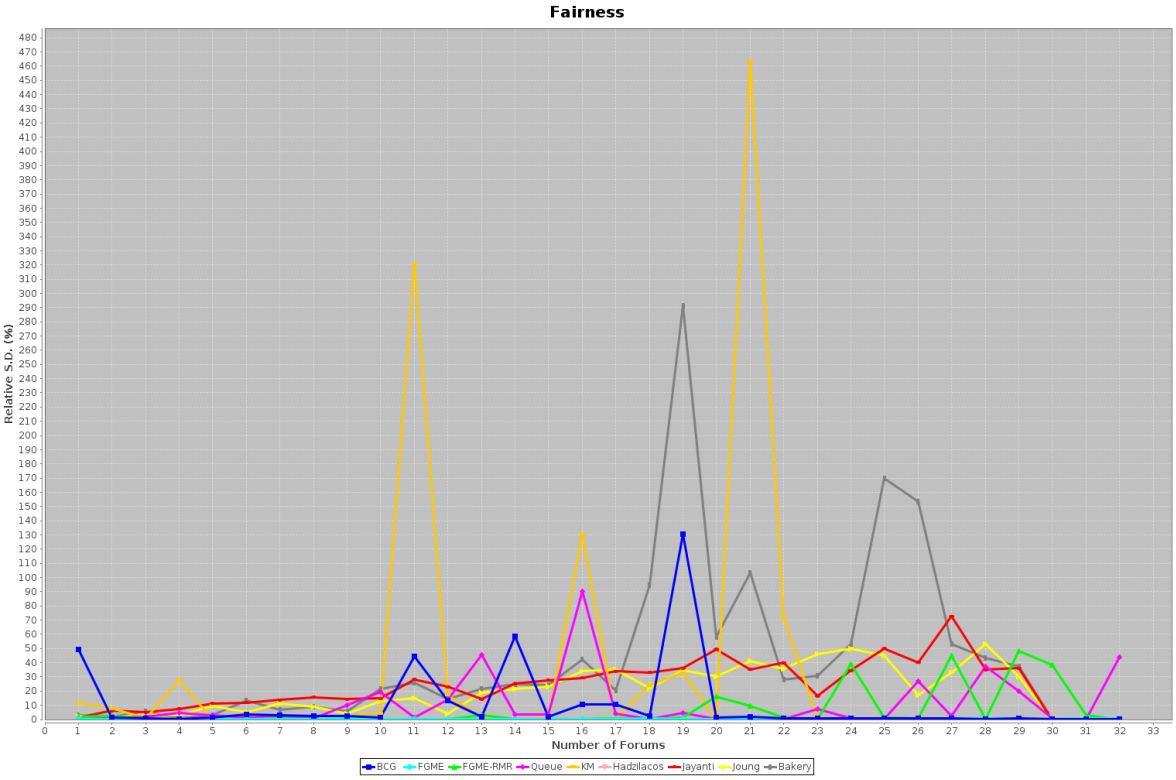

Figure 5.8: Fairness for varying forums experiment

FCFS based algorithms and the ones using higher level instructions for read-write operations.

While observing the delay which is the average time each process waits for its CS entry, the trend witnessed compliments the results of the number of CS entries. With an increase in the number of processes and subsequent rise in contention the wait time for each process keeps increasing. Hence we witnessed increasing exponential graphs for the wait times for all the algorithms. For this experiment, algorithms with simple read-write operations, which are non-FCFS based algorithms tend to achieve less wait time than the FCFS based group mutual exclusion algorithms which is consistent with the results for CS entries. For fairness, non-FCFS based algorithms using simple read-write operations, showed better performance than the FCFS algorithms. However, FCFS based algorithms show better performance for this metric. These algorithms attained fewer spikes and achieved fewer deviations which result in better fairness with many of them achieving deviation under **2%** for most workloads.

5.3.3 Varying Time

For this experiment set, we vary the time of execution from 1 to 5 seconds for each run, keeping the number of processes and number of forums constant at 32 and two respectively. Hence, workloads are generated with increasing number of seconds for execution time, and that forms the domain axis for each graph in this experiment. Processes experience maximum contention for accessing CS in this experiment.

In the first result, we observe the number of CS entries attained by each algorithm for the given workloads. Though there is no rise in contention among processes with each new workload, with an increase in time for each execution, there is scope for attaining more CS entries. Thus, the number of CS entries achieved by processes increase with each subsequent workload, and we witness almost linearly increasing graphs of CS entries for each of the algorithms. Most of the algorithms tend to perform consistently for all the workloads. For workloads with smaller execution time, Aravind's Fair and Scalable group mutual exclusion with RMR and Bakery group mutual exclusion algorithms almost always achieved highest CS entries along with Keane and Moir's and Joung's algorithms. Throughout the experiment, non FCFS algorithms tend to outperform many FCFS based algorithms.

For forum switches, the performance observed is usually consistent with the results observed with CS entries. Intuitively, algorithms achieving higher CS entries should have higher forum switches and less delay. However, for executions where we do not observe this to hold entirely true for forum switches, its performance observed in delay compensates for the balance. For example, Keane and Moir's algorithm tend to achieve much higher CS entries than many other algorithms in the above graph. But it achieves substantially lesser forum switches in Figure 5.10. However, it is compensated with a higher performance in delay which will be discussed shortly. The same implies for any dip or rise in performance observed for any given algorithm while executing a particular workload. The dips and rises are due to the system's performance and are not due to any specific behavior of the algorithms as they are

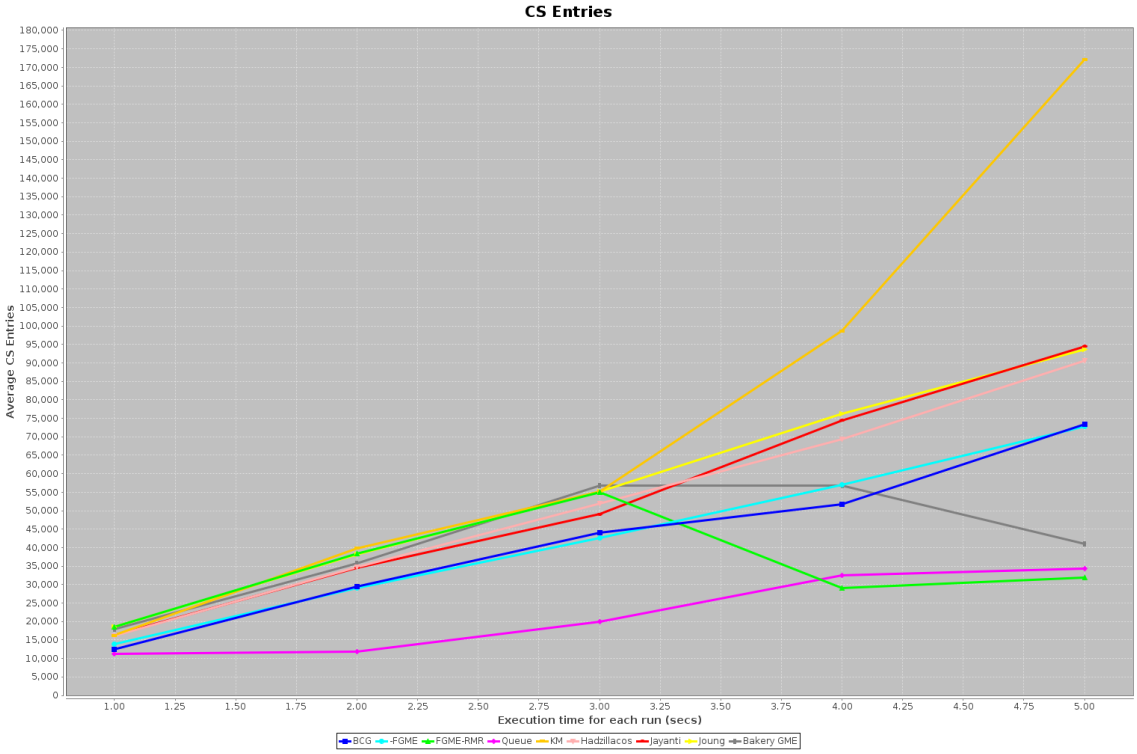

Figure 5.9: Critical Section entries for varying time experiment

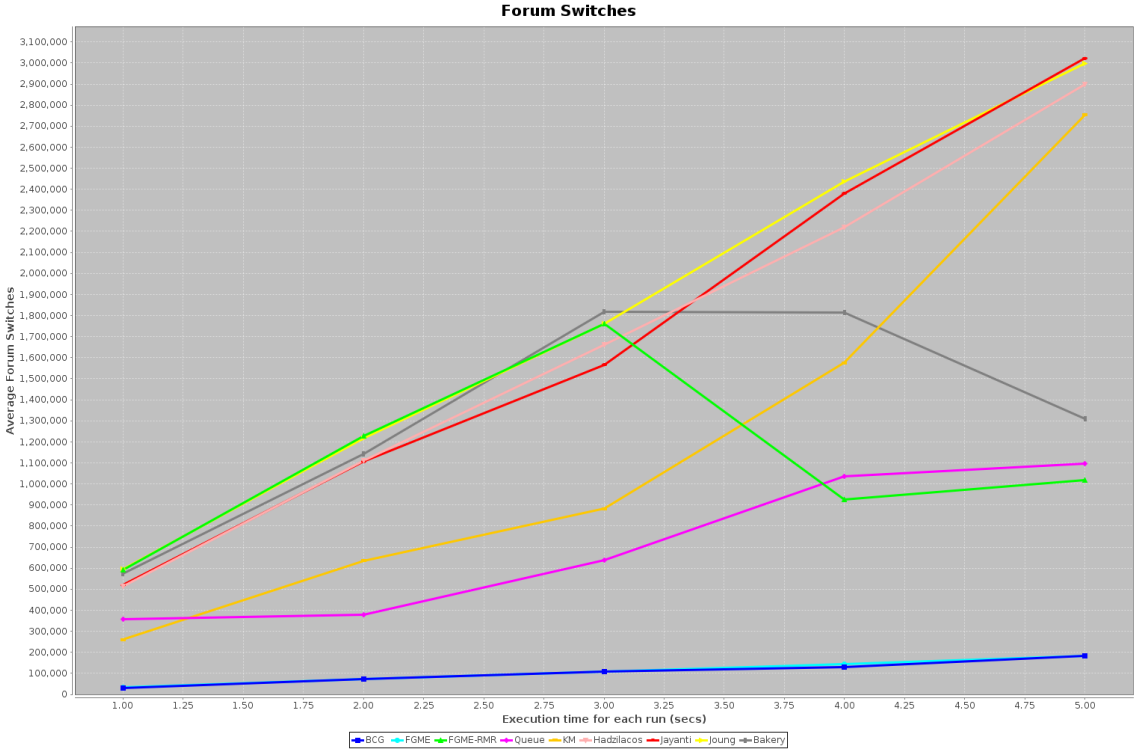

Figure 5.10: Forum switches for varying time experiment

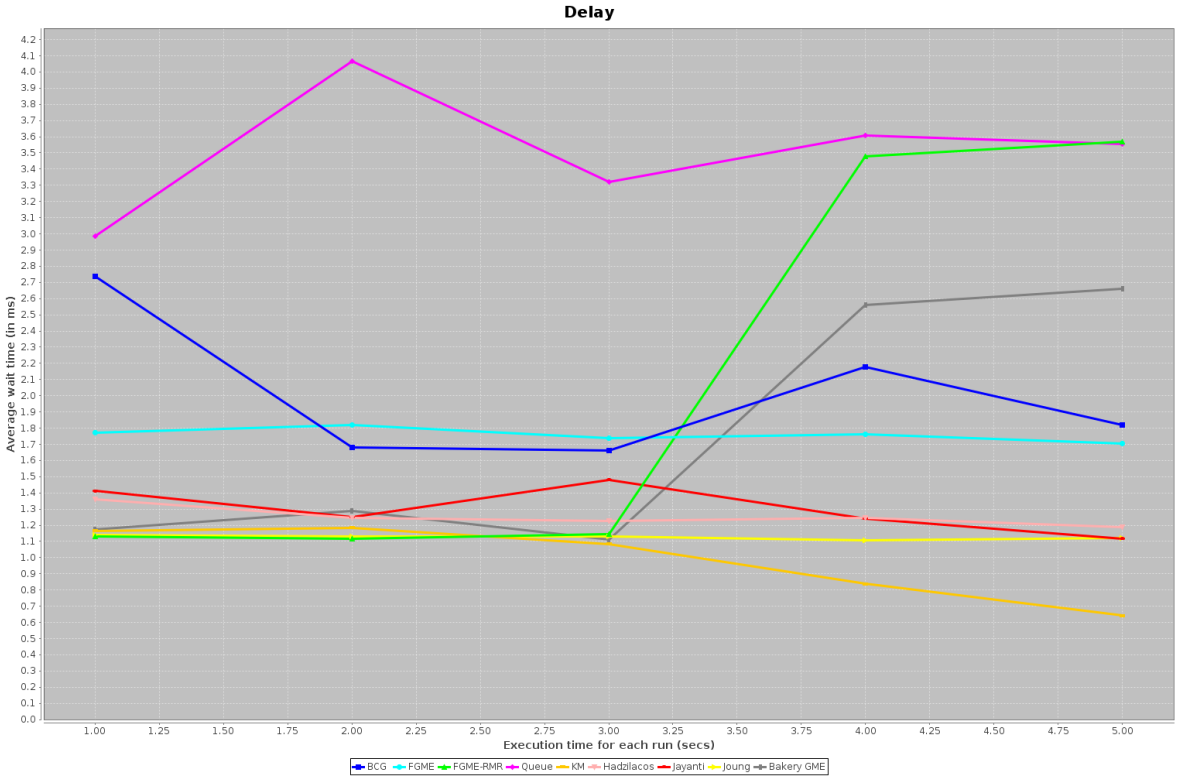

Figure 5.11: Delay for varying time experiment

not observed consistently throughout multiple executions. Such behavior is normal and is observed through our experiments considering the concurrent nature of the algorithms and system latency for their performance.

While observing the delay, the trend witnessed compliments the results of the number of CS entries and forum switches. Though we observe an increase in the number of seconds for each run, in the absence of any changes in the number of processes and forums, we tend to observe that many algorithms show minimal deviation in the wait time over the workloads. For example, algorithms like Joung, Hadzilacos, Jayanti, Aravind's Fair and scalable algorithm where we observed an almost linear increase in CS entries with an increase in run time, demonstrate minimal deviation for the delay. Similarly, for algorithms that provided contrasting performance for the forum switches with respect to the CS entries they achieved, compensate for that with delay. For example, Aravind's queue algorithm that achieved more forum switches than Blelloch, Cheng, and Gibbons's algorithm and Aravind's Fair and Scalable algo-

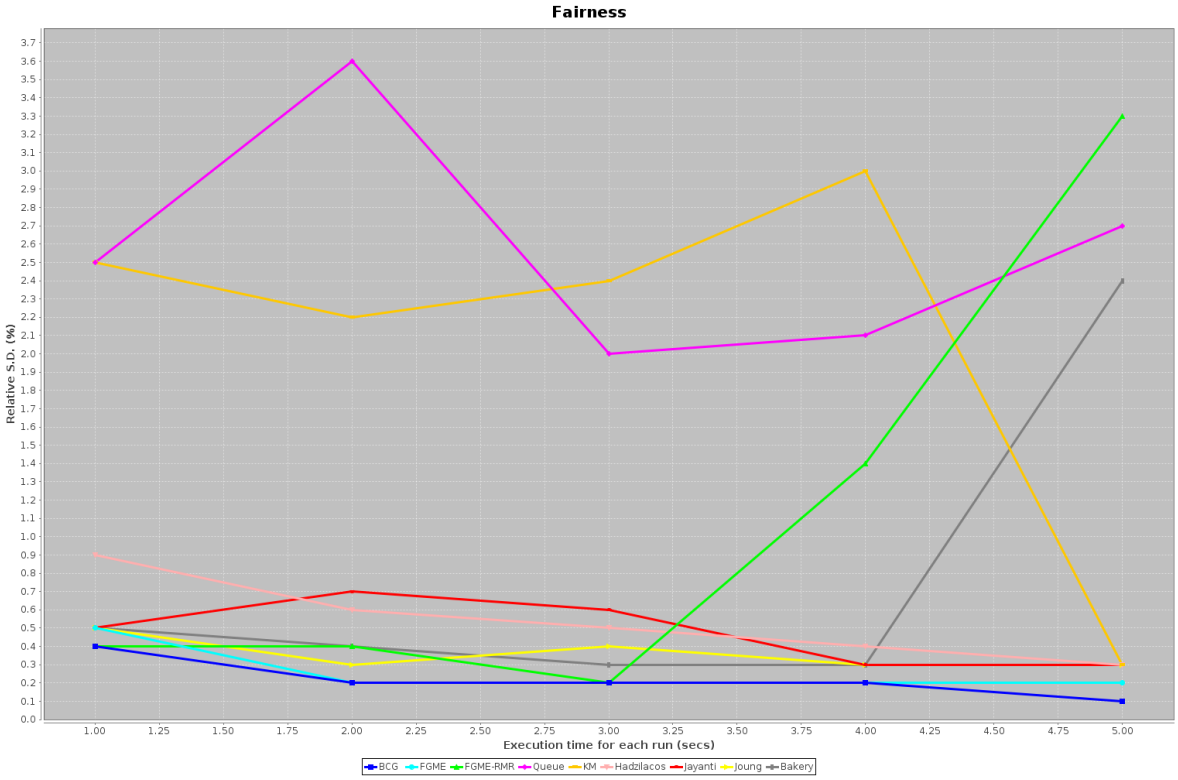

Figure 5.12: Fairness for varying time experiment

rithms, has substantially more delay than the other two which eventually justifies the higher number of CS entries by the two algorithms over Aravind's queue algorithm.

For the fairness results, it has been observed that FCFS based algorithms outperform most of the non-FCFS based algorithms. Also, the algorithms that demonstrated a linear increase in CS entries and minimal deviation in delay with each run of workload achieved better fairness results with a relative deviation of average CS entries under **1%** for most of the workloads.

5.3.4 Mutual Exclusion

Since all the existing algorithms for the group mutual exclusion problem are derived from already existing mutual exclusion algorithms, we were keen to test and observe the performances of these algorithms by using them as mutual exclusion algorithms. To achieve that, we vary both the number of processes and the number

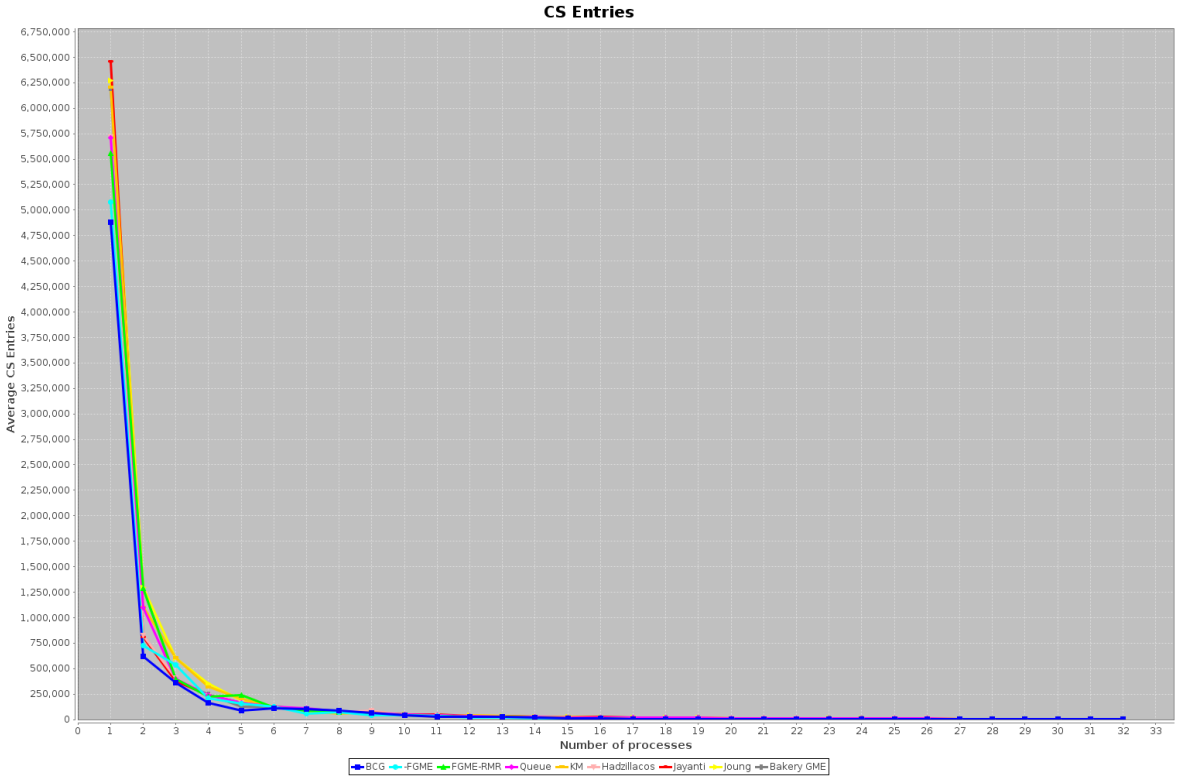

Figure 5.13: Critical Section entries for mutual exclusion experiment

of forums from 1 to 32 for each workload such that each process chooses its own id as the forum while requesting to access CS. Workloads are generated with increasing number of processes and forums. For each workload, the number of processes is equal to the number of forums, consequentially eliminating the use of forums for this experiment. We keep execution time constant at 2 seconds for each run.

As we have two variants for this experiment, we have a choice to make to choose either varying number of processes or varying number of forums as our domain axis for plotting graph. We decided to use the varying number of processes as our domain axis for each graph in this experiment. Processes experience maximum contention for accessing CS in this experiment as each process chooses a different forum to access the CS.

In the first result, we observe the number of CS entries attained by each of the algorithms for the given workloads. The trends were similar to the CS entries ob-

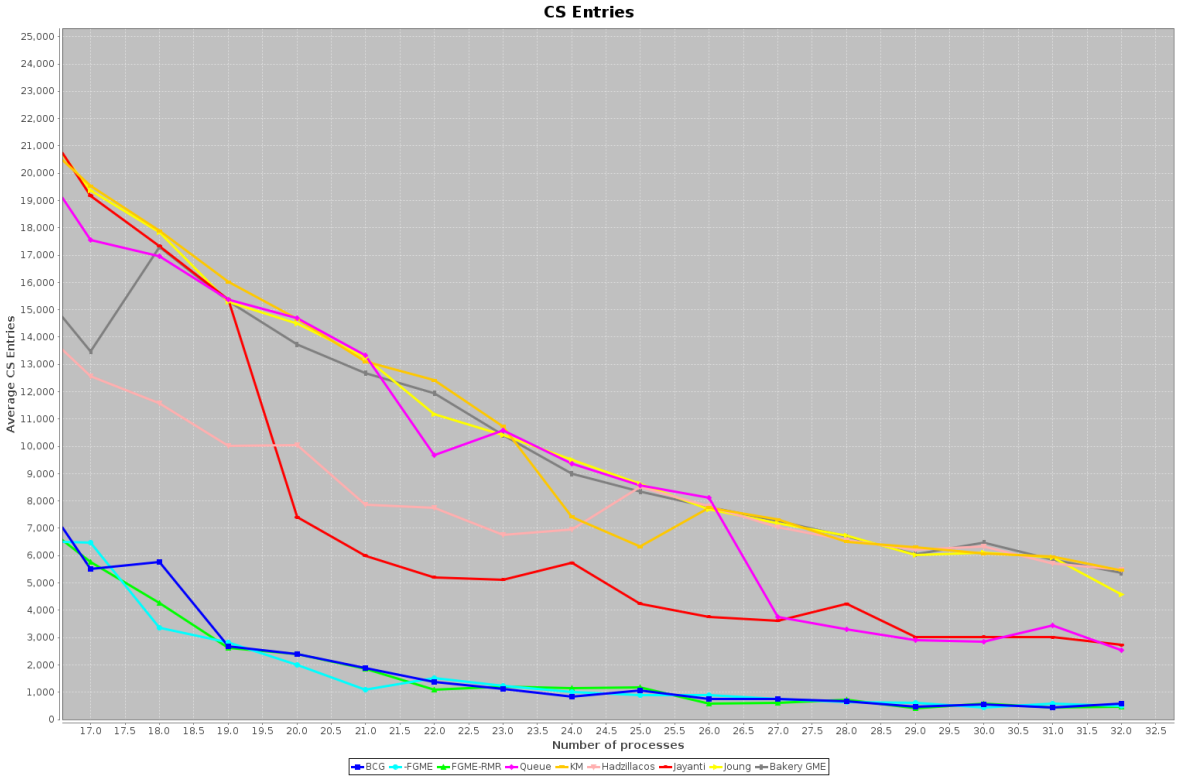

Figure 5.14: CS entries for mutual exclusion experiment (processes 17-32)

served for CS entries achieved in the experiment of varying processes. With each subsequent workload, we observed an exponentially decreasing number of CS entries for all algorithms. For this experiment, the non-FCFS based algorithms show better performance than most of the FCFS based algorithms for workloads with less number of processes. However, for the higher number of processes, some of the FCFS based algorithms like Hadzilacos, Jayanti and Aravind's Bakery group mutual exclusion algorithms, achieve a similar number of CS entries. Aravind's Bakery group mutual exclusion algorithm and Jayanti's algorithm achieves the highest number of CS entries for this experiment among FCFS based algorithms and algorithms using higher level read write operations. Aravind's Fair and Scalable algorithms, both with and without RMR and Blelloch, Cheng, and Gibbons's algorithm achieve a lesser number of CS entries for workloads with higher processes (more than 16).

For forum switches, the results observed are usually consistent with the results observed with CS entries. By understanding, the results for the CS entries and forum

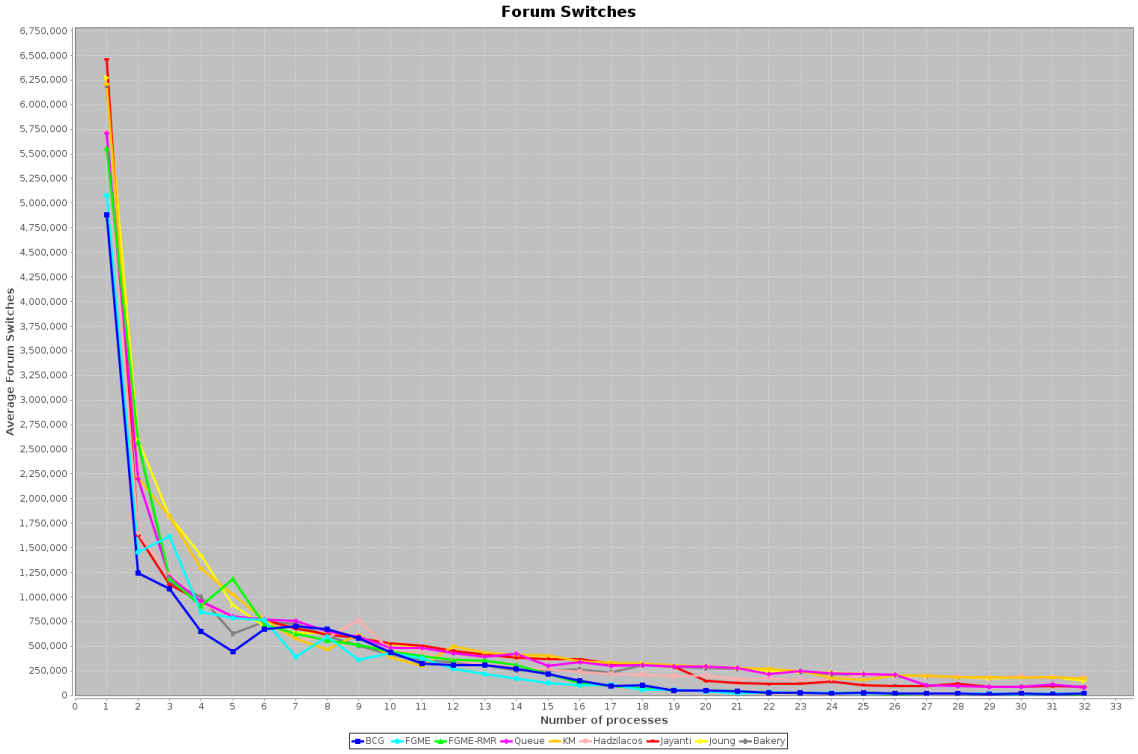

Figure 5.15: Forum switches for mutual exclusion experiment

switches should be same, in the absence of forums, when each process access CS in isolation.

An interesting trend observed was that, for the lesser number of processes (less than 16), the number of CS entries achieved by the algorithms is higher than the number of CS entries achieved by processes for the experiment of varying processes. We assume that this behavior observed could be arising due to the self-checking Critical Section where repeatedly, it's the same process checking for validating its access, unlike other scenarios where one or more processes might be involved performing the same task. However, if this isn't due to the self-checking critical section, it would be an interesting to compare performances of group mutual exclusion algorithms with mutual exclusion algorithms.

The results for delay compliment the results in the number of CS entries. Algorithms that achieve lesser CS entries (Aravind's Fair and Scalable algorithms with and

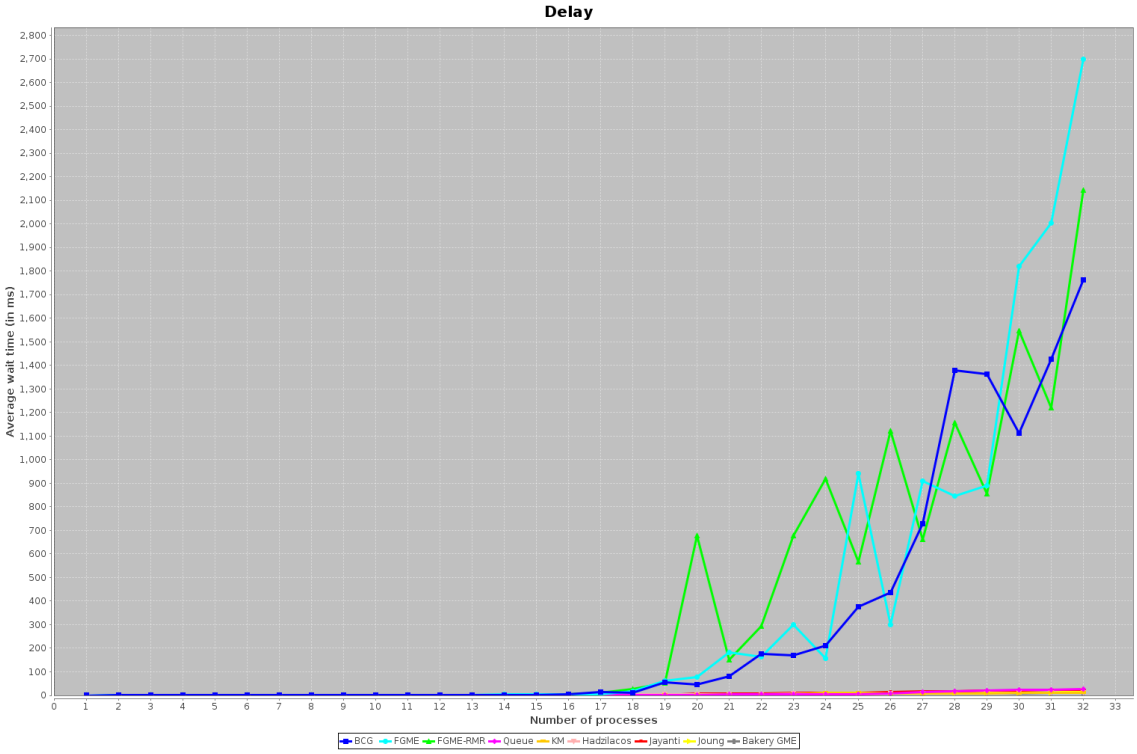

Figure 5.16: Delay for mutual exclusion experiment

without RMR and Blelloch, Cheng, and Gibbons's algorithm) tend to generate the higher delay. For this experiments, the inter-dependency of metrics is only between CS entries and delay, unlike other experiments that would involve forum switches as well.

For fairness, barring the algorithms with lesser CS entries and significant delay (Aravind's Fair and Scalable algorithms and Blelloch, Cheng, and Gibbons's algorithm), all the other algorithms achieved better results with a deviation less than **25%** across all the runs. The deviations for Aravind's Fair and Scalable algorithms and Blelloch, Cheng, and Gibbons's algorithm is on the higher side, especially with the higher number of processes, usually more than 25 processes, occasionally surpassing **300%**.

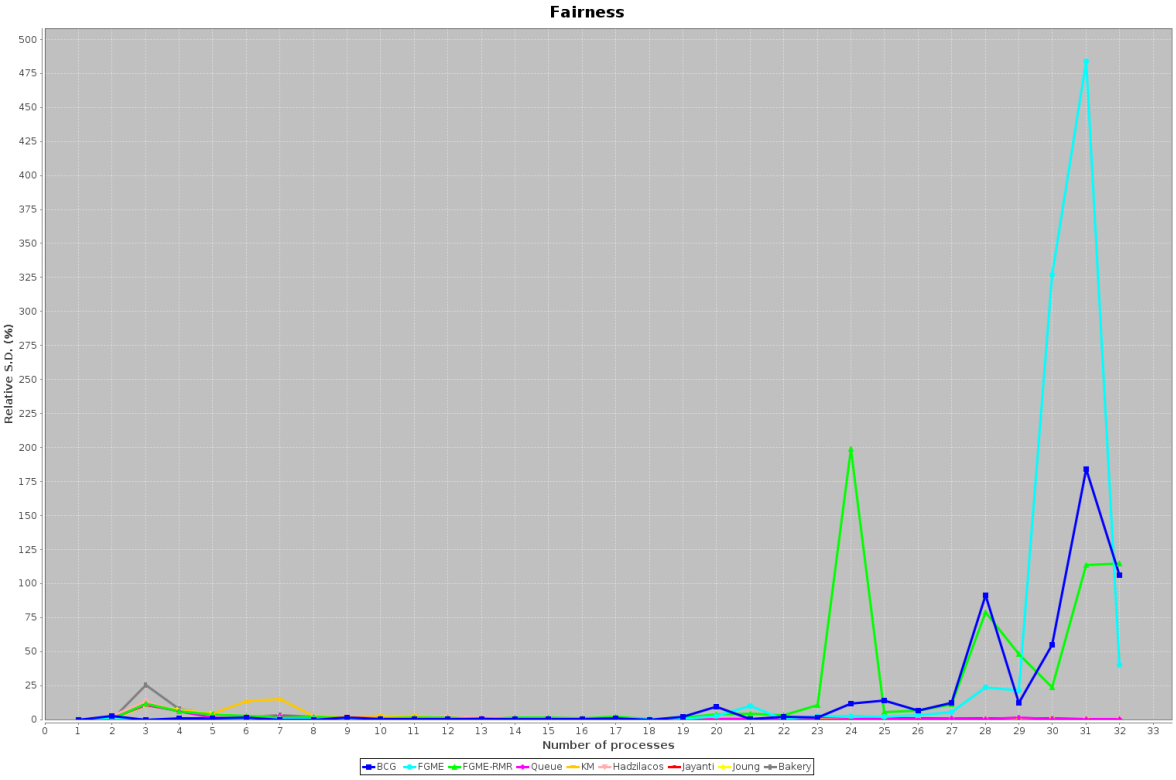

Figure 5.17: Fairness for varying time experiment

5.4 One-on-one Comparison Studies

We were able to successfully deduct our experiment set by drawing inspiration from the existing comparison studies for group mutual exclusion algorithms. Apart from executing algorithms individually, our framework could be used for either one-on-one comparison or overall comparison using all algorithms.Thus, we performed a few of those comparison experiments to determine whether or not we were able to fetch similar results under similar experimental setup. We performed two different one-on-one performance studies which we present in this section. We performed selected experiments from the pst performance studies conducted by Keane and Moir and Blelloch, Cheng, Gibbons. Only the experiments that were similar to our experiment set were selected for this study. These experimental set ups required us to make minimal changes to our existing experiments.We retained our performance metrics for these comparison studies as we believe our metrics collectively provide for an overall

comparison. However, we have excluded fairness from our results as neither of the algorithms is predominantly FCFS or fairness based algorithms.

5.4.1 Blelloch, Cheng, and Gibbons vs Keane and Moir

Our first comparison is between Keane and Moir's and Blelloch, Cheng, and Gibbon's algorithms. Blelloch, Cheng, and Gibbons conducted their study using two experiment sets, one with selecting forums with equal probability and the other with selecting forums with a biased probability.

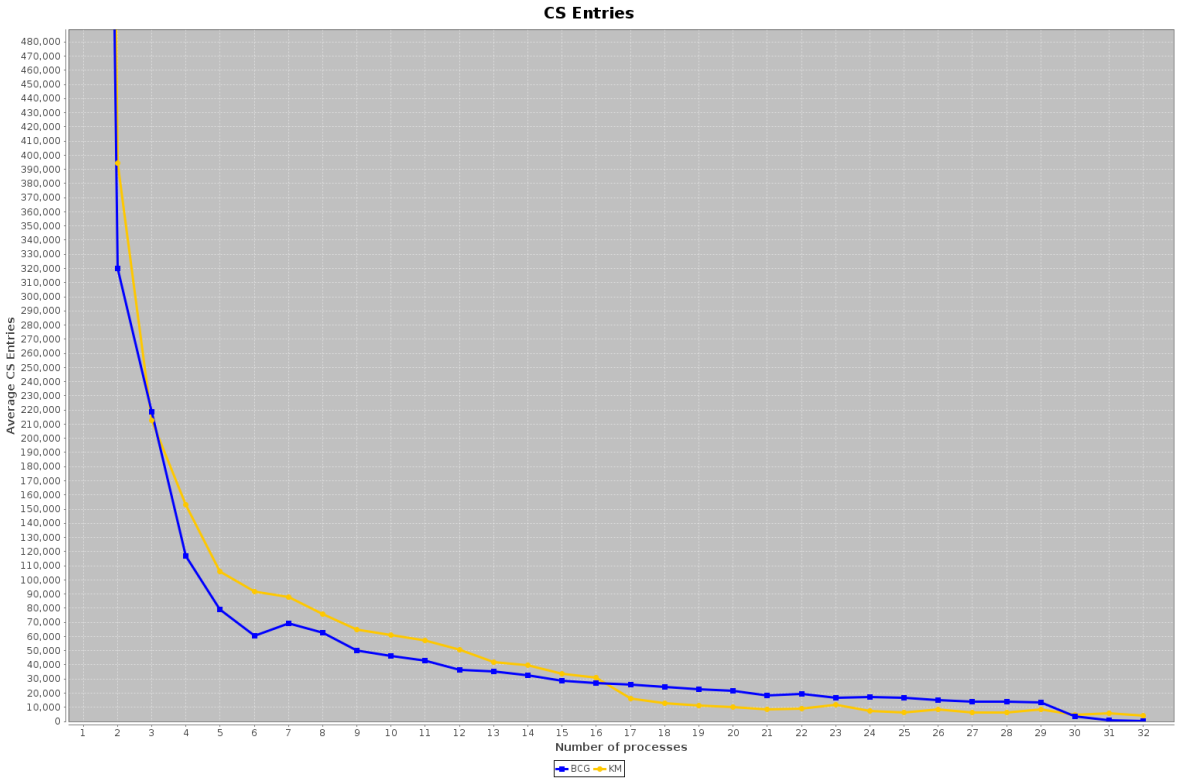

Figure 5.18: CS Entries - Keane-Moir vs Blelloch-Cheng-Gibbons

We conducted our comparison study with biased probability using our self-checking critical section, which would facilitate environment similar to their high load setting

as our processes too, do some work inside the self-checking CS. Since they performed their experiments varying the number of processes from 1 to 32 and keeping the number of forums constant at 2, we retained the same setup . We varied the duration of the execution to 2 seconds per run unlike execution based on attaining 1000 CS entries in their case. For the comparison study, we retained our performance metrics, CS entries, forum switches and delay, as we believe they provide an overall comparison unlike theirs.

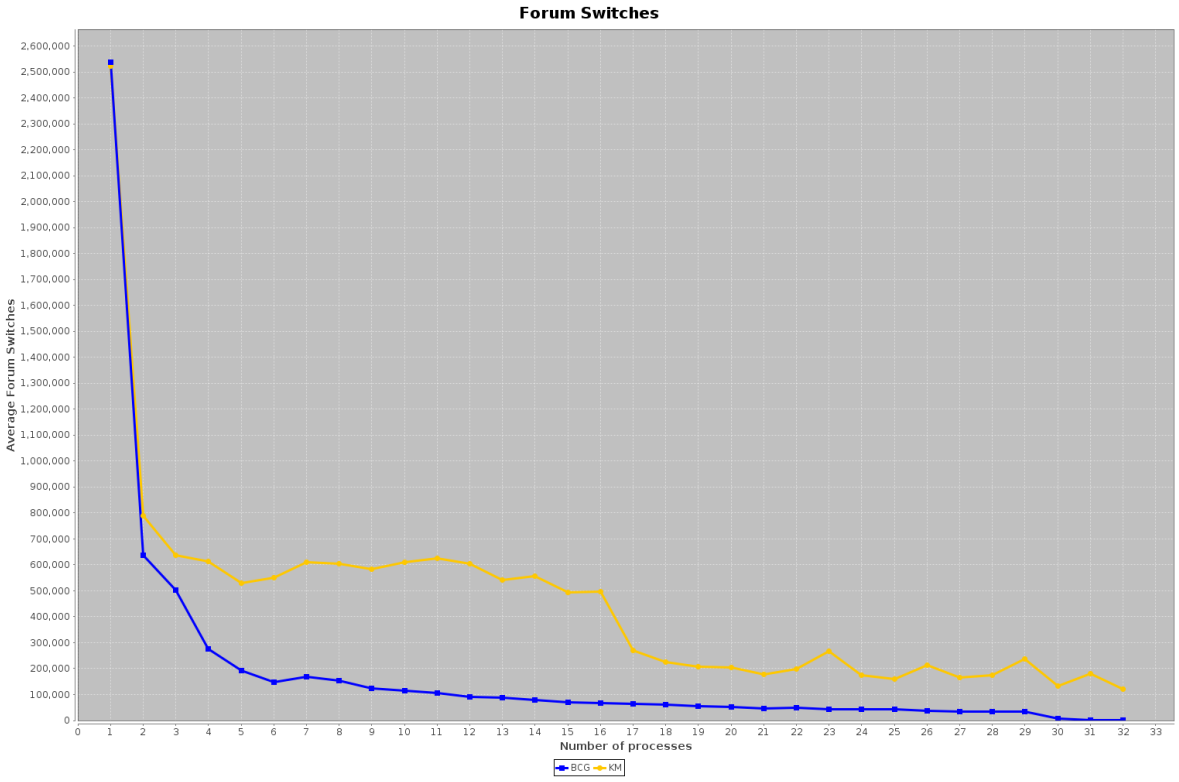

Figure 5.19: Forum Switches - Keane-Moir vs Blelloch-Cheng-Gibbons

The number of CS entries achieved by Keane and Moir's tends to be more than Blelloch Cheng, and Gibbons's for the lower number of processes (less than 16). For higher number of processes, they both perform similar for most runs, with Blelloch, Cheng, and Gibbons's achieving slightly more number of CS entries than Keane and Moir's during a few runs. This observation is akin to the observations made by them.

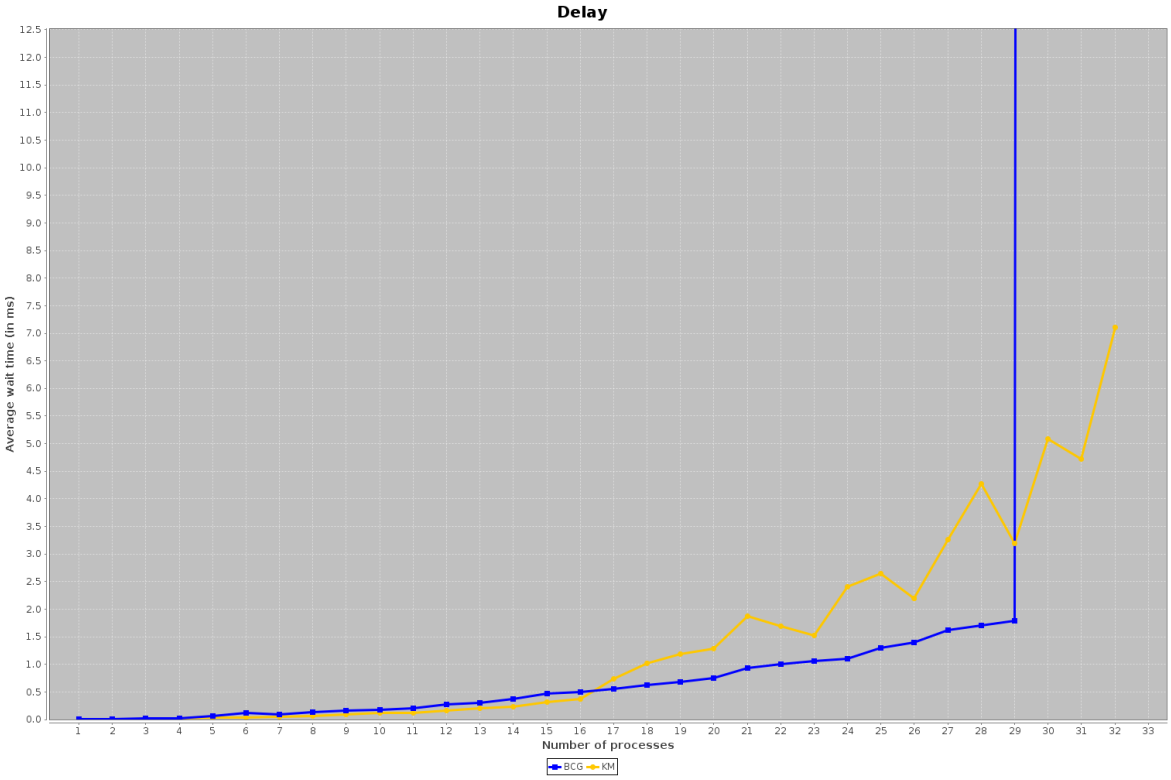

Figure 5.20: Delay - Keane-Moir vs Blelloch-Cheng-Gibbons

The number of forum switches attained by Keane and Moir's is more than that of Blelloch, Cheng, and Gibbons's. The number is much higher for workloads with the lower number of processes, and the observation was consistent for all our executions.

For delay, Keane and Moir's clock lesser waiting time than Blelloch, Cheng, and Gibbons's for workloads with a lower number of processes during all the executions. With an increase in the number of processes, we observed that Keane and Moir's records a slightly more wait time than Blelloch, Cheng, and Gibbons's.

5.4.2 Keane and Moir vs. Joung

Our next comparison is between Keane and Moir's [2] and Joung's [1] algorithms. Keane and Moir conducted their study using five experiments. Three of our experiments, varying processes, varying forums and mutual exclusion are either similar or relatable to their experiments. We conducted a comparison study between these two

algorithms using the above three experiments. We retained each of our experiment's setup for this comparison study. We varied the duration of the execution to 2 seconds per run unlike execution based on attaining 1000 CS entries.

5.4.2.1 Varying processes

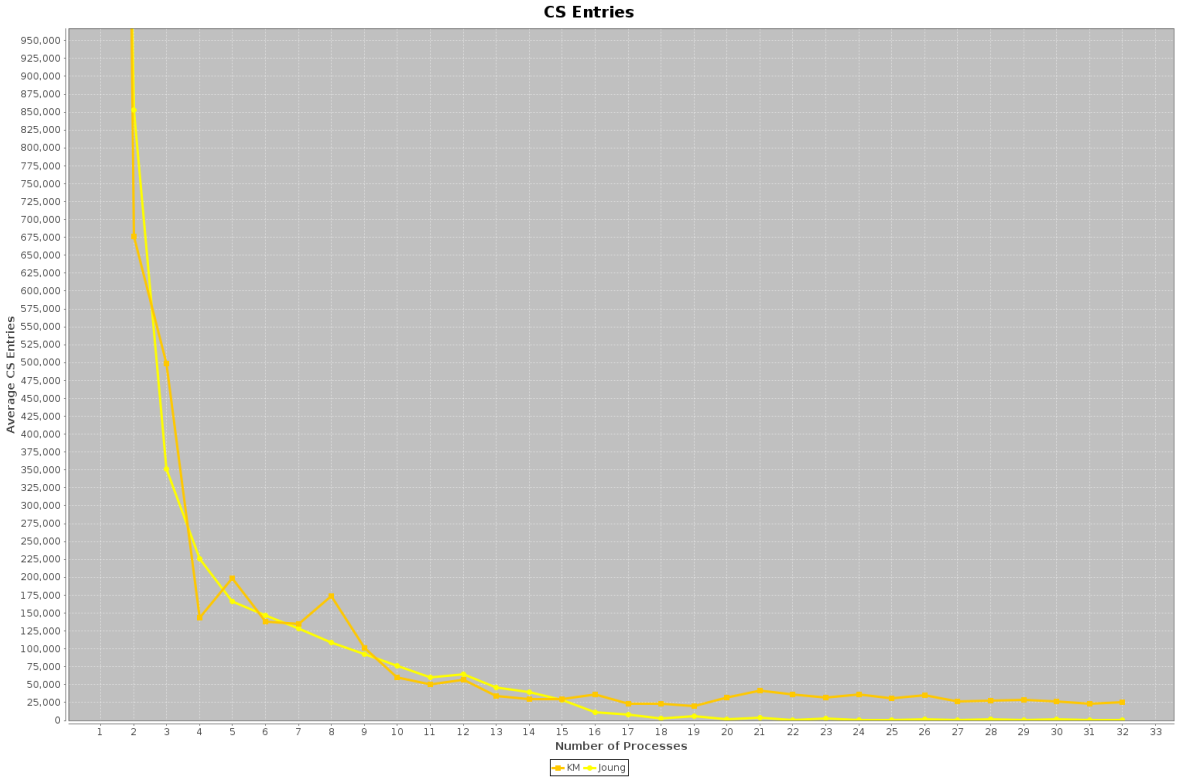

Figure 5.21: CS Entries - Keane-Moir vs. Joung

The number of CS entries achieved by Keane and Moir's and Joung's algorithms are similar for workloads with a lower number of processes (less than 16). For higher number of processes, Keane and Moir's algorithm tends to achieve slightly higher number of CS entries.

The result for forum switches for this experiment produces similar trends to that of the CS entries where both the algorithms perform similarly for the lower number of processes and Keane, and Moir's algorithm achieves a higher number of switches

for workloads with a high number of processes.

For delay, Keane and Moir's algorithm clocks lesser wait time than Joung's throughout the execution for all workloads. With an increase in number of processes, we observed that the difference in wait times for both the algorithms increases many folds with the former achieving significantly lesser wait time than the latter.

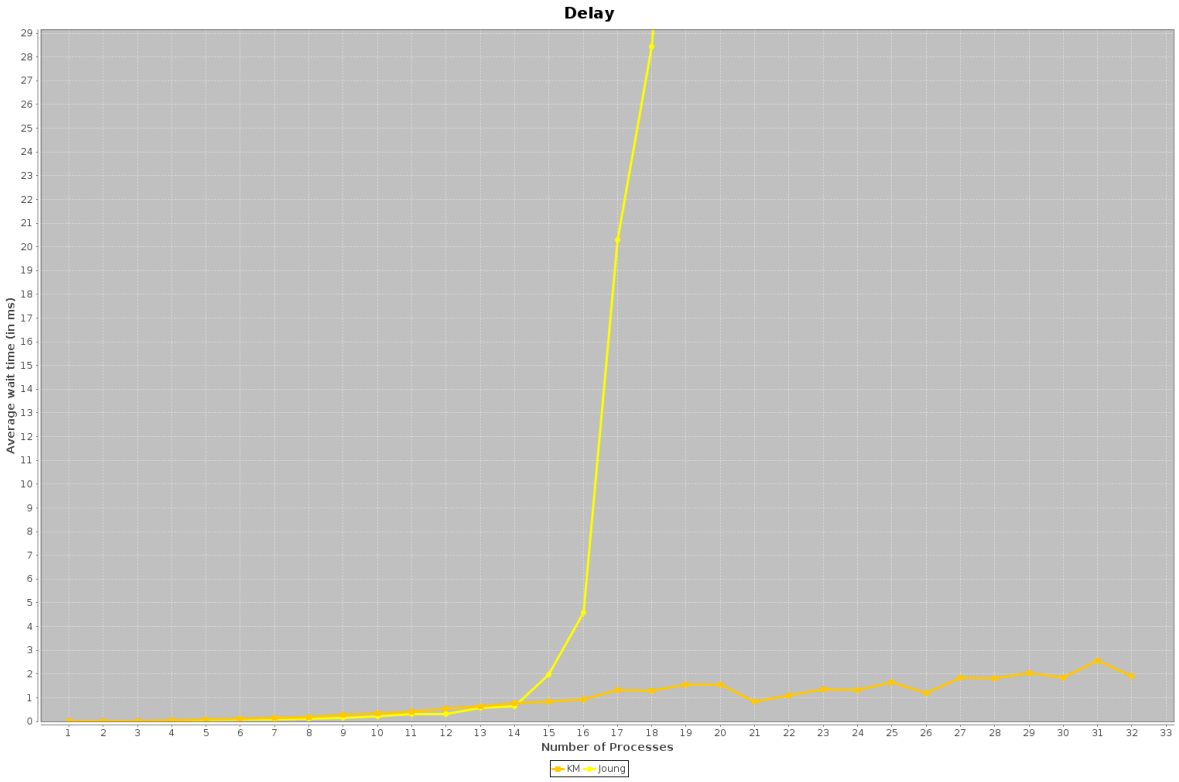

Figure 5.22: Delay - Keane-Moir vs Joung

5.4.2.2 Varying forums

The number of CS entries achieved by Keane and Moir's algorithm turns out to be significantly more than that of Joung's for all the workloads. The difference sometimes

tends to be as much as two-fold more in favor of Keane and Moir's algorithm.

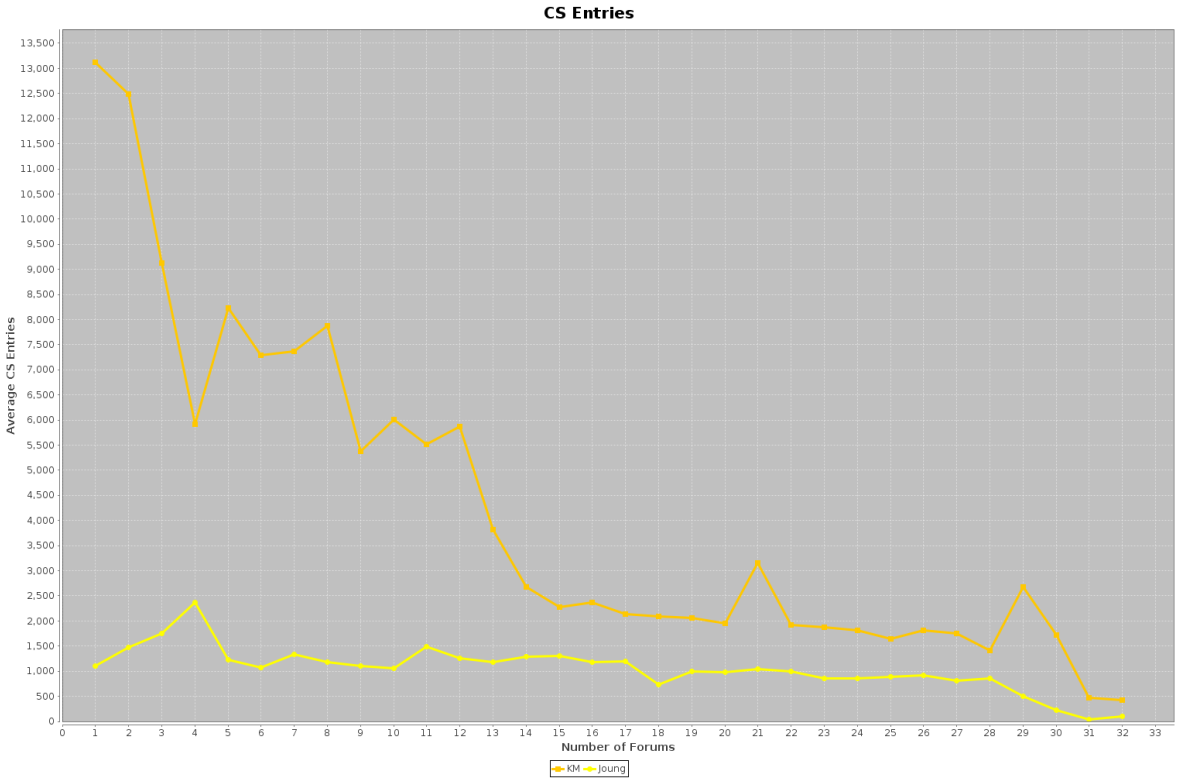

Figure 5.23: CS Entries - Keane-Moir vs Joung

Barring an odd spike encountered occasionally, the number of forum switches attained by Keane and Moir is more than that of Joung and thus, in consistency with the results of CS entries.

For delay, Keane and Moir's algorithm clocks lesser wait time than that of Joung's algorithm for all the workloads, which is also consistent with the CS entries and forum switches.

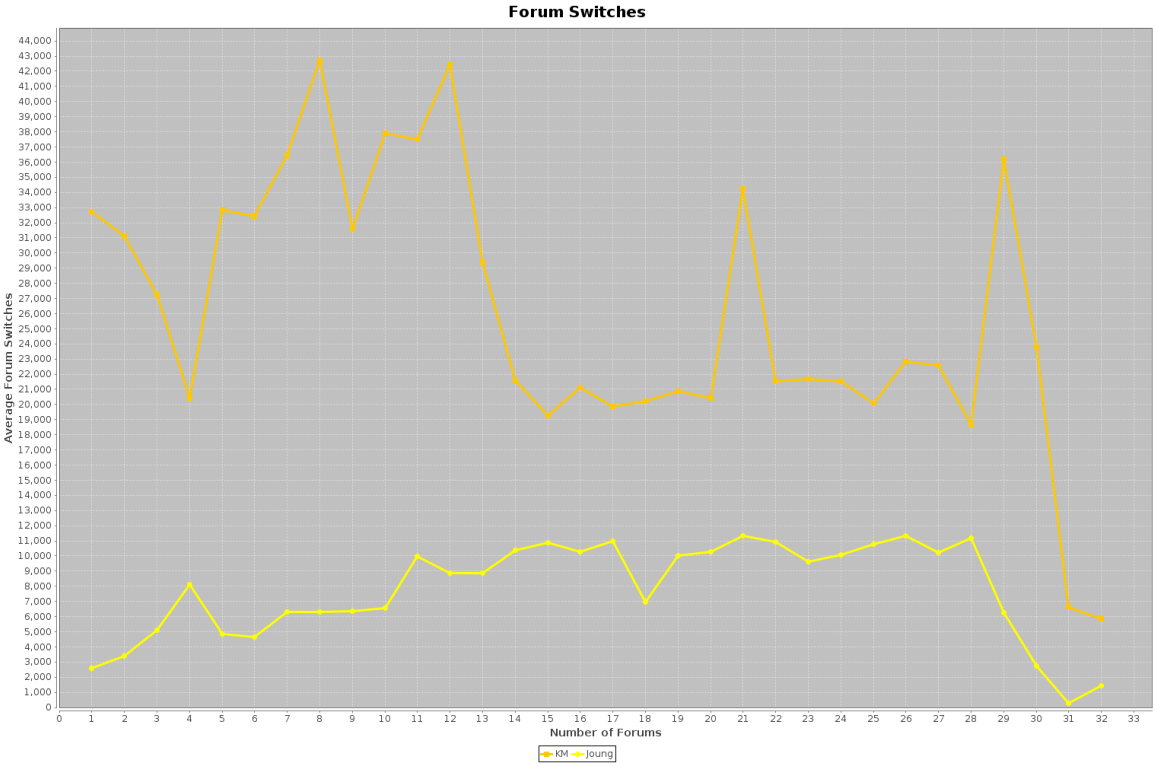

Figure 5.24: Forum Switches - Keane-Moir vs Joung

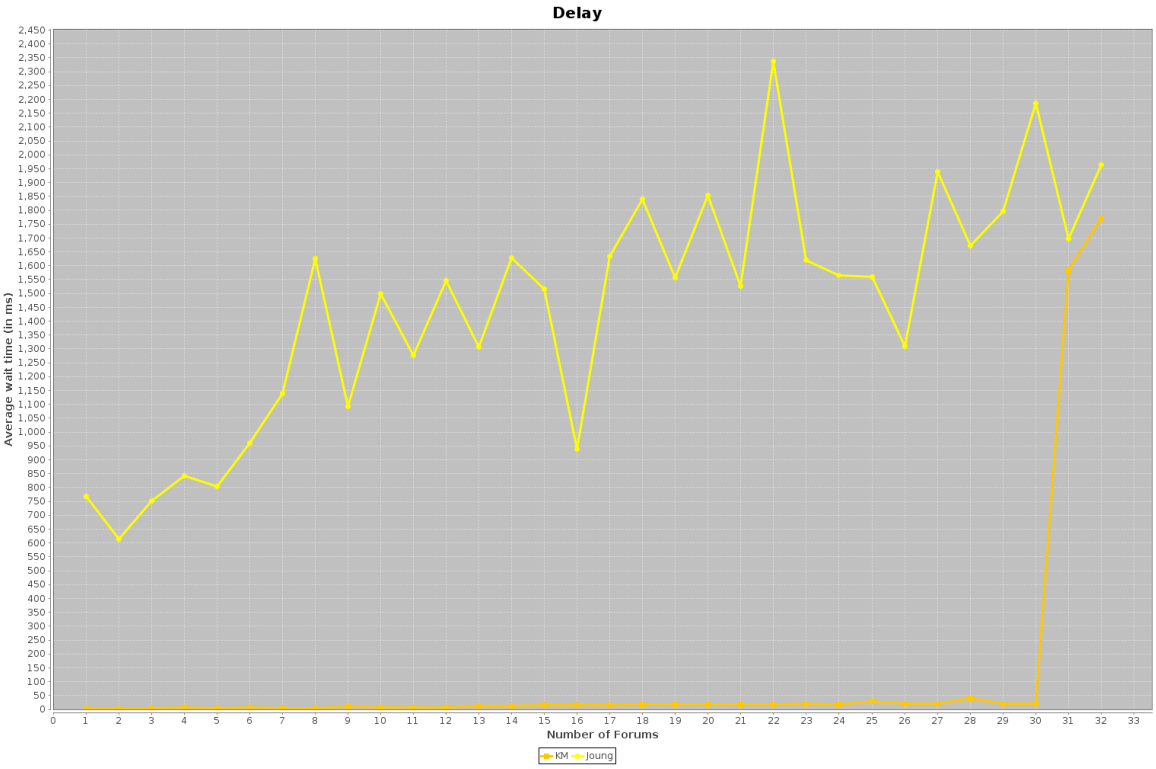

Figure 5.25: Delay - Keane-Moir vs Joung

5.4.2.3 Mutual exclusion

The number of CS entries achieved by Keane and Moir's algorithm is slightly more than Joung's algorithm for the lower number of processes (less than 16), while two achieve similar results for the higher number of processes. This observation partially contrasts the observations made my Keane and Moir as according to their experiment, their algorithm outperforms Joung's for all number of processes.

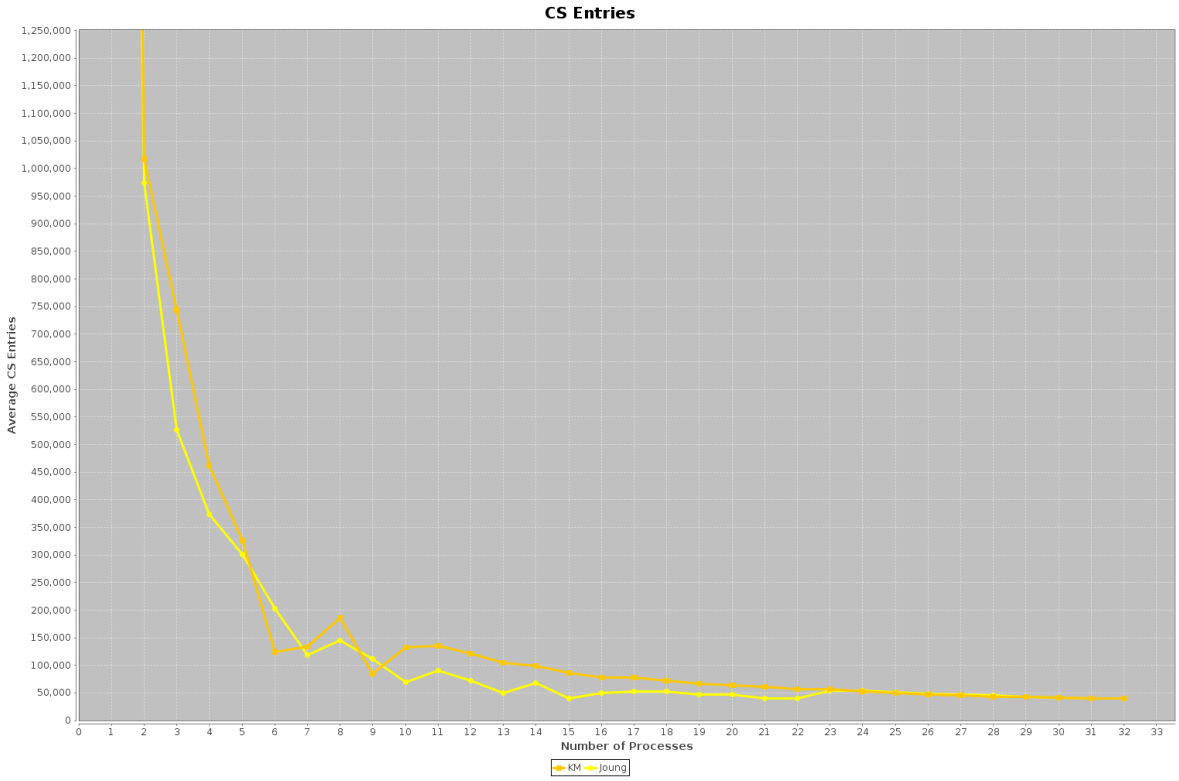

Figure 5.26: CS Entries - Keane-Moir vs Joung

Since this experiment is aimed at testing the algorithms when used in their mutual exclusion form, the graphs for CS entries and forum switches turn out to be the same. This is because the total number of CS entries is equal to the total forum switches. Hence, we have excluded presenting the results of forum switches for this experiment. For the results of delay, Keane and Moir's algorithm achieves less wait

time than Joung's algorithm for workloads with less number of processes. However, the difference is not significantly large. With an increase in number of processes, we observed that both the algorithms have similar wait time.

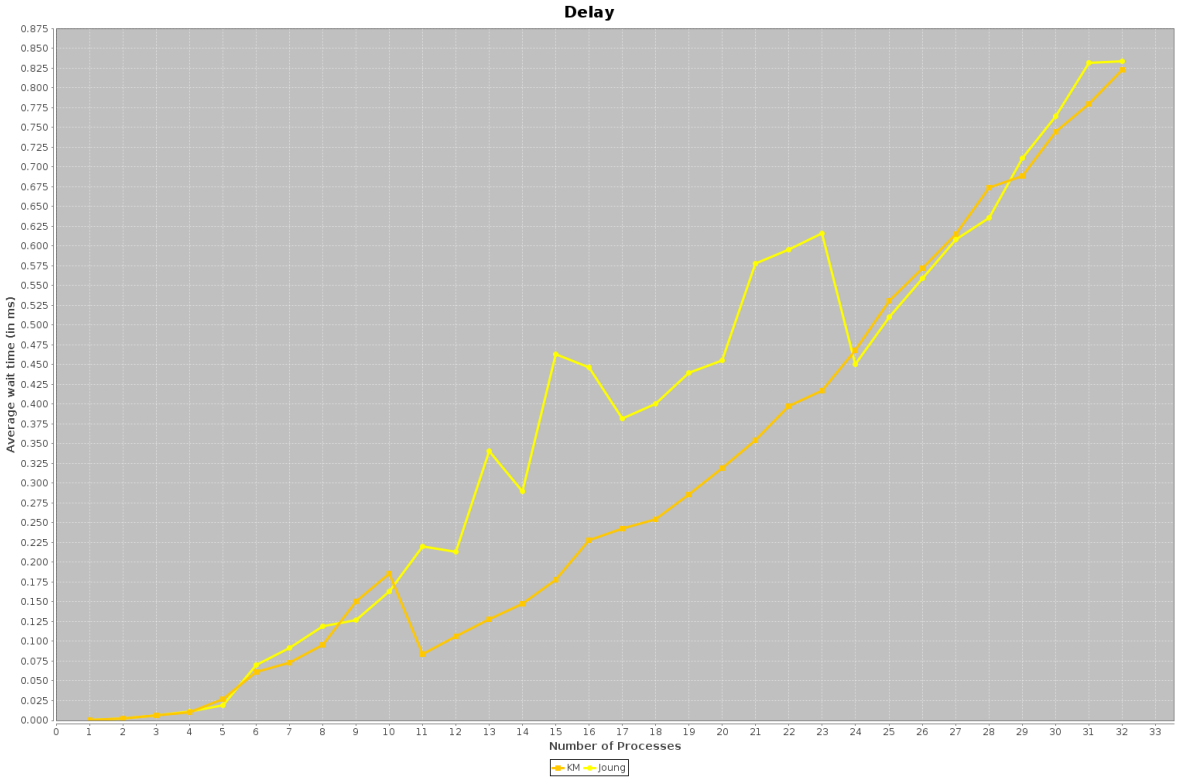

Figure 5.27: Delay - Keane-Moir vs Joung

5.5 Summary

In this chapter, we presented experimentation results of our performance study sets, both comprehensive and one-on-one. We made some important observations with respect to the results generated from our experiments.

For our overall comparison experiments, we observed that for many of our experiments, few algorithms with higher time complexities outperformed algorithms with much lesser time complexities. These observations were in sync with the observations

made by Buhr et al. [16], where few of the software solutions for mutual exclusion problem outperformed hardware solutions, which in theory would contract their behavior. With that, we deduct that the primary reason for such anomalies could be system latency and due to different higher and lower level instructions used by the algorithms. We also observed that most of the FCFS based algorithms tend to achieve consistent CS entries for each of their runs for a given workload in comparison to other non-FCFS based algorithms. For our one-on-one comparison experiments, we tried to replicate the existing experiment studies from literature in our framework, and our observations were mostly consistent with their observations.

In the following chapter, we conclude our thesis and present possible future work that could be intended to achieve.

Chapter 6

Conclusion and Future Directions

This research opportunity greatly helped us in sharpening our learning curve. It was instrumental in helping us understand the essence of academic research along with gaining extensive knowledge of the concerned area. In this chapter, we discuss the contributions of this thesis, and the observations made with respect to our experimentations along with our experiences that we encountered during our work.

6.1 Contributions

For this thesis, we successfully developed an automated test framework that could be used for conduct performance study experiments for group mutual exclusion algorithms and thus observe their comparisons. To demonstrate the working of our framework, we implemented nine algorithms into code bases. Using these algorithms, we successfully extensive comparison studies along with one on one comparison of a few of the algorithms.

We conducted four comparison studies using all the eight algorithms we implemented to observe the performances of all the algorithms and two case studies for one on one comparison of few algorithms. We also performed a comparison study between Aravind's Fair and Scalable group mutual exclusion algorithms (both with

and without RMR) and Blelloch Chen Gibbons's room synchronization algorithm. Though the results of this study aren't provided in this thesis, it is available in the experimentation results section of [14].

6.2 Observations

We made some key observations with the results generated from our performance experiments. Most of our results for one on one comparison experiments were in sync with the observations made by similar studies in the literature. These results helped us confirm the accuracy of our framework and correctness of the algorithms implemented. For few of the extensive comparison experiments, under the given experimental setup, many algorithms with simple read write operations performed better than the algorithms using higher level read write operations.

This observation was an interesting one as those algorithms have higher time complexities that the ones they outperformed. Buhr et al. had a similar observation [15], where they observed some of the software solutions for mutual exclusion problem outperformed the hardware solutions. Since their system constitutes the basis of our framework, such observations were not surprising. We believe, this is due to the nature of instructions and such variation is observed due to the higher execution time needed for the higher level instructions to execute along with system latency playing its part.

6.3 Experiences

Our experiences throughout this research were enthralling, to say the least. We encountered several challenges and also realized how trivial few of the things turned out to be which was initially assumed significant. We discuss our experiences with respect each of the accomplishments achieved through this research.

6.3.1 Automated test framework

Since there exists no publicly available system or testing environment for conducting performance studies for group mutual exclusion, it was a challenge for us to set thresholds for our work and to gain an understanding of insights of the system. Hence we were dependent on drawing inspirations from similar systems designed for mutual exclusion. The execution engine of our framework is written in C language. It was important to do so as we are working with executions on actual machine and not in a simulation environment. Hence, using a lower level language was essential. At the same time, for the ease of usage and plethora of support, designing our GUI components in Java, including the plotting of our results in the form of graphs, was intuitive.

In today's world, cross platform development and modular nature of the system is the norm, and we wanted to exploit the same with our work. Our primary challenge was to establish communication between our components, in particular between the execution engine written in C and the other components for GUI and calculations written in Java. Java Native Interface (JNI) was one of the approaches, but a tedious one. So, we used Java processes to establish this communication. We wrote shell scripts to automate our process.

6.3.2 Algorithms to source codes

Implementing the pseudo codes of algorithms into code bases was intriguing. Having an in-depth understanding of each of the algorithms was a pre requisite for this work. But the most important aspect was to retain the original essence of the algorithms. Some of the algorithms heavily rely on the underlying mutual exclusion algorithms from which they are derived. Understanding those algorithms was essential too. We were fortunate for the fact that the C99 standard GNU that we used for our work has built in read-write operations, including many of the higher level operations. Many of the algorithms that we implemented for our work use these

instructions.

6.3.3 Performance experiments

For this contribution, the knowledge of performance studies from literature came in handy. Many of our experiments are inspired by the previous comparison studies accomplished for mutual exclusion and group mutual exclusion algorithms. In fact, all our One-one-one comparison studies involved repeating the studies for group mutual exclusion conducted in the past. Hence, the process of finalizing experiment sets was not a difficult one. However understanding results, observing patterns across executions and deducing inferences from them was both time-consuming and challenging process. Each of our experiments has undergone at least 100 executions, and we spent over 1000 hours of execution time for our experiments.

6.4 Future Directions

In any research, testing an outcome is as important as creating it. Through our work and our accomplishments, we have initiated a new dimension of testing group mutual exclusions by execution and conducting comparison studies among them. Having an environment is an integral part of the process. With our test framework, we have provided a testing environment to compare and contrast the performances of these algorithms collectively. We believe a substantial research could further be accomplished in this direction.

More algorithms could be implemented and accommodated into our test framework. The experiment sets could be expanded, and new experiments could be introduced. New performance metrics could be presented for these experiments. The existing framework could be deployed and used on other different machines. All these activities will help researchers gain a deeper understanding of the group mutual exclusion problem and meaningful insights to introduce newer solutions for the problem.

Since ours is the first test framework for the group mutual exclusion problem, we might have missed out on accommodating few key features into the framework. Tasks could be taken up to upgrade our environment, add new features to it and rectify any existing limitation if encountered. Also, newer and better test frameworks could be proposed and developed using different programming languages with time.

Bibliography

[1] Y. Joung. Asynchronous group mutual exclusion. *Distributed Computing Springer-Verlag.*, 13: 189-206, 1999.

[2] P. Keane, and M. Moir. A Simple Local-Spin Group Mutual Exclusion Algorithm. *IEEE Trasactions on Parallel and Distributed Computing.*, 212(7):673-685, 2001.

[3] E. Dijkstra. Solution of a problem in concurrent programming control. *Commun ACM.*, 8(9) 569, 1965.

[4] V. Hadzilacos. A note on group mutual exclusion. In, *Proceedings of the 20th annual Symposium on Principles of Distributed Computing* , pages 100-106, 2001.

[5] A. Aravind, and K. Vidyasankar. Elegant solutions for group mutual exclusion problem. *unpublished manuscript*, 1999.

[6] P. Jayanti, S. Petrovik, and K. Tan. Babyos: Fair Group Mutual Exclusion. In, *Proceedings of the twenty-second annual symposium on principles of distributed computing, pages 295-304, New York, NY, USA* , pages 100-106, 2003.

[7] GNU Compiler Collection. Built-in functions for atomic memory access. *https://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/Atomic-Builtins.html.*

[8] M. Takumura, and Yoshihide Igarashi . Group Mutual Exclusion Algorithm Based on Ticket Orders. In, *Proceedings of the 9th Annual International Computing and Combinatorics Conference (COCOON'03)*, LNCS, 2697, 232-241, 2003.

[9] A. Aravind. Fair Group Mutual Exclusion, *Pending approval* . In, *Proceedings of ACM Transactions of Parallel Computing*, 2017.

[10] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Scalable Room Synchronization. In, *Proceedings of Theory of Computing Systems*,36, 5 (2003), 397-430.

[11] J. Mellor-Crummey, and M. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In, *Proceedings of ACM Transactions on Computer Systems*, 9(1):21-65, 1991.

[12] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. In, *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, 106-113. ACM, 1991.

[13] V. Manickam. A flexible simulation framework for processor scheduling algorithms in multicore systems. *Master's Thesis*, University of Northern British Columbia, 2012.

[14] A. Aravind, and W. H. Hesselink. Group Mutual exclusion by fetch-and-increment. In, *Proceedings of ACM Transactions on Parallel Computing*, 2016.

[15] P. Buhr, D. Dice and W. H. Hesselink. High-performance N-thread software solutions for mutual exclusion. In, *Concurrency And Computation: Practice And Experience*, Published online in Wiley Online Library, DOI: 10.1002/cpe.3263, 2014.

[16] P. Buhr, D. Dice, and W. H. Hesselink. Concurrent Locking. *Available at:* , https://github.com/pabuhr/concurrent-locking, 2014.

[17] A. Aravind. Queue Group Mutual Exclusion. *unpublished manuscript*, 2011.

[18] R. Danek, and V. Hadzilacos. Local-Spin Group Mutual Exclusion Algorithms. In, *Distributed Computing*, R. Guerraoui (Ed.), Springer-Verlag Berlin Heidelberg, 2004.

[19] V. Bhat, and C. Huang. Group Mutual Exclusion in O(log n) RMR. In, *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing(PODC'10)*, ACM, New York, USA 2010.

[20] J.H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. In *Distributed Computing*, 9:51-60, 1995.

[21] J. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. In, *Distributed Computing*, 11:1-20, 1997

[22] A.T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In, *Proceedings of the 1996 International Conference on Computer Design*, 1996