Adaptive Concurrency Control Based On Workload Characteristics

D. J. N. Costa
Index No : 14000131

Supervisor: Dr. D. A. S. Atukorale
Co-Supervisor: Dr. M. Jayasinghe

January 2019

Submitted in partial fulfillment of the requirements of the
B.Sc in Computer Science Final Year Project (SCS4124)

Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: D. J. N. Costa

……

Signature of Candidate Date:

This is to certify that this dissertation is based on the work of Mr. Dehiwalage Jude Nilushan Costaunder my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor's Name: Dr. Ajantha Atukorale

……

Signature of Supervisor Date:

This is to certify that this dissertation is based on the work of Mr. Dehiwalage Jude Nilushan Costaunder my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor's Name: Dr. Malith Jayasinghe

……

Signature of Supervisor Date:

Abstract

Unlike servers that were used in the early days of computing, modern application servers utilize multi-core processors with multi-threading to make it possible to perform computations in an efficient manner. Modern servers are required to handle a large number of requests concurrently and therefore, the applications that are running on them need to be written in a manner to best utilize the computing resources available. Performance of such servers depends on a number of tunable configuration parameters such as session timeouts, keep alive timeouts and thread pool sizes. The values of these parameters are set off-line and do not change during run time. However, the best parameter values will depend on the workload conditions and therefore, setting these parameter values to fixed values can cause significant performance degradation.

In this dissertation, we specifically focus on tuning the size of a thread pool on application servers. Experimentally we show that the size of the thread pool which is used to process the requests has a significant impact on the performance. We then propose three adaptive algorithms that can auto-adjust the the thread pool size to optimize performance. The proposed algorithms are capable of optimizing given performance metric (throughput, average latency or 99% latency) online. It does so by periodically measuring the performance and then adjusting the thread pool size in order to optimize performance.

This proposed methodology helps to increase the performance of software systems by better utilizing the available computing resources.

Preface

The implementation presented here in this dissertation utilized several open source libraries developed by parties external to this research.

An HTTP server simulating several different types of applications were implemented by the author for testing purposes of this research.

The algorithms presented in this dissertation are the work the author.

The evaluations conducted throughout the study were performed on a test environment set up at the University of Colombo School of Computing using commercially available hardware. All evaluations were conducted by the author.

Acknowledgement

I wish to thank my Supervisor Dr. D. A. S. Atukorale and my Co-Supervisor Dr. M. Jayasinghe for guiding and motivating me throughout this research project. This project would not have been possible if not for all the guidance and generous help extended to me by them.

I would also like to thank Mr. Isuru Perera for the invaluable assistance given to me during the project.

I would like to thank Dr. H. E. M. H. B. Ekanayake, the coordinator of the undergraduate Computer Science research course of the University of Colombo School of Computing for the guidance given to us during this project.

The test environment used for this research was set up at the Network Operations Center of the University of Colombo School of Computing. Therefore I wish to thank its staff for providing me with the necessary hardware and for setting it up.

Last but not least, I would like to thank my friends and family for their support and motivation given to me to conduct this research.

Table of Contents

List of Figures

List of Tables

List of Acronyms

CPU	- Central Processing Unit
DBMS	- Database Management Systems
HTTP	- Hypertext Transfer Protocol
IaaS	- Infrastructure as a Service
I/O	- Input/Output
JDK	- Java Development Kit
JVM	- Java Virtual Machine
KB/s	- Kilo Bytes per second
STM	- Software Transactional Memory

Chapter 1 - Introduction

1.1 Background to the Research

When Computers were invented, they were used for simple tasks. But with the rapid development of hardware and software running on Computers, the possibilities of computers increased. At present, computers are used to perform various complex tasks.

With the increase in the use of web application servers to serve clients using web technologies, servers must be capable of handling large number of requests concurrently. A commonly used technique to handle such concurrent requests is multi-threading. Pools of threads are created within the server and requests are passed to threads in these pools to be processed. This pool of threads within the server is considered as the request processing thread pool and this thread pool shall be referred as the worker thread pool in the rest of this dissertation.

Different requests require the servers to perform different types of operations. Some of the operations that the server may perform include CPU dominant tasks where the use of the CPU to perform computations is dominant or Input/Output dominant tasks where I/O operations are dominant when processing the request. Multi-threading is one of the techniques used to handle large number of requests. The number of threads to use in order to get the best performance depend on both characteristics of the application (i.e. I/O dominant, CPU dominant) and the incoming workload (arrival rate into server, number of concurrent users).

In this thesis, an effort was made to dynamically adjust the thread pool size depending on the application characteristics and incoming workload properties.

Two commonly used performance metrics were used to evaluate the performance: throughput and latency. Throughput is defined as the rate of processing requests and is usually expressed as the number of requests processed per second. Latency is the time

taken for the reply to be received from the server since the request is sent from the client. Therefore, it includes the processing time and queuing time as well. Throughput is said to improve when the value of throughput increases while latency is said to improve when the value of latency decreases.

When multiple threads of execution try to access the same resource simultaneously (which can only be accessed by a single thread at a time), thread contentions are said to occur. For example, thread contention may occur when multiple threads of execution try to write to a database simultaneously.

1.2 Research Problem and Research Questions

As already pointed out, the primary objective of this research is to bring self-configuration into servers. In particular, the focus of this project was on dynamically changing the thread pool size based on changing workload and application characteristics.

The size of the worker thread pool of a web server can be configured in the following manner.

1. Hard-coding the size by the web server programmer
2. Manually by system administrators

Although it is possible for the server programmer to hard-code the size of the worker thread pool into the server during development, this is not usually done. This is due to

the fact that the developed server would run in different hardware environments and the size of the thread pool to be used would depend on it.

Another approach to setting the value of the thread pool size is to allow system administrators to configure it by keeping this value as a configuration parameter in the developed web server. Figure 1.1 below shows a screenshot from the performance tuning guide of WSO2 Stream Processor [1, p. 2] explaining how to configure the worker thread pool size.

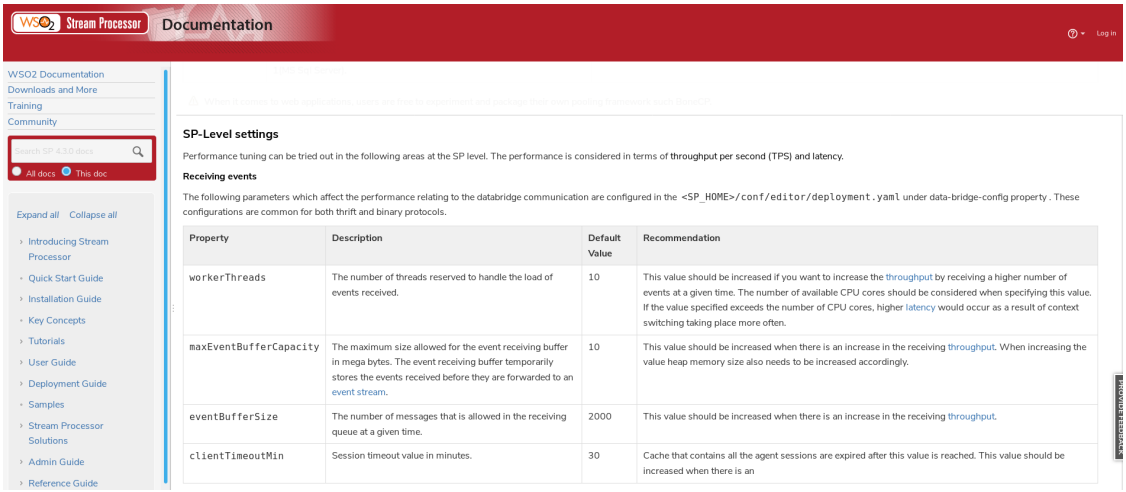

Figure 1.1: Screenshot of WSO2 SP performance tuning guide

System administrators can then find a thread pool size to be used based on given recommendations and through trial-and-error. However this value may need to be changed over time. Reasons for this include server hardware changes, changes of traffic patterns, changes made to processing of requests etc. In addition to that, modern trends in the use of cloud computing technologies such as Infrastructure as a Service (IaaS) has seen system administrators moving systems to remote servers. Unlike physical hardware on premise owned by an organization, cloud computing makes it much easier to upgrade or downgrade instances based on the needs of an organization. Therefore the type of computing resources on which systems are run may change much frequently. Considering these issues, the experiments need to be re-run to find a new thread pool size. Although possible, this is a cumbersome process.

This research project was aimed at finding a solution to the problem of finding the worker thread pool size to be used in a system to optimize throughput or latency. To answer this problem, the following research questions were identified.

1. How does the thread pool size impact the performance of different applications that receive requests under different arrival patterns?
   The initial step of this study was to understand the effect of thread pool size on throughput and latency. This was done by conducting experiments with different applications running on the server for varying worker thread pool sizes and incoming workloads.

2. How can an algorithm/s be developed that can dynamically change the thread pool size depending on the changing behavior of application and incoming workload?
   A significant step of this research project was to find an answer to this question by developing algorithms to dynamically change the thread pool size based on the changing behavior of applications and incoming workloads.

3. How can the parameters of the adaptive thread pool adjustment algorithms be tuned in order to improve results?
   The final step of this study was to search for an answer to this research question by experimentally tuning the parameters of the developed algorithms in order to improve their performance.

1.3 Justification for the research

This research study has both practical and theoretical benefits to the field of computing.

By varying the applications running on the server, worker thread pool sizes and number of concurrent requests, the effect of the size of thread pools on throughput and latency was studied. As thread pools are used in many software systems, this

information would help to get a better understanding on how to use them in an efficient manner.

Furthermore, the adaptive concurrency control mechanism developed through this research study has important practical benefits. As explained earlier, finding the thread pool size to optimize performance is a cumbersome task if done manually. However, by using the proposed method, systems can be made to change them automatically based on the traffic patterns and application characteristics. This means, if the system administrators use the proposed model, they do not need to experiment and reconfigure the systems when the server hardware in use changes, when traffic patterns change etc.

1.4 Methodology

In this section, the methodology followed during the course of this study is explained.

Initially, an HTTP server was implemented in Java using the Netty framework [2]. This HTTP server was implemented to simulate four I/O dominant and CPU dominant applications.

An environment consisting of two networked nodes dedicated to this study was set up where one node was designed as the server node on which the HTTP server was executed. The other node was designated as the client node and concurrent requests were generated and sent to the server using Apache JMeter [3] running on the client node.

An initial set of experiments were conducted by varying the worker thread pool size and the number of concurrent users for different applications, and the effects on throughput, mean latency and 99_{th} percentile of latency were recorded and studied.

This knowledge was used to develop three algorithms to adaptively adjust the thread pool size. The three algorithms were created to optimize performance of three different metrics namely, throughput, mean latency and 99_{th} percentile of latency.

After the development of these three algorithms, they were tested against fixed workloads. That is, in these experiments, a fixed number of concurrent users were simulated by Apache JMeter [3] running on the client node. The algorithms were

evaluated on how well they were able to adjust the worker thread pool size in order to improve performance.

Finally, the algorithms were tested under a varying workload. That is, in these experiments, the number of concurrent requests generated by Apache JMeter [3] running on the client node was made to vary with time. The algorithms were evaluated on how well they were able to respond to varying workloads and adjust the worker thread pool size in order to improve the relevant performance metric.

Experiments were also conducted to understand and tune the parameters of the algorithms.

1.5 Outline of the Dissertation

Chapter 1 – Introduction

This chapter presents information regarding the background to the problem and the research questions which this study aims to find answers to

Chapter 2 – Literature review

This chapter presents background information related to this study and a review of the existing literature

Chapter 3 – Design

This chapter presents the design of the research study and includes information about the applications tested, workloads simulated and the tools and technologies used.

Chapter 4 – Implementation

Details about the implementation of the proposed solution is presented in this chapter. This chapter presents the three algorithms that are proposed through this study.

Chapter 5 – Results and evaluation

This chapter presents the results that were obtained through the experiments conducted and explanations of those results.

Chapter 6 – Conclusion

This dissertation concludes with this chapter by presenting the final outcomes of the research and a summary of contributions made through this research.

1.6 Delimitations of Scope

There are various concurrency control mechanisms in existence in the field of Computer Science. This research was not intended at creating a new concurrency control mechanism. Instead it used thread pools as the concurrency control mechanism and implemented three algorithms that can adaptively adjust the thread pool size.

1.7 Conclusion

This chapter presented the background of the research problem area and explained the questions that this research study aims to find solutions to. These were then justified. The research methodology and the outline of the dissertation were also presented. With this foundation, the dissertation can continue with explaining the research study in greater detail in the subsequent chapters.

Chapter 2 - Literature Review

2.1 Background

2.1.1 Threads and thread Pools

A sequence of instructions executed within the context of a process is defined as a thread [4]. Using multiple threads of control for processing is defined as multithreading. Therefore multithreading separates a process into many threads of execution where each thread runs independently. Some of the benefits of multithreaded programming as explained in [4], [5] are as follows

- Multithreading increases responsiveness of a system.
  For example, a single threaded server processing a request would not listen for subsequent requests until one request has been processed and the reply is sent. In contrast, a multithreaded server could use a single thread to listen to requests and hand over processing to separate threads.

- Costs less compared to multiple processes
  A thread created within a process uses the same address space of the process. Therefore it is less expensive to create a new thread than creating a process. Furthermore, a switch of processes requires a switch of address space. Therefore, the time taken to switch between threads is less than the time taken to switch between processes.

A thread pool is a group of threads that are created before any work is assigned to them. All threads in a freshly created thread pool are idle. When tasks are ready to be executed on a thread, an idle thread is selected from this pool of threads and assigned to process the given task. Once the thread has completed processing the assigned task, the thread becomes idle and ready to accept and process more tasks. Tasks may be

assigned to idle threads in the pool until idle threads in the thread pool are exhausted. When this occurs, processing of new tasks must wait until a thread becomes idle.

2.2 Related work

In [6] Praphamontripong et al. show that it is necessary to analyze the performance of web servers prior to deployment. In their paper, a performance analysis methodology is presented to analyze the performance of an asynchronous web server. They show through experimental results that after a certain threshold, increase of the size of the event handler pools and queues does not provide significant performance improvements.

Behren et al. in [7] show that although people claim that thread based programming does not perform well, the reason for such low performance is not due to the threading paradigm, but due to shortcomings in specific threading implementations. In their paper they first explain the claims that people use to state that threads are not suitable to handle high concurrency servers and argue why those claims do not hold. They then use a threading library to show that threads actually perform well in servers that require high concurrency.

Each core in a processor may execute multiple threads. Although it is assumed that the number of threads should equal the number of cores in the processor to obtain better performance, Pusukuri et al. in [8] show that this does not hold true for processors having a high number of cores. Using the PARSEC benchmark on a 24 core computer, they experimentally show that this is not the case. The number of threads that were required to obtain the maximum speedup varied from 16 to 63 across different tests in the benchmark. As a solution to this, they propose the Thread Reinforcer framework, a framework for automatically selecting the near optimal or optimal number of threads. However, in a client server application, the amount of contentions that may occur for resources, the amount of concurrent users are factors that need to be considered to decide the number of threads. The Thread Reinforcer framework described here, runs the application multiple times by varying the number of threads, analyses performance

and then identifies the optimal number of threads. But this approach may not be practical for use in a client server application where the
performance is affected by the server load caused by the amount of concurrent users, the types of requests they are making etc.

The reactor pattern presented in [9] presents a design pattern that can be used to handle service requests that clients send concurrently to an application. Coplien and Schmidt in [9] argue that although it is possible to use multi-threading to implement such a scenario, thread synchronization problems, context switches, and difficulty of programming as threads require complex concurrency controls are some reasons why a multi threaded approach may not be preferred. (Fear of programmers regarding thread synchronization complexities prevent making use of the hardware capabilities of systems by using multi-threading). They present a methodology for handling multiple concurrent requests by multiplexing between different tasks by using a minimal number of threads. However a problem may occur with this approach if the few threads that are used, get blocked. In such a case it is necessary to ensure that the code that performs the tasks associated with providing services for the requests never get blocked or use a pool of threads to handle blocking operations.

Harrison et al. Identifies several issues in conventional concurrency models in [10]. They argue that in situations where hardware parallelism exist, the Reactor design pattern is not the most efficient. They show that due to the use of minimal number of threads in the Reactor pattern, it increases the complexity of programming as programmers are burdened with ensuring that the code does not get blocked when handling the request of a particular client. Programmers using the Reactor pattern must also ensure that all the clients sending requests to the server share the thread by performing non-blocking operations which use a short duration of time. As a solution to these problems, they present the Proactor pattern to design asynchronous servers which could handle high concurrency. Proactor pattern utilizes the I/O capabilities of the OS and therefore does not require a very high number of threads. However a disadvantage of the Proactor pattern is that programs written using it can be hard to debug.
The Java programming language is currently a popular programming language that is used to develop many client server applications. It is a commonly used language to

develop multithreaded applications. Chen et al. in [11] presents an evaluation of Java multithreading performance by using different number of processor cores and application threads with the HotSpot Java Virtual Machine with OpenJDK 1.7. Experiments conducted using multithreaded benchmarks show that different results were obtained based on different benchmarks used. For JGF benchmarks MolDyn, MonteCarlo and RayTracer, the peak in throughput was observed when the number of cores used was equal to the number of threads used. Also for JGF benchmarks and SPECjbb2005 [12] benchmarks, higher throughput was observed with more cores and threads. Experiments conducted to study lock contentions which are caused by multiple threads trying to access the same shared object showed that they could reduce the performance of the system. However, it should be noted that tuning of the JVM has shown to provide some performance improvements in their experiments.

While threads can be used to provide concurrency to applications, the performance of threads differ depending on the kernel in use. This was shown by Gu et al. In [13]. They show that although the Java Virtual Machine (JVM) makes Java programs portable across environments, the performance of threads differ because Java threads have to be mapped to native system threads. Using the EP benchmark [14], they experimented with thread behavior with respect to their creation and computation performance under different workloads. Through these experimental results, it is shown that the use of a large number of threads should be avoided. The reason for this is the thread management overhead incurred when there is a large number of threads.

In [15] Hu et al. show that to achieve maximal performance on different operating system platforms, it is necessary to utilize different I/O mechanisms provided by them instead of using common I/O mechanisms. This is due to the fact that the implementations of I/O mechanisms specific to a particular platform may have been optimized for that platform itself. They also show that the choice of concurrency strategy has a major impact on web server performance. Experimental results obtained using JAWS show that the throughput of each connection decreases when the number of connections made per second increases and that the latency increases as the connections made per second increases. Their results show that concurrency mechanisms have a great impact on latency and throughput. Such I/O operations and some other operations happening within a server may require access to shared

resources. Most notably these include things such as access to Databases and files. If multiple instances of an application tries to access these shared resources, resource contentions may occur which would degrade server performance. Zheng et al. In [16] address these issues by proposing a method to detect such contentions. Their technique was tested experimentally by using several large web applications. Results presented show that the technique is effective. However, a problem with this presented technique is that it sometimes provides false positives and false negatives in the results.

Database Management Systems (DBMS) employ concurrency control mechanisms. As computers are moving towards architectures making use of a large number of cores on a single chip, applications running on them should be able to make use of the available hardware resources in order to do computations efficiently. DBMSs being a commonly used type of application must also be able to scale well to such many core environments. Yu et. al. in [17] argue that Database Management Systems currently in use are not scalable to many core computers. By implementing several concurrency control algorithms to handle concurrency in a DBMS running in main memory, they have analyzed how these algorithms scale. Results show that all algorithms used did not scale well and therefore did not provide good performance.

Using multiple threads of execution in an application incurs certain costs. According to Goetz in order to obtain an improvement in performance when using multiple threads, the performance benefits of using multiple threads must outweigh that of the overheads introduced by concurrency. When a thread gets blocked, the Java Virtual Machine suspends the thread and allows that thread to be switched out during a context switch. Therefore if a program causes a thread to block frequently, more context switches will occur and this thread will not be able to make full use of the scheduling quantum used. Therefore when threads have to execute many I/O bound operations, scheduling overhead increases and throughput decreases rather than when executing CPU bound operations.

When using multiple threads of execution, there may be sections of code that should not be accessed by multiple threads of execution simultaneously. Access to these critical sections must be controlled so that only one thread of execution has access to them at any given time.

Traditionally this was achieved using locks. When a thread of execution needs access to such a critical section a lock is acquired and released after processing that section is complete. Only the thread holding the lock is allowed to access a given critical section. Acquiring and releasing locks in a program incurs additional overhead. An alternative to locks to handle concurrency is Software Transactional Memory (STM) [18]–[20]. With STM, instead of acquiring and releasing locks, sections of code are designated as transactions. Such sections of code that are designated as transactions will have atomicity, isolation and consistency. Therefore STM is somewhat similar to database transactions. STM is a software based implementation and it provides a non-blocking mechanism to handle concurrency. Saha et al. in [21] presents a performance evaluation of various STM implementations and locking mechanisms and presents a STM system that is a part of multi-core runtime called McRT. The paper presents performance evaluations made between locks and their STM implementation for a hashtable benchmark, binary search tree benchmark (both balanced and unbalanced) and that of a linked list benchmark (both sorted and unsorted). Results show that STM performs better than locking mechanisms for higher number of processors. Saha et al. also shows the performance of STM on non-synthetic workloads as well. For this purpose, the sendmail application had been used with McRT-STM and tested with upto 8 threads. Results obtained show that the performance of STM and locking are comparable. Zhang et al. in [22] presents a STM implementation named LarkTM which provides low overhead. The adaptive version of LarkTM, named LarkTM-S has been shown to outperform other high-performance Software Transactional Memory implementations. Zhou et. al. in [23] show the importance of thread mapping through feedback loops and regulating parallelism online to improve performance of Transactional Memory applications. Their approach adds a time stall when the commit of the transaction occurs and as a result this may show slight performance difference with ones that do not use it. This stall time is added to make the management of contentions work similar to a back off policy.

2.3 Conclusion

From the current literature in these areas of concurrency control, it can be seen that a study about workload characteristics of different application types will be useful and that a methodology to automatically identify and adaptively adjust the number worker threads based on performance metrics will provide numerous benefits.

Chapter 3 - Design

3.1 System Architecture

Experiments were conducted to study the behavior of a system under different applications and incoming workloads during the course of this study. Following test setup was utilized for this purpose.

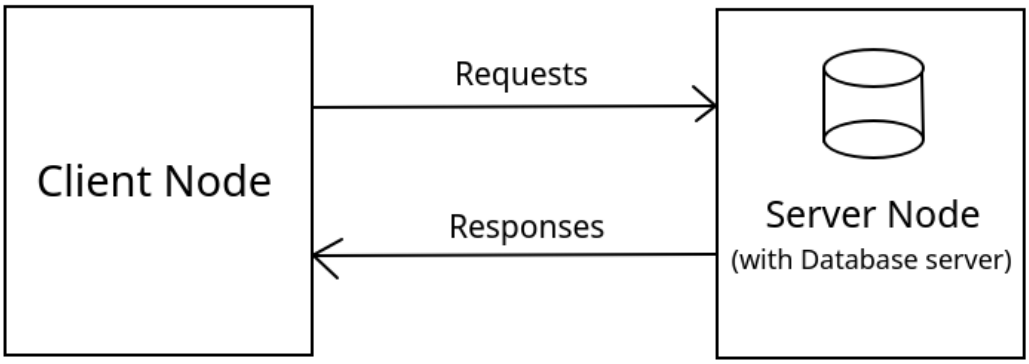

Figure 3.1: Architecture of the test setup

The client node and server node were two physical nodes that were dedicated for this research project. Apache JMeter [3] running on the client node sent requests to a Netty [2] HTTP Server running on the server node. The HTTP Server processed requests and sent the replies back to the client node. Some applications has database operations and in such situations, the server node utilize the database server running on it.

Details about the client node and the server node are as presented in table 3.1.

Table 3.1: Information about the nodes

	Client node	Server node
CPU	Intel Xeon E5506 @ 2.13GHz	Intel Xeon 5160 @ 3.00GHz
RAM	8GB	16GB
Network interface	Intel Corporation 82576 Gigabit Ethernet Controller Capacity - 1Gbit/s	Broadcom Corporation NetXtreme II BCM5708 Gigabit Ethernet Controller Capacity - 1Gbit/s
Operating System	Ubuntu 16.04.1 LTS	Ubuntu 16.04.1 LTS
OS Kernel version	4.4.0-128-generic	4.4.0-128-generic

3.2 Performance metrics

Throughput and latency were used to evaluate performance of the server. Throughput is defined as the number of requests processed per second and latency is defined as the total time taken to receive a reply since a request leaves the client.

Therefore in order to study performance characteristics of different applications and incoming workloads, throughput, mean latency and 99^{th} percentile of latency was measured during the execution of tests.

Furthermore, these three performance metrics are also used to drive the the thread pool size adjustment decisions.

3.3 Applications

Several applications were created and tested during the course of this study. The purpose of creating such applications and testing them was to study how thread pool sizes and concurrent users behave under different applications. These can be divided into two categories as CPU dominant workloads where a significant amount of

computations are utilizing the CPU and I/O dominant workloads where a significant amount of operations are input/output operations. iostat [24] and pidstat [25] utilities on were used to analyze and verify the CPU utilization and I/O utilization of these applications.

The details of the application are as follows.

I/O dominant

1. Database write operation

   Database write operations are common in real world applications. This is an I/O dominant operation as the database needs to be accessed. Multiple writes cannot occur concurrently. The database server used for theses tests is MySQL. For each request received, this test writes the current timestamp to the database. This application shall be referred as 'DbWrite' in the rest of this dissertation.

2. Database read operation

   Another frequently performed database operation is to read values from a database. This is also an I/O dominant operation. Similar to the previous test, the database server used for theses tests is MySQL. For each request received, a random row of a database table containing timestamps is selected and sent as the reply. This application shall be referred as 'DbRead' in the rest of this dissertation.

MySQL 5.7.24-0ubuntu0.16.04.1 was used as the underlying database on the Server node for the above I/O bound experiments.

CPU Dominant

3. Prime 1 million

   Prime 10k test is a primality test. For each request received, a pseudo-random number between 1,000,000 and 1,000,020 was generated and whether or not this number was a prime number was checked. This application shall be referred as 'Prime1m' in the rest of this dissertation.

4. Prime 10 million

   Prime 100k test is also a primality test but which requires more computations. For each request received, a pseudo-random number between 10,000,000 and 10,000,020 was generated and whether or not this number was a prime number was checked. This application shall be referred as ‘Prime10m’ in the rest of this dissertation.

The purpose of the above CPU dominant tests was not be perform primality checks using the most efficient algorithm available at the time, but to run tests which would be having various CPU intensity levels in order to simulate CPU bound workloads of varying CPU intensities. Therefore the following naive algorithm was used to check for primality.

```
ALGORITHM - CheckPrime()
INPUT: low, high
OUTPUT: true/false

1:     number = generate pseudo-random number between low and high
2:     FOR each integer i from 2 to number
3:           IF number mod i is 0 THEN
4:                 RETURN false
5:           END IF
6:     END FOR
7:     RETURN true
```

The input values low and high for Prime1m were 1,000,000 and 1,000,020 respectively while the input values low and high for Prime10m were 10,000,000 and 10,000,020 respectively.

3.4 Software Tools

3.4.1 Apache JMeter

Apache JMeter [3] is an open source load testing tool written in Java. Apache JMeter version 4.0 was used in this project to simulate concurrent users and generate requests by running on the client node. In order to test the server, HTTP requests were generated by Apache JMeter. As recommended by the user manual, the GUI mode of

JMeter was used only to create the tests. This is due to the fact that non-GUI mode consumes less resources than GUI mode. Therefore, all tests required for this project were executed in Command-line mode (non-GUI mode). These two modes of JMeter are depicted below in figure 3.2 and 3.3 respectively.

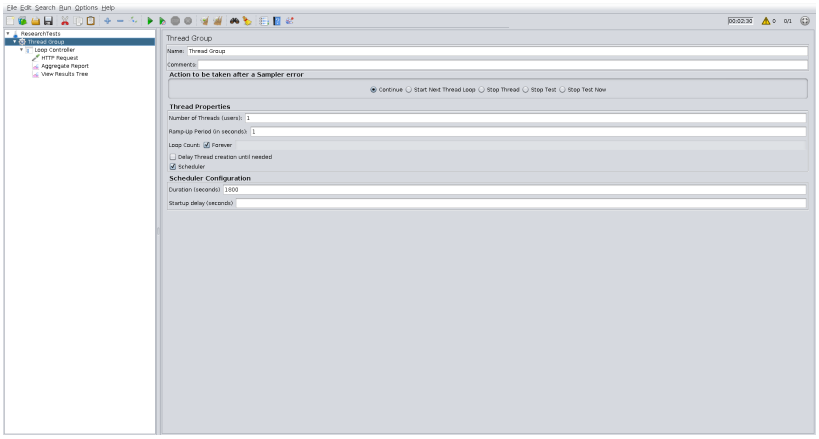

Figure 3.2: Apache JMeter GUI-mode

```
Creating summariser <summary>
Created the tree successfully using /home/client/singleConcurrency.jmx
Starting the test @ Thu Dec 20 15:55:02 IST 2018 (1545301502529)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +      1 in 00:00:01 =    1.0/s Avg:   823 Min:   823 Max:   823 Err:     0 (0.00%) Active: 517 Started: 517 Finished: 0
summary +  12103 in 00:00:26 =  467.8/s Avg:  2079 Min:   793 Max:  4486 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  12104 in 00:00:27 =  449.8/s Avg:  2079 Min:   793 Max:  4486 Err:     0 (0.00%)
summary +  19567 in 00:00:30 =  652.2/s Avg:  1520 Min:   654 Max:  3445 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  31671 in 00:00:57 =  556.5/s Avg:  1734 Min:   654 Max:  4486 Err:     0 (0.00%)
summary +  24106 in 00:00:30 =  803.6/s Avg:  1255 Min:   524 Max:  4189 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  55777 in 00:01:27 =  641.8/s Avg:  1527 Min:   524 Max:  4486 Err:     0 (0.00%)
summary +  15641 in 00:00:31 =  507.2/s Avg:  1835 Min:   601 Max:  4341 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  71418 in 00:01:58 =  606.5/s Avg:  1594 Min:   524 Max:  4486 Err:     0 (0.00%)
summary +  21554 in 00:00:29 =  739.2/s Avg:  1333 Min:     2 Max:  6893 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  92972 in 00:02:27 =  632.9/s Avg:  1534 Min:     2 Max:  6893 Err:     0 (0.00%)
summary +  16008 in 00:00:31 =  514.6/s Avg:  1937 Min:     7 Max:  6832 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 108980 in 00:02:58 =  612.2/s Avg:  1593 Min:     2 Max:  6893 Err:     0 (0.00%)
summary +  21194 in 00:00:29 =  733.5/s Avg:  1326 Min:     1 Max:  7191 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 130174 in 00:03:27 =  629.1/s Avg:  1549 Min:     1 Max:  7191 Err:     0 (0.00%)
summary +  14959 in 00:00:32 =  468.1/s Avg:  2159 Min:     1 Max:  7094 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 145133 in 00:03:59 =  607.6/s Avg:  1612 Min:     1 Max:  7191 Err:     0 (0.00%)
summary +  20696 in 00:00:28 =  731.2/s Avg:  1331 Min:     3 Max:  6965 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 165829 in 00:04:27 =  620.7/s Avg:  1577 Min:     1 Max:  7191 Err:     0 (0.00%)
summary +  14482 in 00:00:30 =  487.0/s Avg:  2285 Min:     2 Max:  6999 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 180311 in 00:04:57 =  607.3/s Avg:  1634 Min:     1 Max:  7191 Err:     0 (0.00%)
summary +  20563 in 00:00:31 =  666.6/s Avg:  1355 Min:     2 Max:  6955 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 200874 in 00:05:28 =  612.9/s Avg:  1605 Min:     1 Max:  7191 Err:     0 (0.00%)
summary +  14004 in 00:00:29 =  480.2/s Avg:  2283 Min:     2 Max:  6884 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 214878 in 00:05:57 =  602.0/s Avg:  1650 Min:     1 Max:  7191 Err:     0 (0.00%)
summary +  20078 in 00:00:32 =  627.2/s Avg:  1422 Min:     1 Max:  7080 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 234956 in 00:06:29 =  604.1/s Avg:  1630 Min:     1 Max:  7191 Err:     0 (0.00%)
summary +  12512 in 00:00:28 =  447.2/s Avg:  2414 Min:     2 Max: 10132 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary = 247468 in 00:06:57 =  593.6/s Avg:  1670 Min:     1 Max: 10132 Err:     0 (0.00%)
summary +  20584 in 00:00:30 =  685.8/s Avg:  1425 Min:     2 Max:  7183 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
```

Figure 3.3: Apache JMeter non-GUI mode

3.4.2 Netty

Netty [2] is an open source client-server framework that can be used to create protocol servers and clients in Java. This framework has been used in this research project to implement an HTTP server which runs on the server node.

Netty has two types of thread pools namely the boss thread pool and worker thread pool. The boss thread pool's job is to accept incoming connections. Once a connection is accepted by the boss thread pool, it is passed on to a thread in the worker thread pool to be processed.
Netty version 4.1.22 was used to implement the test server.

3.4.3 Dropwizard Metrics

Dropwizard metrics [26] is an open source framework that can be used to measure the behavior of applications. It has been used to measure the performance metrics on the server which are used to drive thread pool size adjustment decisions.
Dropwizard metrics version 3.1.0 was used during implementation of the test server.

3.4.4 Implementation language

The Java programming language [27] is a popular programming language used at present and it was selected as the language of choice for this research study. It was used in the implementation of the HTTP server created for conducting experiments, implementation of the test applications and for the implementation of the proposed algorithms. Oracle Java [28] version 1.8.0_172 was used during this research.

3.5 Experiments

The experiments performed during this study were divided into 3 stages. All experiments were conducted for 25 minutes.

3.5.1 Static worker thread pool size with fixed workload simulation

The initial set of experiments of this study were performed to gain insight about the effects of the size of the thread pool on the two performance metrics observed, throughput and latency. All four applications created were tested during these experiments. In these experiments, the number of worker threads of the server were kept at a constant value. The number of concurrent users simulated was also kept at a constant value in order to simulate a fixed workload.
Ex. DbWrite application with a worker thread pool size of 10 and 100 concurrent users

Table 3.2 lists the combinations of applications, worker thread pool sizes and the number of concurrent users that tested during this stage of experiments.

Table 3.2: Fixed worker thread pool size with fixed load simulations conducted

Application	**Worker thread pool size**	**Number of concurrent users**
DbRead	2	1
DbRead	2	10
DbRead	2	100
DbRead	4	1
DbRead	4	10
DbRead	4	100
DbRead	10	1
DbRead	10	10
DbRead	10	100
DbRead	50	1
DbRead	50	10
DbRead	50	100
DbRead	100	1
DbRead	100	10
DbRead	100	100
DbWrite	2	1
DbWrite	2	10
DbWrite	2	100
DbWrite	4	1

DbWrite	4	10
DbWrite	4	100
DbWrite	10	1
DbWrite	10	10
DbWrite	10	100
DbWrite	50	1
DbWrite	50	10
DbWrite	50	100
DbWrite	100	1
DbWrite	100	10
DbWrite	100	100
Prime1m	2	1
Prime1m	2	10
Prime1m	2	100
Prime1m	4	1
Prime1m	4	10
Prime1m	4	100
Prime1m	10	1
Prime1m	10	10
Prime1m	10	100
Prime1m	50	1
Prime1m	50	10
Prime1m	50	100
Prime1m	100	1
Prime1m	100	10
Prime1m	100	100
Prime10m	2	1
Prime10m	2	10
Prime10m	2	100
Prime10m	4	1
Prime10m	4	10
Prime10m	4	100
Prime10m	10	1
Prime10m	10	10
Prime10m	10	100
Prime10m	50	1
Prime10m	50	10

Prime10m	50	100
Prime10m	100	1
Prime10m	100	10

3.5.2 Dynamic worker thread pool size with fixed workload simulation

Based on the knowledge gained about the effects of thread pool size on throughput and latency, three optimization algorithms were developed. The purpose of this set of experiments was to test the developed algorithms' ability to dynamically adjust the thread pool size and improve performance. The incoming workload to the server was kept at a fixed value (Eg. 100 concurrent users) during these experiments.

3.5.3 Dynamic worker thread count with varying workload simulation

Servers used in the real world do not receive requests with the same number of concurrent users all the time. The number of concurrent users requesting services from a server varies with time making the workload on the server a varying one. In order to test the performance of the algorithms against such varying workloads, a dynamic load was simulated in Apache JMeter [3]. Figure 3.4 below shows how the number of concurrent users varied over time in this varying workload simulation.

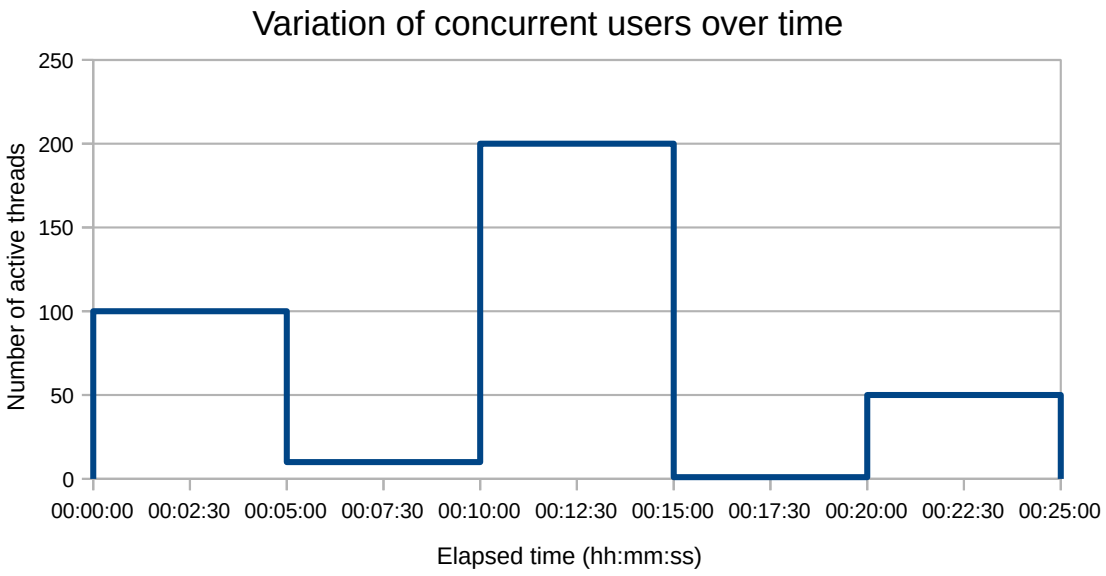

Figure 3.4: Varying workload simulated

Chapter 4 - Implementation

4.1 Thread Pool Size adjustment algorithms

The goal of this project was to develop a methodology to adjust the thread pool sizes on the server automatically. To do so three algorithms were developed which are

1. Throughput optimization algorithm
2. Mean Latency optimization algorithm
3. 99^{th} Percentile of Latency optimization algorithm

4.1.1 Throughput optimization algorithm

The throughput optimization algorithm developed during the course of this study is aimed at optimizing the throughput by adjusting the worker thread pool size. This algorithm measures the throughput every 10 seconds and makes decisions based on it. The throughput optimization algorithm is presented below.

```
ALGORITHM - optimizeThroughput()
INPUT: previousThroughput, currentThroughput
OUTPUT: N/A

1.     Initialize INC_ITR, INC_CHECK_ITR to false
2.     Initialize HAS_STARTED to false
3.     Initialize throughputTable
4.     REPEAT every 10 seconds
5.           IF ( HAS_STARTED == false and NumberOfRequests > 0 ) THEN
6.                 HAS_STARTED = true
7.                 INC_ITR = true
8.           END IF
9.           IF ( DEC_ITR == false or ( DEC_ITR == true and DEC_IMPROVED == false )
             and throughputDecrement > 10% ) THEN
10.                sizeFound = findThreadPoolSize(currentThroughput)
11.                Set thread pool size to sizeFound
12.          IF ( INC_ITR == true and INC_IMPROVED == true ) THEN
13.                increase thread pool size by 10
14.          END IF
15.          IF (INC_CHECK_ITR == true and INC_IMPROVED == true ) THEN
16.                IF Throughput increment < 10% THEN
17.                      INC_IMPROVED = false
18.                      decrease thread pool size by 10
19.                      incrementLock = 6
20.                END IF
21.          END IF
22.          updateThroughputTable(currentThroughput, currentThreadPoolSize)
23.          IF incrementLock > 0 THEN
24.                incrementLock = incrementLock -1
25.          ELSE IF incrementLock == 0 THEN
26.                INC_IMPROVED = true
27.          END IF
28.          IF decrementLock > 0 THEN
29.                decrementLock = decrementLock -1
30.          ELSE IF decrementLock == 0 THEN
31.                DEC_IMPROVED = true
32.          END IF
33.          IF current iteration is INC_ITR THEN
34.                set INC_CHECK_ITR as next iteration
35.          ELSE IF current iteration is INC_CHECK_ITR THEN
36.                set INC_ITR as next iteration
37.          END IF
38.    END
```

During the initial set of experiments with a fixed thread pool size and fixed load, it was observed that throughput increases with the thread pool size. The throughout optimization algorithm developed through this study utilize a table (named throughputTable in the above Pseudocode) to record the mean throughput obtained at each thread pool size during the server's execution. For each thread pool size the server uses, the throughput table records the mean throughput and the number of 10 second intervals during which the server remained at the given thread pool size (named count). The throughput table is sorted by the recorded thread pool sizes. The abstract view of the throughput table is presented below. Thereafter when the algorithm detects a decrease in throughput, it refers to throughputTable to find the value of the thread

pool size to use to provide this detected level of throughput. This value is found by using the findThreadPoolSize() function.

Table 4.1: Abstract view of throughputTable

Thread pool size	Mean throughput	Count

```
ALGORITHM - findThreadPoolSize()
INPUT: currentThroughput
OUTPUT: threadPoolSize

1: FOR  i < number of rows in throughputTable DO
2:     mean = ( row(i).getThroughput + row(i+1).getThroughput )/2
3:     IF (currentThroughput <= mean ) THEN
4:           RETURN row(i).getThreadPoolSize
5:     END IF
6: END FOR
```

```
ALGORITHM - updateThroughputTable()
INPUT: currentThroughput, currentThreadPoolSize

1.     threadPoolSizeExists = false
2.     FOR i < number of rows in throughputTable DO
3.           IF row(i).getThreadPoolSize == currentThreadPoolSize THEN
4.                 threadPoolSizeExists = true
5.                 value = row(i).getThroughput
6.                 count = row(i).getCount
7.                 newValue=((value*count)+ currentThroughput)/count+1
8.                 row(i).setThroughput(newValue)
9.           END IF
10.    END FOR
11.    IF threadPoolSizeExists is false THEN
12.                create new table entry x
13.                x.setCount(1)
14.                x.setThreadPoolSize(currentThreadPoolSize)
15.                x.setThroughput(currentThroughput)
16.                add entry x to throughputTable
17.                sort throughputTable by threadPoolSize
18.    END IF
```

4.1.2 Mean Latency optimization algorithm

This algorithm measures the mean latency of the requests every 10 seconds in the server and adjusts the worker thread pool size accordingly to minimize the mean latency. The algorithm used is as follows

```
ALGORITHM - meanLatencyOptimization()
INPUT: oldMeanLatency, currentMeanLatency
OUTPUT: N/A

1.     Initialize INC_ITR, INC_CHECK_ITR, DEC_ITR, DEC_CHECK_ITR to false
2.     Initialize HAS_STARTED to false
3.     REPEAT every 10 seconds
4.            IF ( HAS_STARTED == false and NumberOfRequests > 0 ) THEN
5.                   HAS_STARTED = true
6.                   INC_ITR = true
7.            END IF
8.            IF ( INC_ITR == true and INC_IMPROVED == true ) THEN
9.                   increase thread pool size by 10
10.           END IF
11.           IF (INC_CHECK_ITR == true and INC_IMPROVED == true ) THEN
12.                  IF mean latency decrement < 5% THEN
13.                         INC_IMPROVED = false
14.                         decrease thread pool size by 10
15.                         incrementLock = 6
16.                  END IF
17.           END IF
18.           IF ( DEC_ITR == true and DEC_IMPROVED == true ) THEN
19.                  decrease thread pool size by 10
20.           END IF
21.           IF (DEC_CHECK_ITR == true and DEC_IMPROVED == true ) THEN
22.                  IF (mean latency decrement < 5%) THEN
23.                         DEC_IMPROVED = false
24.                         increase thread pool size by 10
25.                         decrementLock = 6
26.                  END IF
27.           END IF
28.
29.           IF incrementLock > 0 THEN
30.                  incrementLock = incrementLock -1
31.           ELSE IF incrementLock == 0 THEN
32.                  INC_IMPROVED = true
33.           END IF
34.           IF decrementLock > 0 THEN
35.                  decrementLock = decrementLock -1
36.           ELSE IF decrementLock == 0 THEN
37.                  DEC_IMPROVED = true
38.           END IF
39.           IF current iteration is INC_ITR THEN
40.                  set INC_CHECK_ITR as next iteration
41.           ELSE IF current iteration is INC_CHECK_ITR THEN
42.                  set DEC_ITR as next iteration
43.           ELSE IF current iteration is DEC_ITR THEN
44.                  set DEC_CHECK_ITR as next iteration
45.           ELSE IF current iteration is DEC_CHECK_ITR THEN
46.                  set INC_ITR as next iteration
47.           END IF
48.    END
```

4.1.3 99th Percentile of latency optimization algorithm

This algorithm measures the 99th percentile of latency of the requests every 10 seconds in the server and adjusts the worker thread pool size accordingly to minimize the 99th percentile of latency. The algorithm used is as follows.

```
ALGORITHM - 99PercentileLatencyOptimization()
INPUT: old99PcerntileLatency, current99PercentileLatency
OUTPUT: N/A

1.     Initialize INC_ITR, INC_CHECK_ITR, DEC_ITR, DEC_CHECK_ITR to false
2.     Initialize HAS_STARTED to false
3.     REPEAT every 10 seconds
4.            IF ( HAS_STARTED == false and NumberOfRequests > 0 ) THEN
5.                   HAS_STARTED = true
6.                   INC_ITR = true
7.            END IF
8.            IF ( INC_ITR == true and INC_IMPROVED == true) THEN
9.                   increase thread pool size by 10
10.           END IF
11.           IF (INC_CHECK_ITR == true and INC_IMPROVED == true ) THEN
12.                  IF 99th percentile latency decrement < 5% THEN
13.                         INC_IMPROVED = false
14.                         decrease thread pool size by 10
15.                         incrementLock = 6
16.                  END IF
17.           END IF
18.           IF ( DEC_ITR == true and DEC_IMPROVED == true ) THEN
19.                  decrease thread pool size by 10
20.           END IF
21.           IF (DEC_CHECK_ITR == true and DEC_IMPROVED == true) THEN
22.                  IF (99th percentile latency decrement < 5%) THEN
23.                         DEC_IMPROVED = false
24.                         increase thread pool size by 10
25.                         decrementLock = 6
26.                  END IF
27.           END IF
28.
29.           IF incrementLock > 0 THEN
30.                  incrementLock = incrementLock -1
31.           ELSE IF incrementLock == 0 THEN
32.                  INC_IMPROVED = true
33.           END IF
34.           IF decrementLock > 0 THEN
35.                  decrementLock = decrementLock -1
36.           ELSE IF decrementLock == 0 THEN
37.                  DEC_IMPROVED = true
38.           END IF
39.           IF current iteration is INC_ITR THEN
40.                  set INC_CHECK_ITR as next iteration
41.           ELSE IF current iteration is INC_CHECK_ITR THEN
42.                  set DEC_ITR as next iteration
43.           ELSE IF current iteration is DEC_ITR THEN
44.                  set DEC_CHECK_ITR as next iteration
45.           ELSE IF current iteration is DEC_CHECK_ITR THEN
46.                  set INC_ITR as next iteration
47.           END IF
48.    END
```

The algorithms have the following common characteristics

Repetition period

The outermost repeat loop of the algorithms repeat every 10 seconds. All proposed algorithms use this period between iterations.

Improvement measurement

The algorithms measure improvements of the relevant performance metrics during execution in order to decide if thread pool size changes were worthwhile.

An improvement of 5% of mean latency was selected as the minimum acceptable improvement for mean latency. Similarly an improvement of 5% of the 99th percentile of latency was selected as the minimum acceptable improvement for the 99th percentile of latency optimization algorithm. For the throughput optimization algorithm, an improvement of 10% of throughput is minimum acceptable improvement to increase the thread pool size.

Periodic iterations

The proposed algorithm repeats every 10 seconds. The algorithms have designated iterations as follows.

- INC_ITR – Iteration during which the thread pool size is increased
- INC_CHECK_ITR – This iteration checks if the thread pool size increment in the previous iteration has made an improvement to the performance metric in consideration.
- DEC_ITR - Iteration during which the thread pool size is decreased
- DEC_CHECK_ITR - This iteration checks if the thread pool size decrement in the previous iteration has made an improvement to the performance metric in consideration.

The throughput optimization algorithm iterates between INC_ITR and INC_CHECK_ITR while the mean latency optimization algorithm iterations between

INC_ITR, INC_CHECK_ITR, DEC_ITR and DEC_CHECK_ITR. These iterations limit the increment/decrement of thread pool size and evaluating whether the increments/decrements provided an improvement to specific periods. Limiting the steps of the algorithm to such designated iterations helps to minimize the effects of outliers from affecting thread pool size adjustment decisions. This is because the probability that outliers always occur at a given stage is lower than the probability of an outlier occurring every 10 seconds.

Increment/Decrement lock

During INC_CHECK_ITR iteration and DEC_CHECK_ITR iteration, the system checks if the increments and decrements that were made in the previous iteration provided the minimum required improvement of the performance metric. If such an improvement is not detected during the INC_CHECK_ITR iteration, the algorithm temporarily locks thread pool size increments for a duration of 1 minute. Similarly if improvement is not detected during the DEC_CHECK_ITR iteration, the algorithm temporarily locks thread pool size decrements for a duration of 1 minute.
The rationale behind using such locks on thread pool size increments and decrements is to prevent the algorithm from increasing and decreasing thread pool sizes unnecessarily if they do not provide an improvement in performance. However, these locks are only temporary and expire one minute after setting them. This is because, incoming workloads may change from time to time and when they do, thread pool size need to be changed accordingly.

4.2 Performance Metric collection

In order to dynamically adjust the thread pool size, it was necessary to obtain performance metrics from the server itself instead of measuring them from the client side. Therefore Dropwizard metrics [26] was used to gather performance metrics on the server when tests were executed. The Dropwizard metrics library use several components to obtain performance metrics. A meter is a component which measures the rate at which a set of events occur. A histogram measures the distribution of values

in a stream of data. A timer is another component which is a histogram of the duration of event and a meter of the rate of it's occurrence.

The traditional method of calculation of quantiles is to obtain the entire data set, sort it and take the values (ie – 1% from the end is taken for the 99th percentile). Although this works for small data sets, it is not suitable for high-throughput, low-latency services. The solution to this problem is to use a technique called reservoir sampling. Here, the data is sampled as it passes through. A reservoir is statistically representative of the data stream as a whole and Dropwizard metric's Histograms use reservoirs. The default type of reservoir used by Timers is called an Exponentially Decaying Reservoir and it produces quantiles which are representative of (roughly) the last 5 minutes of data. However Exponentially Decaying Reservoirs have shortcomings. They are lossy as they do not store every sample. They are only statistically representative. These shortcomings of Exponentially Decaying Reservoirs could lead to inaccurate measurements due to the inclusion of much older samples.

High Dynamic Range Histogram (HdrHistogram)[29] is a lossless histogram implementation which addresses shortcomings in Exponentially Decaying Reservoirs. Therefore HdrHistogram was found to be more suitable to be used to collect performance metrics in the server that was created. However, HdrHistogram was not available to be used in Dropwizard metrics at the time of this research project. This problem was solved by using the rolling-metrics library [30] which combines HdrHistogram with Dropwizard metrics. Version 2.0.4 of the rolling-metrics library was used during implementation.

Dropwizard metrics was also used to measure throughput on the server. Dropwizard metrics by default only provided methods to obtain 1, 5 and 15 minute throughput. However the thread thread pool size adjustment algorithm that was used was designed to run periodically every 10 seconds and make decisions about the thread pool size. The minimum default throughput of 1 minute would not have provided updated throughput values . Therefore, the Dropwizard metrics the source code of the Dropwizard metrics framework was modified to obtain 10 second throughput values.

4.3 Netty thread pools

As explained in Chapter 3, Netty has a boss thread pool and a worker thread pool. During implementation it was identified that the thread pool size of the worker thread pool of Netty cannot be changed dynamically as required when the HTTP server is running. Since the objective of this study is to dynamically adjust the thread pool size used for processing requests, this default architecture of Netty was found to not be adequate. Therefore the HTTP server was implemented in such a way that the requests that are received by the worker thread pool of Netty are passed onto a separate thread pool without processing in the Netty worker pool. The size of this separate thread pool can be adjusted while the server is running.

Chapter 5 - Results and Evaluation

5.1 Experimental results

The following section presents and explain the results that were obtained through the experiments that were conducted during this research study. As explained previously in Chapter 1 and Chapter 3, three sets of experiments were conducted and they were

1. Fixed worker thread pool size with fixed workload simulations
2. Dynamic worker thread pool size with fixed workload simulations
3. Dynamic worker thread pool size with varying workload simulations

5.1.1 Fixed worker thread pool size with fixed workload simulations

An initial set of experiments were conducted to gain insight into the effect of worker thread pool size on the performance metrics that were measured. These experiments made use of a fixed worker thread pool size and a fixed incoming workload. The proposed algorithms were not used in this set of experiments. This set of experiments were conducted with the sole purpose of gaining more insight into the effects of worker thread pool size on performance.

The results obtained through these tests showed that the performance that can be obtained by a given number of threads is not the same across different applications and workloads.

CPU dominant Prime10m application and I/O dominant DbWrite application were two applications tested. Figure 5.1 shows the variation of throughput over different worker thread pool sizes for these two applications. It can be observed that when the worker thread pool size was varied between 2, 4, 10, 50 & 100, the Prime1m application did not show a drastic change in throughput. However for the DbWrite application, using

thread pool sizes of 10, 50 and 100 provided much better throughput than when using a worker thread pool size of 2 or 4.

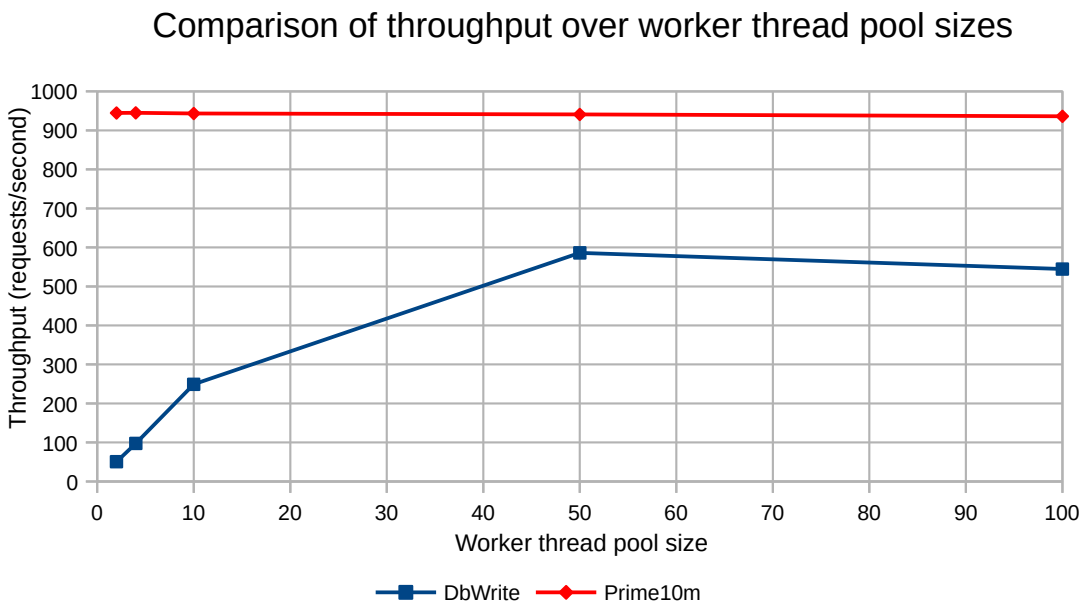

Figure 5.1: Throughput comparison of Dbwrite and Prime10m

However, analysis of the 99th percentile of latency showed that an increase of worker thread pool size provided opposite results for the two tests as depicted in figure 5.2. It was observed that as the worker thread pool size increased, latency increased for the Prime10m application. However, the increase in worker thread pool size decreased the latency experienced by the DbWrite application. This observation can be explained as follows. Database servers utilize a pool of threads to perform database operations. From the point of the HTTP server, I/O operations such as database writes are blocking operations. Therefore when a small number of worker threads were used by the HTTP server in the DbWrite application, they were unable to issue requests for database writes to the database server's thread pool until previous requests completed. This caused the an increase in latency for small worker thread pool sizes.

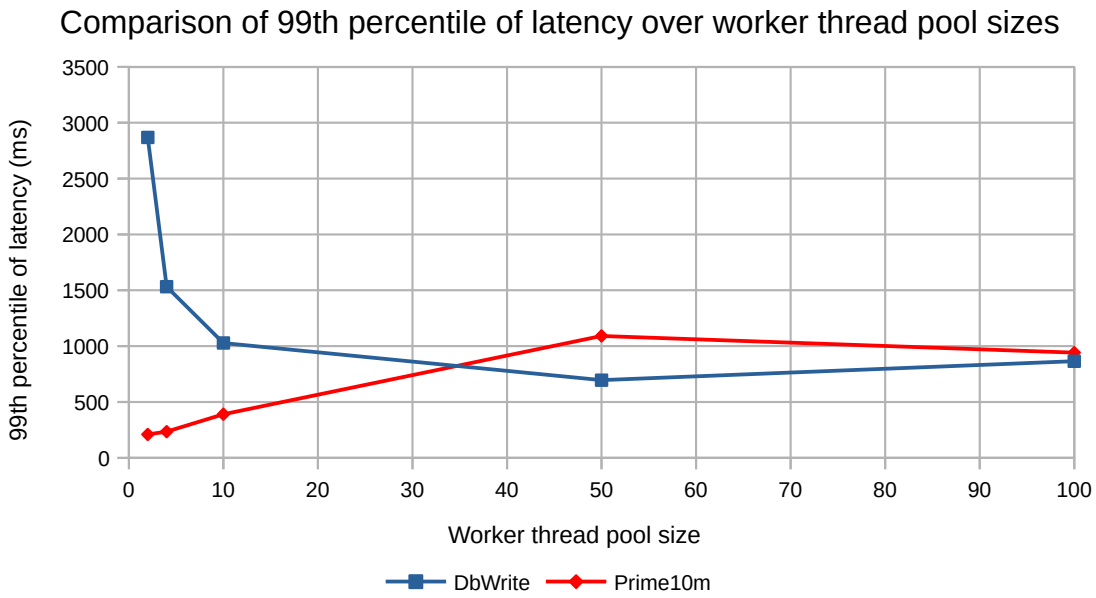

Figure 5.2: 99th percentile of latency comparison for DbWrite and Prime10m

Therefore, as the throughput did not show a significant increase for the Prime10m application when the worker thread pool size was increased, a small worker thread pool size provided better performance due to low latency.

Analysis of CPU utilization and disk I/O utilization showed that the Prime10m test was indeed CPU dominant. It showed high CPU utilization and low disk I/O utilization. In contrast, the DbWrite application showed that it was I/O dominant due to high disk I/O utilization and low CPU utilization. A graph depicting the CPU and I/O utilization for a period of 10 minutes for 2 worker threads and 10 concurrent users for these two tests are shown in figure 5.3 and figure 5.4 respectively. The Prime10m I/O graph shows a write rate of several Kilobytes per second, due to logging of data during experiments. However this was negligible compared to the disk I/O utilization by the DbWrite application.

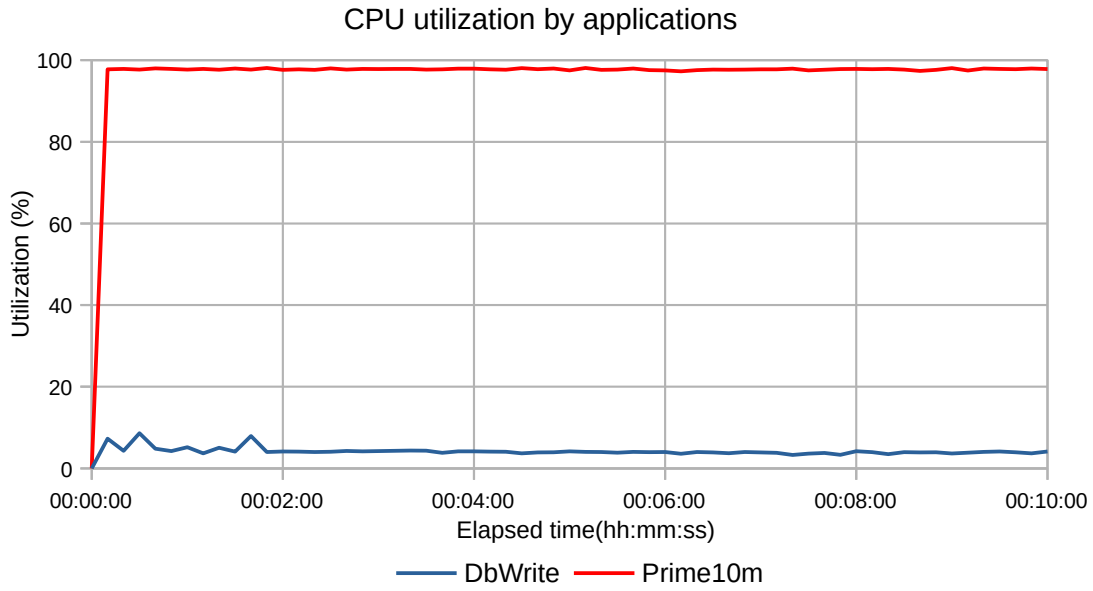

Figure 5.3: CPU utilization of DbRead and Prime10m

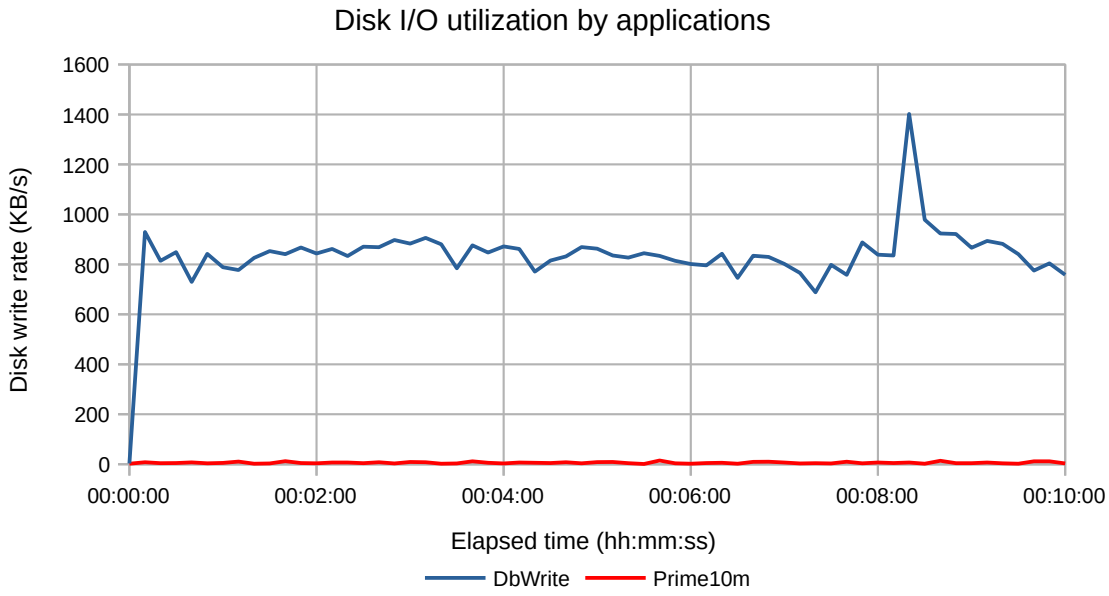

Figure 5.4: Disk write rate of DbWrite and Prime10m

Similar results were observed in the other applications as well and these observations showed that a small worker thread pool size provided better latency for CPU dominant applications tested while a larger thread pool size provided better latency for the I/O dominant applications that were tested.

The important fact that was observed during this stage of experiments was that the performance provided in terms of throughput and latency by a given thread pool size depended on the application and incoming workload. This verified the initial hypothesis that finding a fixed value for the worker thread pool size is a cumbersome task. It also proved that even if a value for the worker thread pool size was specified, it would need to be changed based on incoming workloads in order to improve performance.

5.1.2 Dynamic thread pool size with fixed workload simulations

As explained in Chapter 1, the main objective of this dissertation is to propose a solution to the cumbersome task of manually finding and configuring the worker thread pool size. The proposed solution consists of three algorithms which are able to find and configure the worker thread pool size by optimizing throughput, mean latency or 99th percentile of latency.

This section of the dissertation presents evaluations of the three proposed algorithms for different application. The word dynamic of the title of this sections reflects the fact that the worker thread pool size was allowed to be changed by the proposed algorithms dynamically during these experiments. The incoming workload to the server remained fixed during these experiments.

In order to evaluate the performance of the proposed algorithms in dynamically adjusting the worker thread pool size, the results obtained by using fixed worker thread pool sizes are also presented along them.

Mean latency optimization

This section presents comparisons of performance of the proposed mean latency optimization algorithm.

Figure 5.5 shows the mean latency values recorded for the Prime10m test for the fixed workload scenario. The fixed workload used was 100 concurrent users.

Fixed1, Fixed10 and Fixed100 in the figure 5.5 depicts the mean latency values over time for experiments with fixed number of worker threads. The numerical value given

as the suffix in them refer to the worker thread pool size used. Empirical data shows that the mean latency has improved when the number of worker threads was increased. ‘Adaptive’ in this graph depicts the mean latency values over time obtained for the experiment conducted by using the mean latency optimization algorithm starting with a worker thread pool size of one. As the test progressed, the algorithm adjusted the number of worker threads and have successfully minimized the mean latency value.

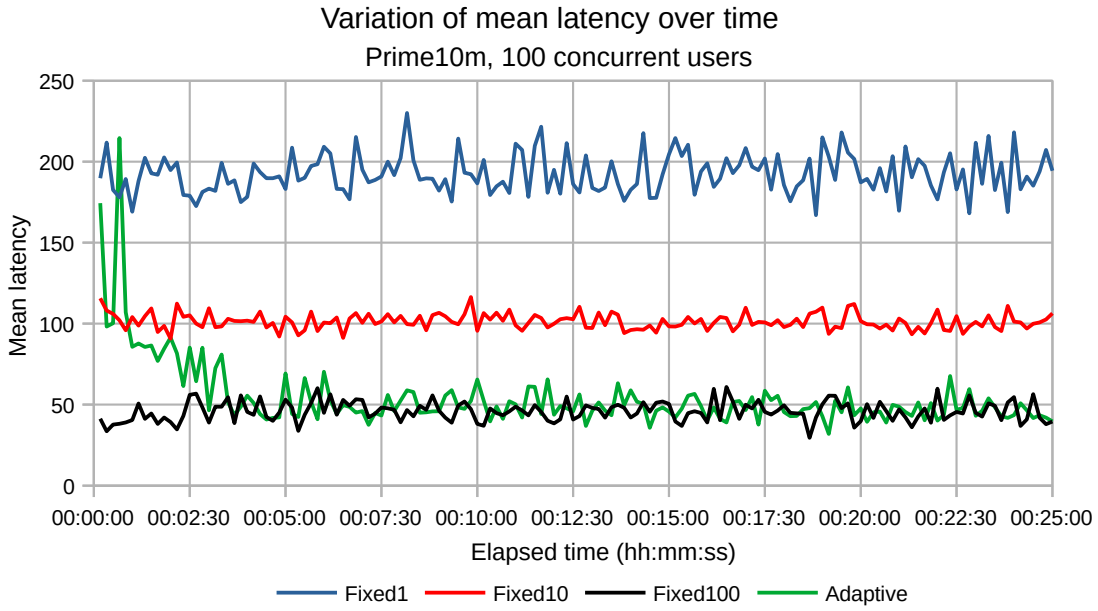

Figure 5.5: Mean latency comparison in Prime10m

The variation of thread pool size by the mean latency optimization algorithm over time for this experiment is shown in 5.6. This graph shows how the system has increased the thread pool size from 1 in order to minimize the mean latency.

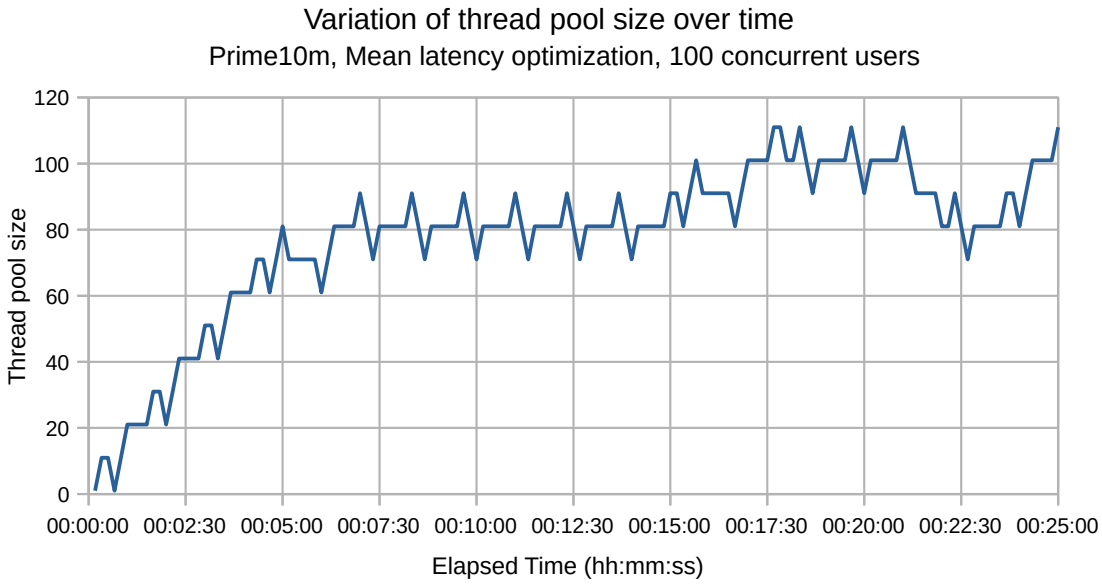

Figure 5.6: Thread pool size variation in Prime1m

The period from 00:06:20 to 00:14:50 shows that the system has converged at 81 threads with periodic increments and decrements. These periodic increments and decrements are expected as they are deliberate increments and decrements caused by the algorithm. This can be explained further by considering a part of the above mentioned time period. Table 5.1 shows how the thread pool size has varied from 00:06:40 to 00:09:00. The thread pool size is at 81 from 00:06:40. At 00:07:00, the thread pool size is increased to 91 by the INC_ITR iteration of the algorithm which increases the thread pool size by 10. Next, at 00:07:10, the INC_CHECK_ITR iteration of the algorithm checks whether the thread pool size increment has made an improvement. Since improvement obtained by this thread pool size increment was a decrement of 2.02% of mean latency, which is less than the minimum required 5% decrement of mean latency. As this has not resulted in an improvement, the thread pool size increment is undone by decreasing the thread pool size by 10 to back to 81. At 00:07:20 the thread pool size is decremented by 10 threads to 71 by the DEC_ITR iteration of the algorithm. Then at 00:07:30 the DEC_CHECK_ITR iteration checks the improvement of mean latency. Since the change in mean latency obtained by this thread pool size decrement was actually an increase of 19.45% of mean latency, the thread pool size decrement was undone by increasing the thread pool size back to 81.

Table 5.1: Increments and decrements of thread pool size

Elapsed time (hh:mm:ss)	Worker thread pool size	Mean latency (ms)
00:06:40	81	48.62
00:06:50	81	45.10
00:07:00	91	46.01
00:07:10	81	37.54
00:07:20	71	44.84
00:07:30	81	43.27
00:07:40	81	55.97
00:07:50	81	45.98
00:08:00	81	52.48
00:08:10	81	58.81
00:08:20	91	57.81
00:08:30	81	45.02
00:08:40	71	45.26
00:08:50	81	45.79
00:09:00	81	45.88

Since the increment of the thread pool size did not provide an improvement, the algorithm locks improvements for the next minute. This is the reason why a thread pool size increment was not observed at 00:07:40 although it was the INC_ITR iteration of the algorithm. Similarly, the thread pool size decrement was not observed at 00:08:00 in the DEC_ITR of the algorithm because the thread pool size decrement in the previous DEC_ITR iteration did not provide the required improvement of mean latency. However, increments and decrements have resumed at 00:08:20 and 00:08:40 respectively due to the expiration of the locks on increments and decrements.

Similar improvements of mean latency was also observed in other applications as well. Figure 5.7 depicts the comparison of mean latency values that was observed in the DbWrite application.

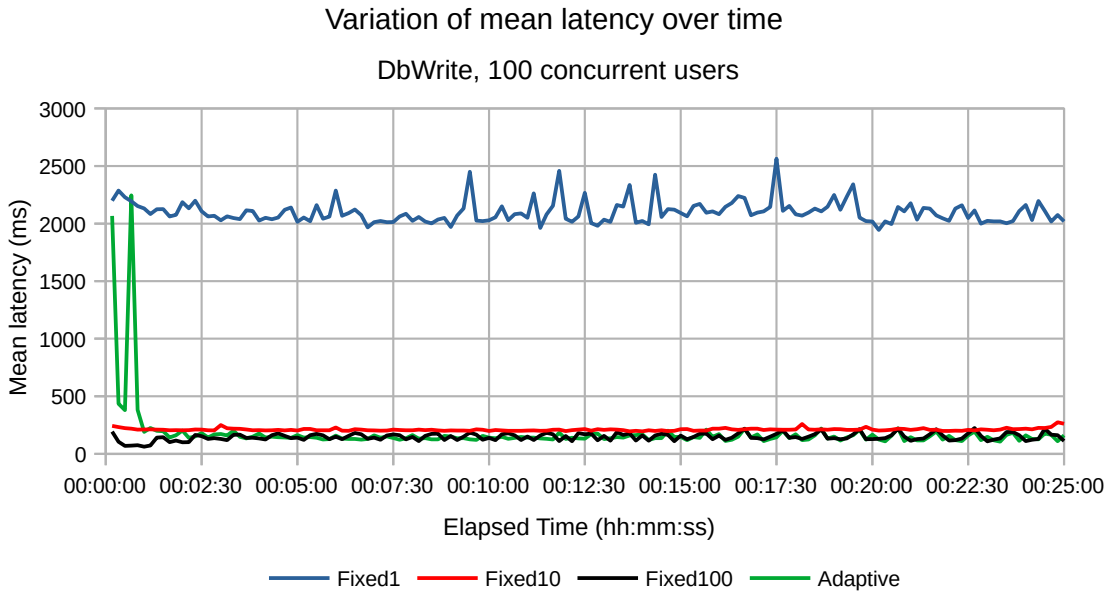

Figure 5.7: Performance of the mean latency optimization algorithm in DbWrite

The DbWrite application showed poor performance in terms of latency when a fixed worker thread pool size of 1 was used. When the mean latency optimization algorithm was used starting with a worker thread pool size of 1, the system detected the high mean latency at the beginning and reduced it by increasing the thread pool size.

By considering the mean latency range between 0ms and 300ms of the above graph, it can be observed that the mean latency values obtained by using a fixed worker thread pool size of 100 and by using the adaptive concurrency control mechanism showed similar results. Figure 5.8 shows the section of the y axis between 0ms and 300ms of figure 5.7 in order to observe this clearly.

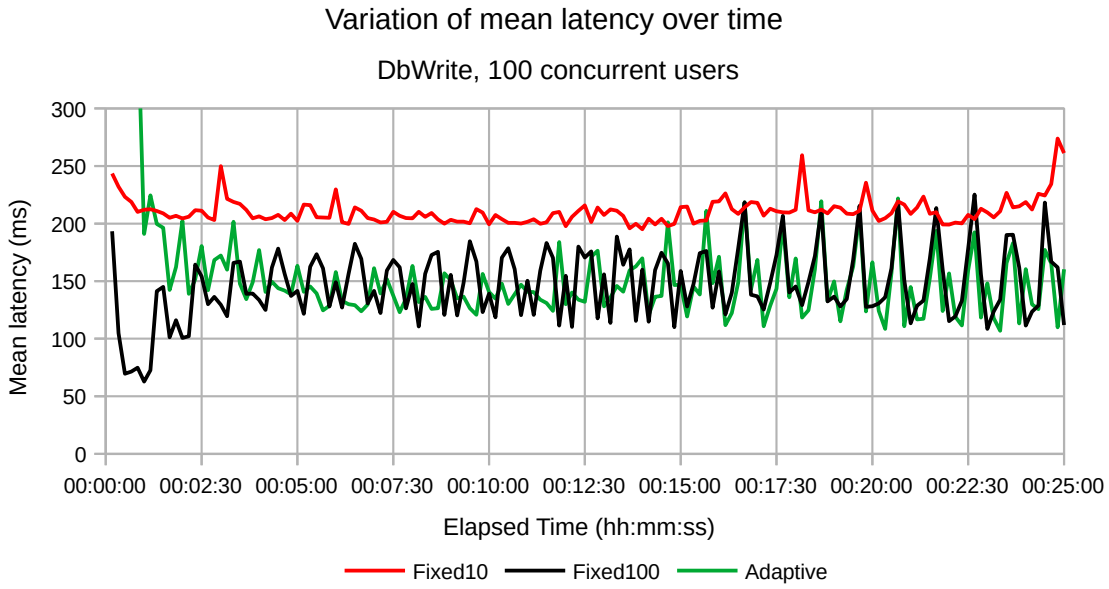

Figure 5.8: Comparison of mean latency in DbWrite over selected latency range

Figure 5.9 shows the variation of thread pool size against time of the adaptive concurrency control mechanism for the above test. This shows a very important observation that was made during this experiment. The proposed adaptive concurrency control mechanism was able to obtain mean latency values similar to the experiment with a fixed worker thread pool size of 100 by utilizing less than or equal to 100 threads. Practically, it is possible for a system administrator to configure a thread pool size such as 100 instead of 10 and be satisfied with it because it provided improved performance. However the proposed algorithm has detected online that mean latency cannot be improved by increasing the thread pool size and has therefore not increased it up to 100 as depicted in this graph.

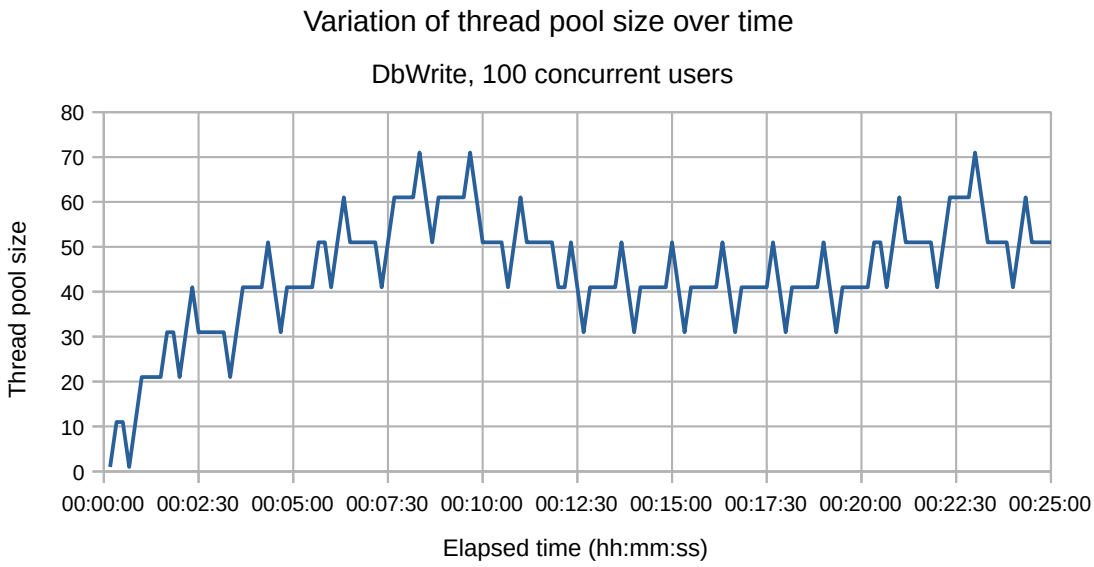

Figure 5.9: Thread pool size variation in DbWrite using mean latency optimization

Furthermore, another fact that can be observed by examining the figure 5.8 and figure 5.9 is that the increase in thread pool size after the test started has indeed shown a decrement in the mean latency of the workload. This shows that the proposed algorithm has been able to successfully decrease the mean latency by increasing the worker thread pool size as required.

Figure 5.10 shows the results obtained for the Prime1m application with 100 concurrent users. It can be observed that the proposed adaptive concurrency control mechanism using the mean latency optimization algorithm has successfully minimized the mean latency during this experiment. Figure 5.11 provides a closer look at the curved obtained for the fixed thread pool size of 50 and for the adaptive algorithm. By examining the information about the thread pool size along with the obtained mean latency values, it can be observed that the proposed adaptive concurrency control mechanism has obtained better mean latency values than when using a with a fixed worker thread pool size of 50. Practically, a system administrator may experiment with worker thread pool sizes of 1, 2, 4 and 50 and decide to use 50 as it provides the best performance out of the ones tested. He/She may or may not experiment with thread pool sizes between 4 and 50 because there is no common guideline to say which thread pool sizes to check.

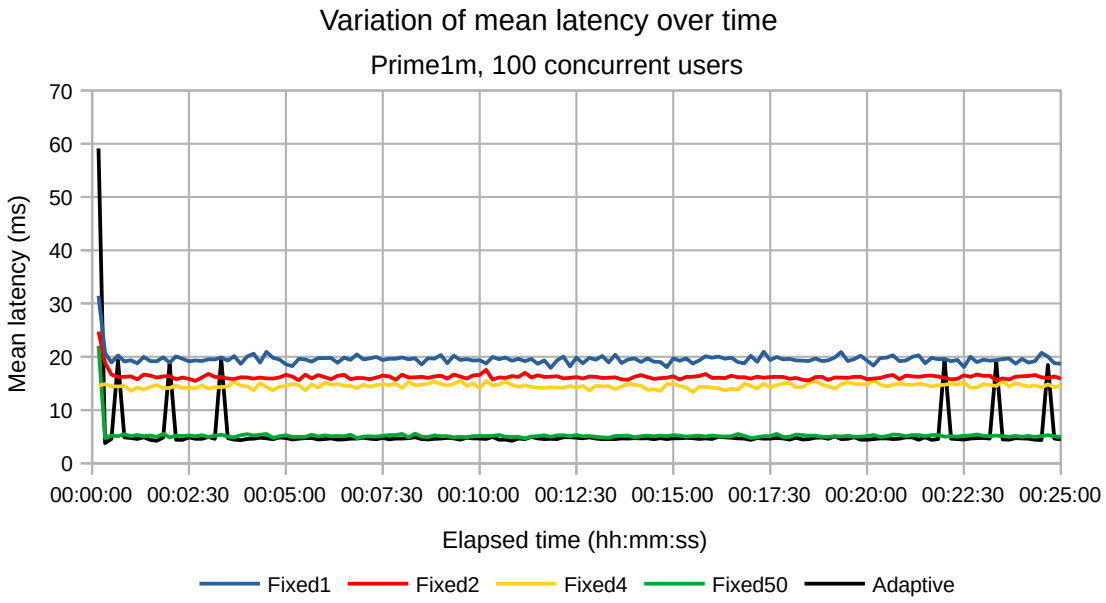

Figure 5.10: Comparison of mean latency in Prime1m

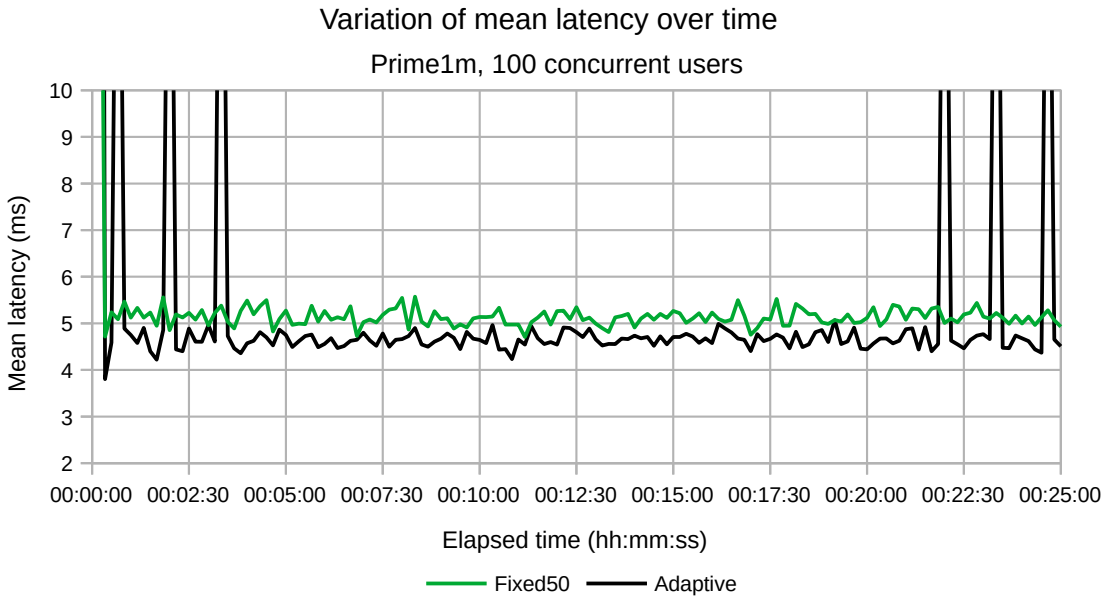

Figure 5.11: Comparison of mean latency in Prime1m over a selected range

It should also be noted that the mean latency optimization algorithm has obtained better performance while using a worker thread pool size less than 50 thereby proving that using a higher thread pool size does not always result in better mean latency. In that case, the proposed method has shown to have an advantage over using fixed

thread pools. Figure 5.12 shows the corresponding thread pool size over time graph for the adaptive algorithm.

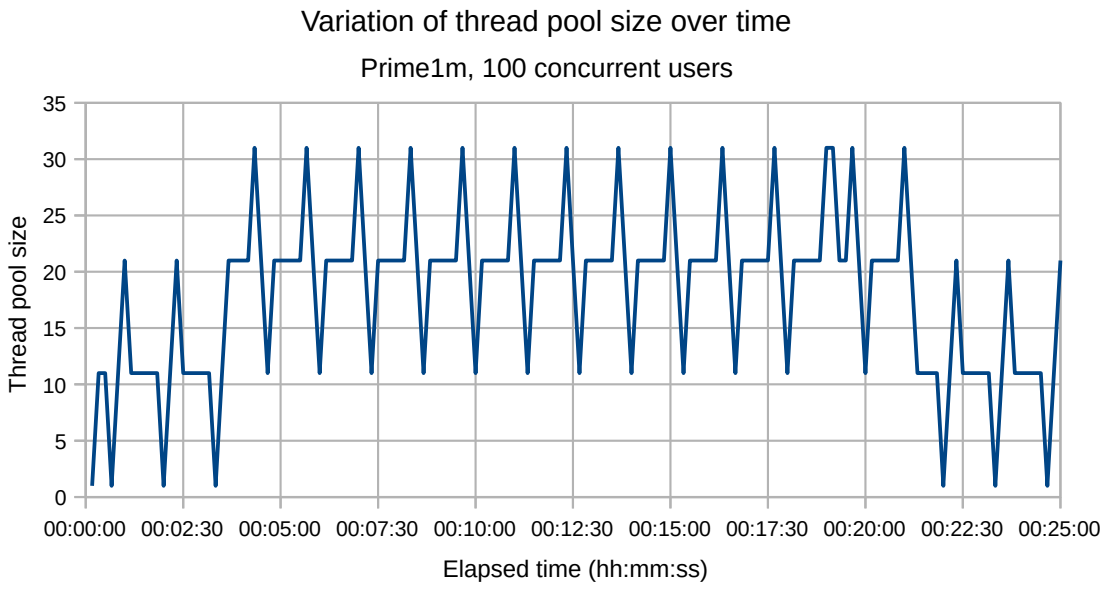

Figure 5.12: Thread pool variation in Prime1m for mean latency optimization

Throughput Optimization

This section presents the results obtained for the throughput optimization algorithm for different applications when tested with a fixed incoming workload.

Figure 5.13 depicts a comparison of the variation of throughput over time for several fixed worker thread pool sizes and the proposed adaptive concurrency control mechanism using the throughput optimization algorithm for the Prime1m workload. Figure 5.14 presents a closer look at the variation of throughput for fixed thread pools of size 10, 50, 100 and for the adaptive mechanism in order to understand the results in a much clearer manner. It shows that a worker thread pools of size 10 has provided slightly better throughput over using a worker thread pool of size 50 or 100. Furthermore it can be seen that the adaptive mechanism has obtained a similar throughput to that of the fixed thread pool of size 10.

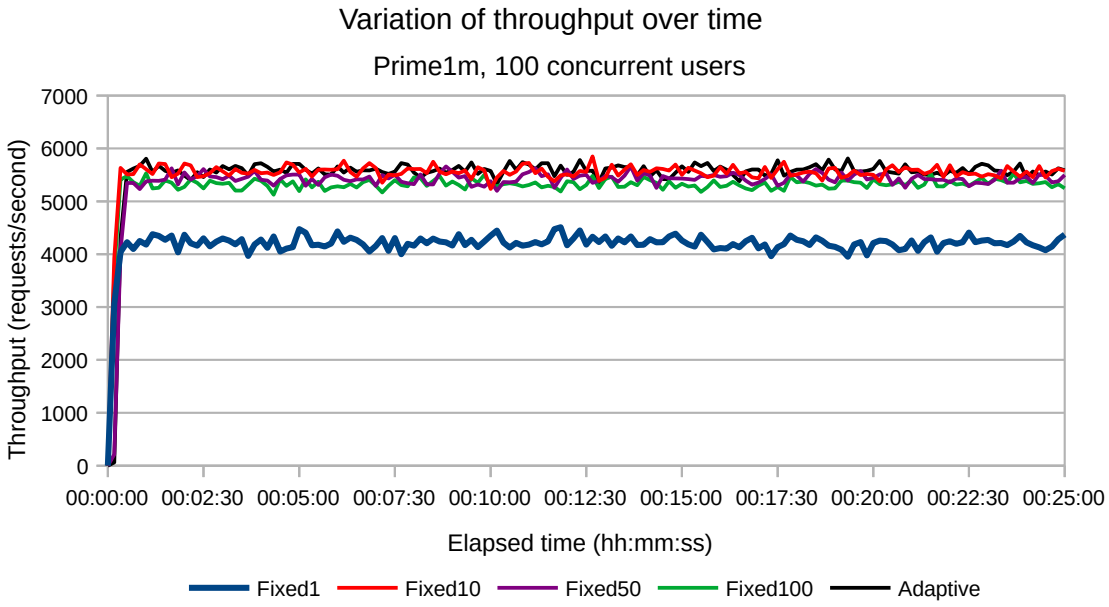

Figure 5.13: Performance of the throughput optimization algorithm in Prime1m

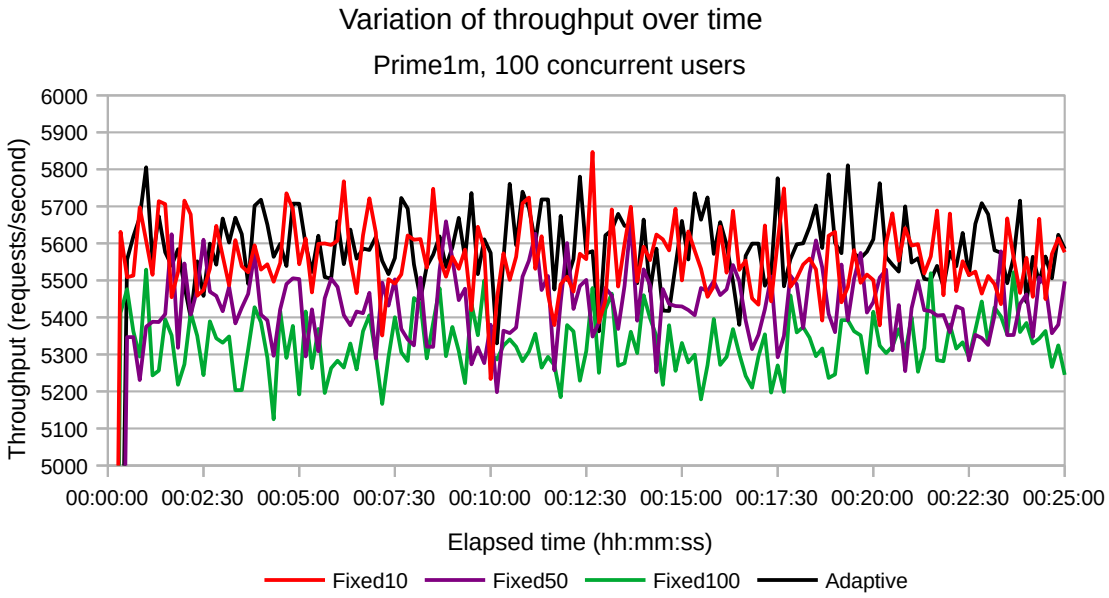

Figure 5.14: Throughput comparison in Prime10m over selected throughput range

Examining the variation of worker thread pool size provided by the adaptive concurrency control mechanism depicted in figure 5.15 shows that the worker thread pool size has converged at a thread pool size of 11 to obtain this throughput.

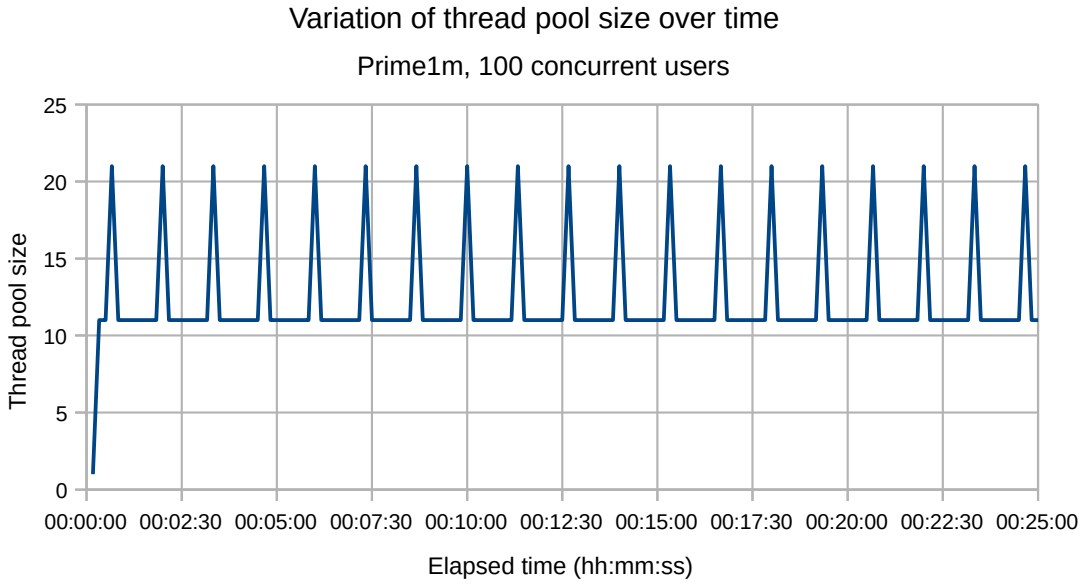

Figure 5.15: Thread pool size variation in Prime1m using throughput optimization

This shows that the proposed adaptive concurrency control mechanism with throughput optimization has converged at a thread pool size of 11 and has not increased further because further increments did not provide better throughput. The momentary increase in thread pool size shown above has occurred periodically every minute. This was because when the throughput optimization algorithm increased the thread pool size from 11 to 21, an increment of throughput of at least 10% was not observed and this resulted in it waiting for one minute before checking again if an increment of worker thread pool size improved throughput. By examining the performance of fixed thread pool sizes, it can be observed that the fixed thread pool of size 10 provided very similar throughput. The adaptive algorithm converging at a thread pool size of 11 shows that it has properly identified the thread pool size to improve throughput and has not increased unnecessarily.

Figure 5.16 depicts a comparison of the variation of throughput over time obtained for fixed worker thread pool sizes and the proposed adaptive concurrency control mechanism with throughput optimization algorithm. This presents results for the DbRead application. Fluctuations can be observed in throughput in the DbRead workload. However, they were not observed only when using the proposed concurrency control mechanism, but for fixed thread pool sizes as well. (Java's

Garbage Collection would be one of the reasons for these fluctuations in throughput). It is evident from this comparison that the proposed method has obtained similar throughput as that of the fixed thread pool size of 10.

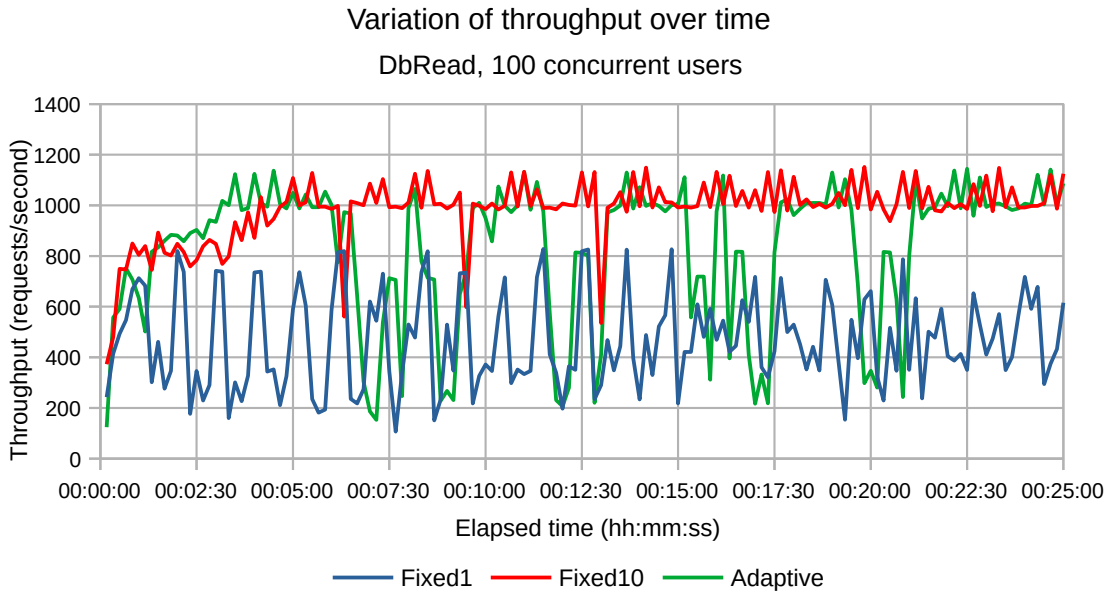

Figure 5.16: Performance of the throughput optimization algorithm in DbRead

Figure 5.17 depicts the corresponding graph depicting the variation of thread pool size over time. It can be observed that the algorithm has kept the thread pool size at 11 most of the time during the experiment. However, fluctuations in measured throughput had caused few changes in the thread pool size. But it can also be observed that the algorithm managed to successfully recover from the thread pool size changes due to fluctuations.

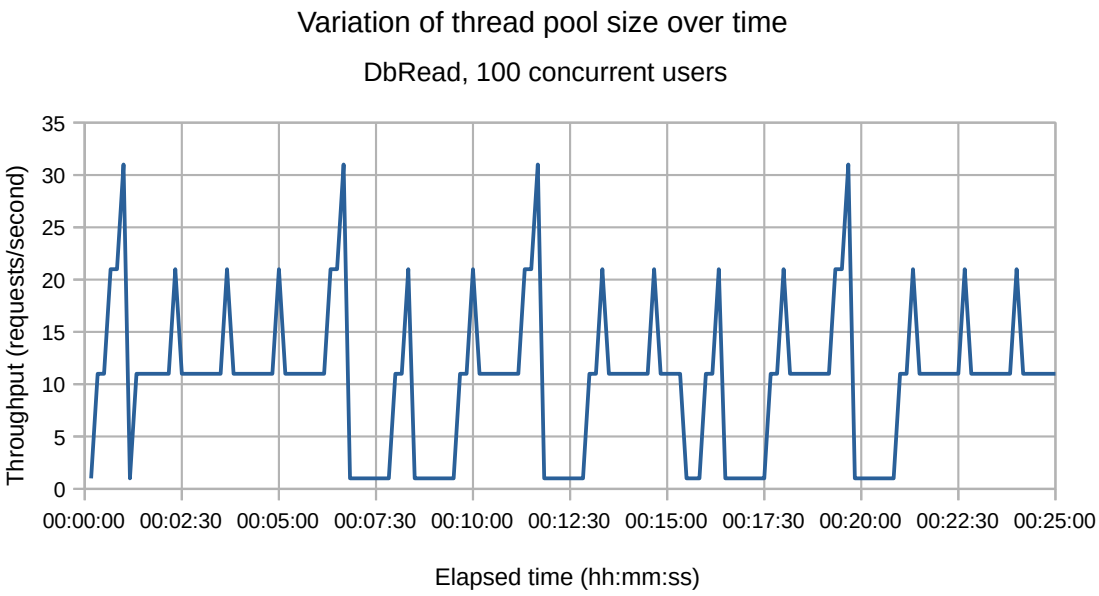

Figure 5.17: Variation of thread pool size in DbRead using throughput optimization

99th percentile of latency optimization

Results obtained by testing the proposed 99th percentile of latency optimization algorithm is presented in this section.

Figure 5.18 shows a comparison of the 99th percentile of latency values recorded when using fixed worker thread pool sizes and the proposed adaptive concurrency control mechanism using the 99th percentile of latency optimization algorithm for the Prime10m workload. 99th percentile of latency values recorded for fixed thread pool sizes of 1 and 10 were much less than those obtained for fixed thread pool sizes of 50 and 100 showing that less number of threads provided better latency for this application. It can also be observed that the 99th percentile of latency values recorded for fixed thread pool sizes of 1 and 10 show overlaps at times. This explains why the proposed adaptive concurrency control mechanism has not obtained clear convergence of thread pool size as depicted in figure 5.19. Furthermore, the momentary increments of latency observed in the adaptive mechanism was due to the algorithm checking if a higher thread pool size provided better latency as explained in the algorithm presented in Chapter 4 of this dissertation.

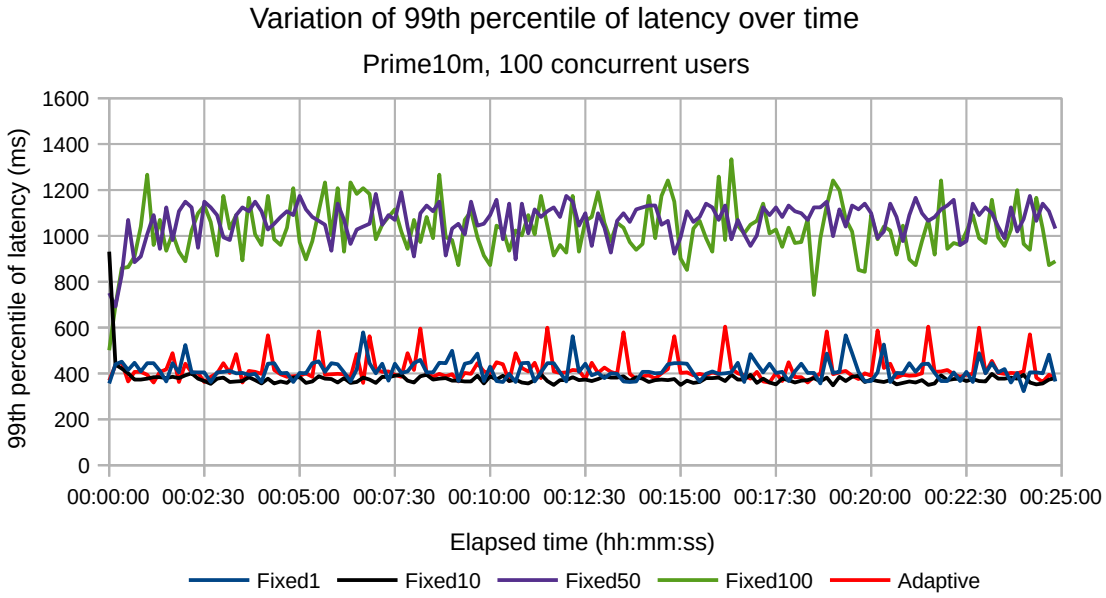

Figure 5.18: Performance of the throughput optimization algorithm for Prime10m

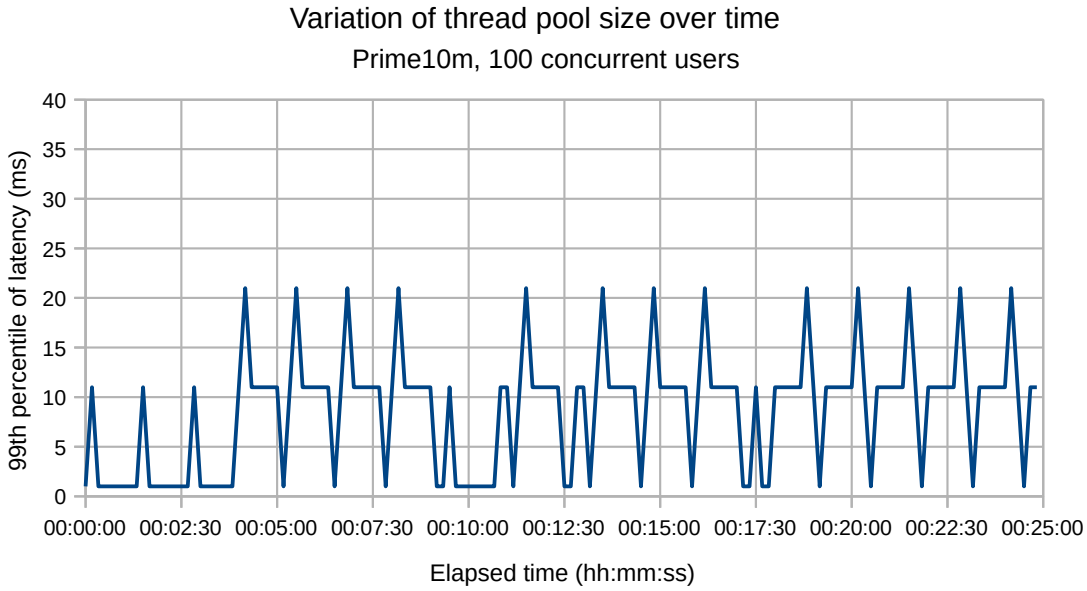

Figure 5.19: Thread pool size variation in Prime10m using throughput optimization

5.1.3 Dynamic thread pool size with varying load simulations

As explained in Chapter 3, this set of experiments were conducted to evaluate the behavior of the proposed adaptive concurrency control mechanism against dynamic

incoming workloads. The variation of the number of concurrent users which was used to simulate the varying load on the server has been explained in Chapter 3 of this dissertation.

All three algorithms were again tested with different applications during this stage of experiments.

Throughput optimization

The following section presents the results obtained for the throughput optimization algorithm.

Figure 5.20 shows the variation of throughput against time for the Prime1m application. As shown by the graph, the throughput optimization algorithm shows positive results in adjusting the thread pool size to maximize throughput even when the load is dynamic. It can be observed from the graph that the throughput from 00:00:00 to 00:05:00, 00:10:00 to 00:15:00 and 00:20:00 to 00:25:00 remained the same. This is not because the algorithm has failed to increase the thread pool size to obtain better throughput but due to the processing limits of the server hardware in processing the requests associated with the Prime1m application. This fact becomes clearer by examining figure 5.21 which shows the corresponding variation of thread pool size against time for this experiment. As can be seen in this graph, there are spikes that are caused by the adaptive concurrency control algorithm increasing the thread pool size and checking if the increment caused an improvement. However, because it did not do so, the system reverts back to the previous thread pool size. Therefore the throughput has remained the same in this experiment due to the physical limits of the processing hardware.

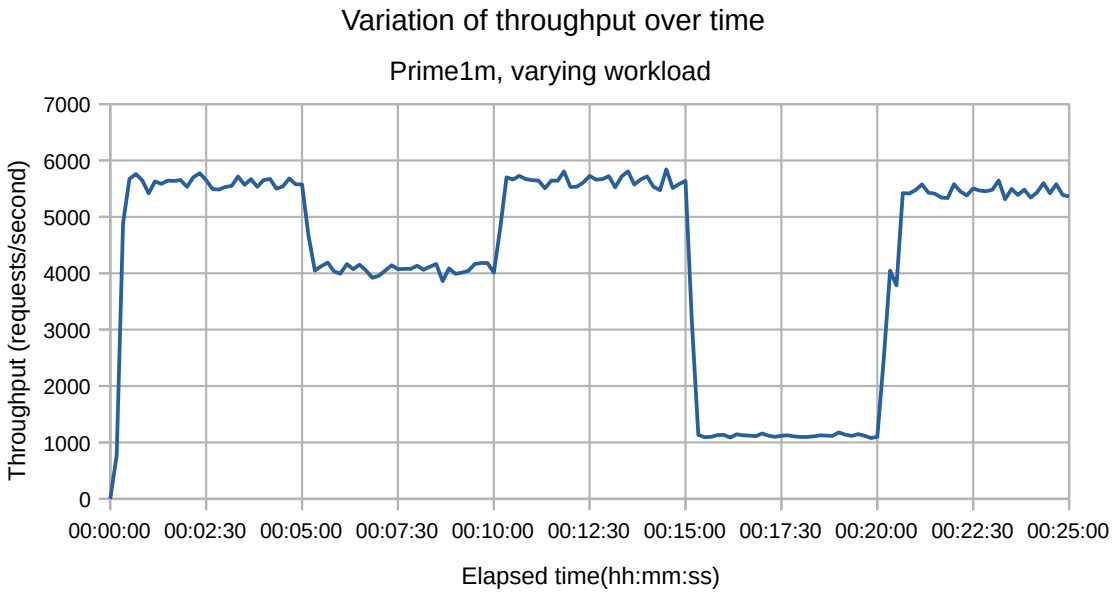

Figure 5.20: Variation of throughput in Prime1m with varying workload

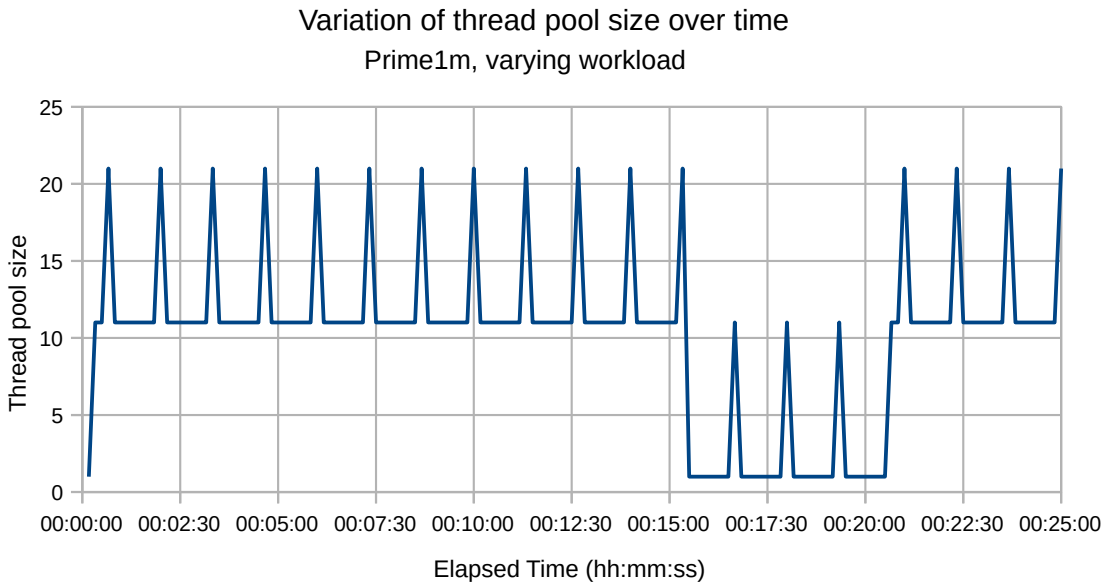

Figure 5.21: Variation of thread pool size in Prime1m with a varying workload using the throughput optimization algorithm

Another observation that can be made by looking at the above graph is that during the period from 00:05:00 to 00:10:00, the size of the thread pool has remained at 11 similar to 00:00:00 to 00:05:00, 00:10:00 to 00:15:00 and 00:20:00 to 00:25:00. When the number of concurrent users decrease to one at 00:05:00, the algorithm detects it and checks which thread pool size to use. As the throughput optimization algorithm uses a step size of 10 when adjusting the thread pool size, the thread pool sizes it can use are 1, 11, 21 etc. Since the test was initialized, the algorithm recorded the throughput that a thread pool size of 1 and 11 provides. Since the throughput provided by a thread pool size of 1 significantly low, the algorithm has decided the keep the thread pool size at 11. The method of deciding the thread pool size to decrease to in such a situation has been explained in Chapter 4 of this dissertation.

A similar situation was observed with the DbRead application as well and is depicted in figure 5.22. The corresponding thread pool size over time graph for this experimented is presented in figure 5.23.

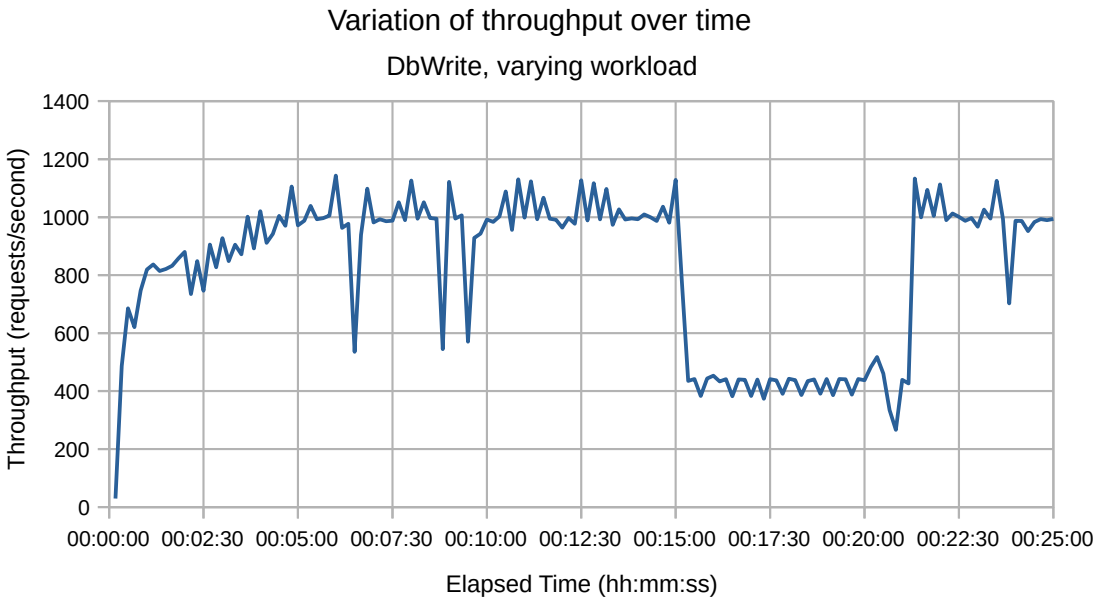

Figure 5.22: Throughput variation in DbWrite with a varying workload using throughput optimization

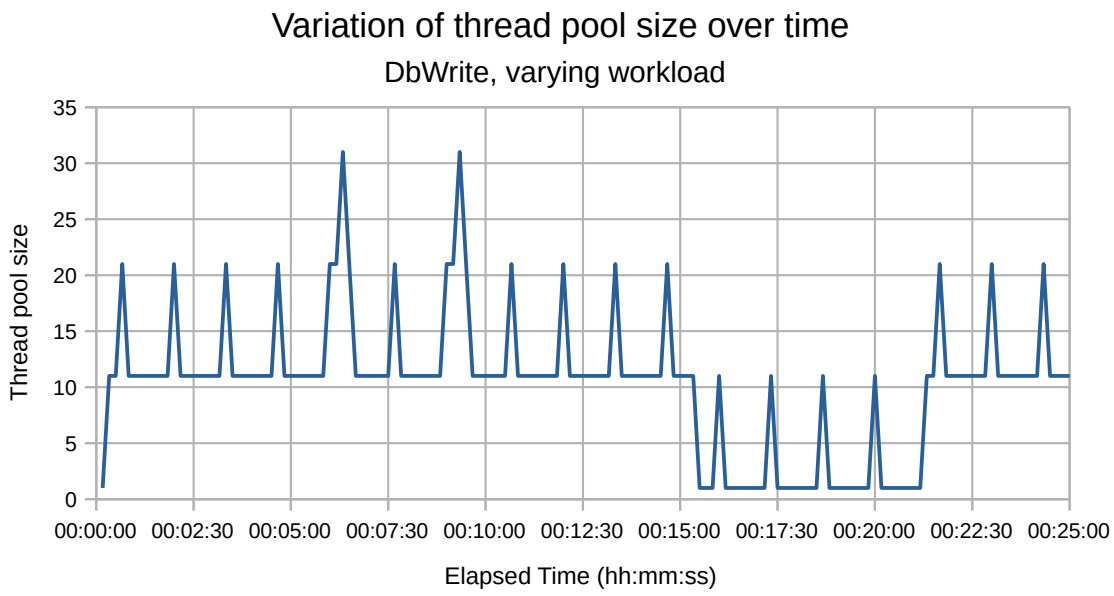

Figure 5.23: Variation of thread pool size in DbWrite using a varying workload

Mean Latency optimization

Results of the experiments conducted to evaluate the performance of the mean latency optimization algorithm for varying workloads is presented in this section.

Figure 5.24 shows the variation of mean latency over time for the Prime10m application when tested with the varying workload. It can be observed that the mean latency when the test started was more than 200ms and that towards the beginning of the test it has a high value.

However, it can be observed that the mean latency decreased as the test progressed. This was caused by the optimization algorithm adjusting the thread pool size to minimize the mean latency. Figure 5.25 presents the corresponding graph depicting the variation of thread pool size over time containing this information. The periodic spikes visible in this graph were deliberately caused by the algorithm making periodic changes and checking if an improvement was obtained from the change. However, when it detected that no improvement were made, the increments and decrements were locked for a period of 1 minute which is reflected by the momentary pause in these spikes.

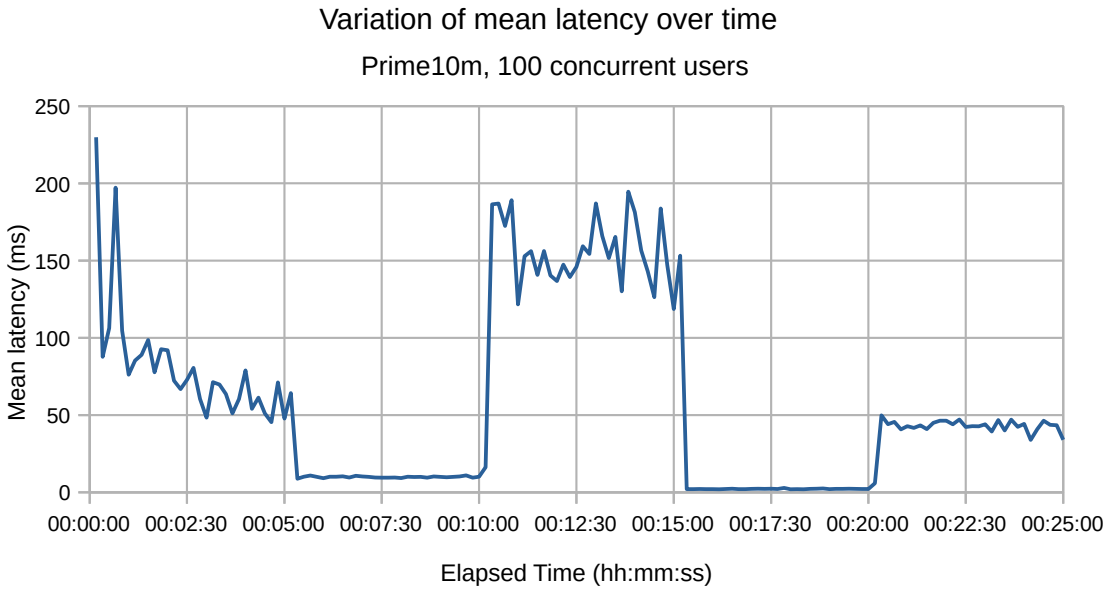

Figure 5.24: Variation of mean latency in Prime10m using a varying workload

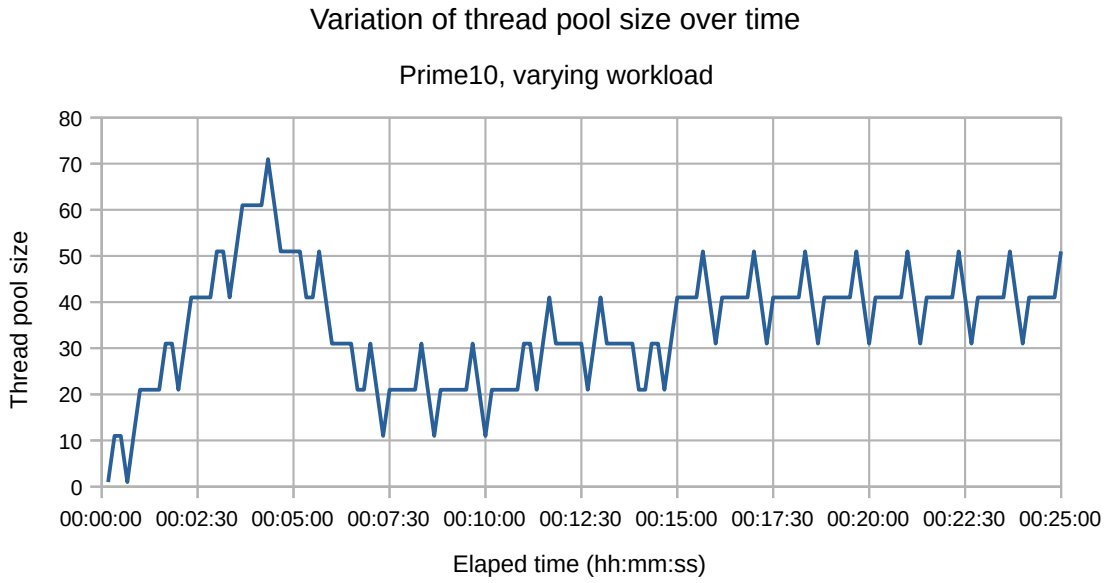

Figure 5.25: Thread pool size variation in Prime10m

99th Percentile of latency optimization

Following section presents the results obtained when experiments were conducted using the adaptive concurrency control mechanism with the 99th percentile of latency optimization algorithm. These experiments also used the varying workload presented in Chapter 3.

Figure 5.26 shows the variation of 99th percentile of latency over time for the Prime1m application. Figure 5.27 presents the corresponding variation of the thread pool size over time graph caused by the algorithm. It can be observed that the algorithm has converged at thread pool sizes of 11 and 31 at different intervals. After converging, the algorithm has periodically checked whether increments or decrements of thread pool size provides an improvement in 99th percentile of latency. This can be seen by the periodic increments and decrements occurring every minute in the graph.

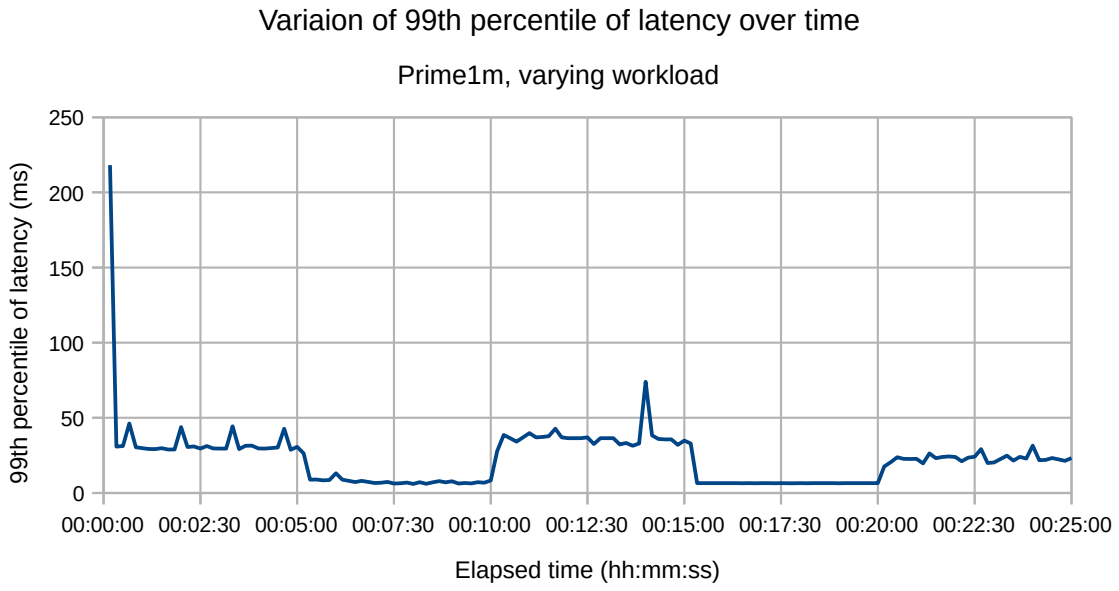

Figure 5.26: 99th percentile of latency variation in Prime1m

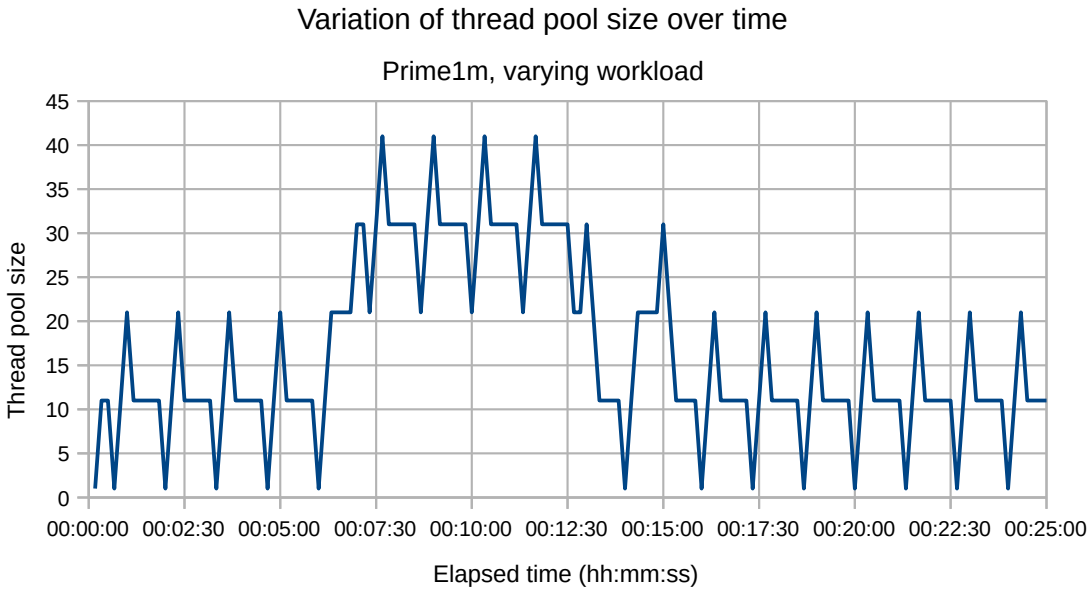

Figure 5.27: Prime1m thread pool size variation

Figure 5.28 shows the variation of 99th percentile of latency over time for the DbWrite application. In this experiment with a varying load, some fluctuations of latency was observed.

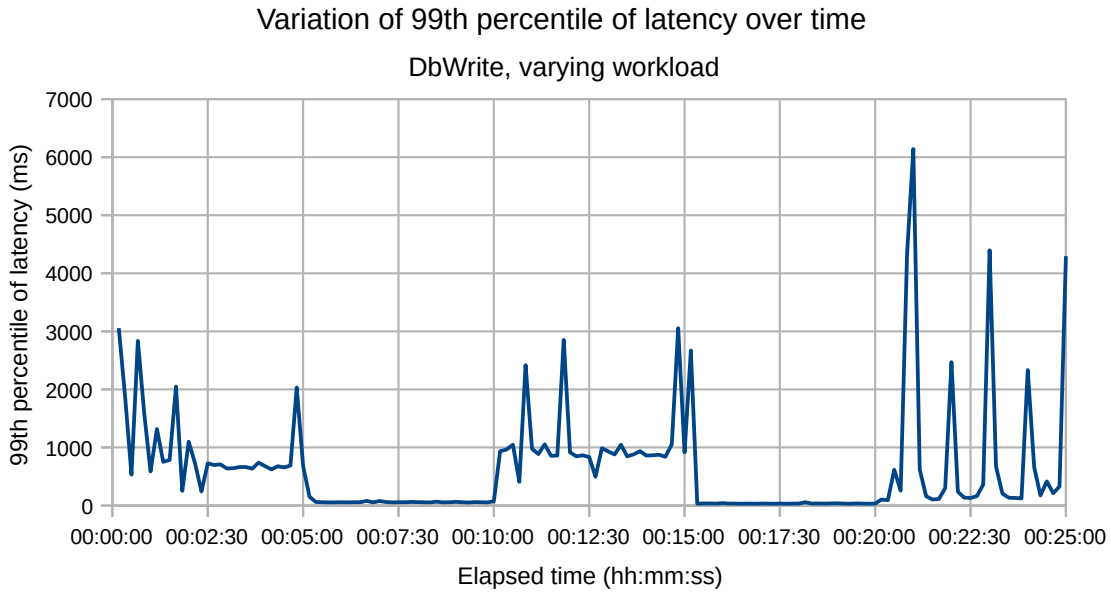

Figure 5.28: 99th percentile of latency variation in DbWrite

These fluctuations have made it difficult for the proposed algorithm to maintain smooth thread pool sizes as can be seen in the figure 5.29. The proposed algorithm only accepts improvements due thread pool size increment at the 2nd iteration and improvements due to decrements at the 4th iteration respectively as explained in Chapter 4. This has made it possible for the algorithm to not cause the thread pool size to increase without control and to try and correct fluctuations.

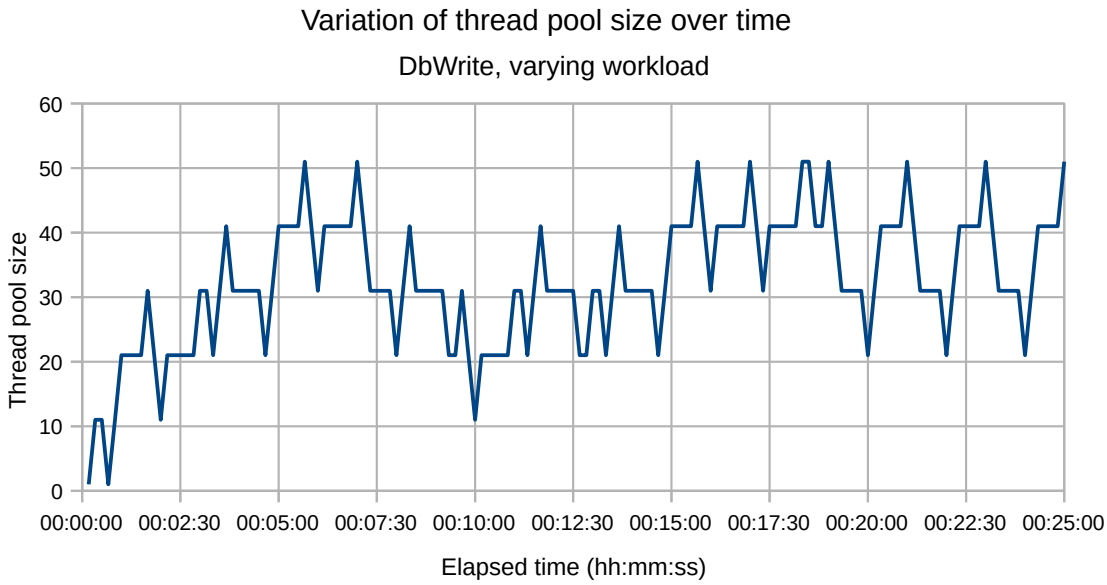

Figure 5.29: DbWrite thread pool size variation for 99th percentile optimization

5.2 Analysis of the effects of parameters of the algorithms

The three proposed algorithms make use of several parameters to function correctly. During the course of this study, these parameters were fine tuned by studying their effects on the adaptive concurrency control mechanism. This section discusses the effects of these parameters.

5.2.1 Period between iterations of algorithms

As explained in Chapter 3, the proposed algorithms' outer loop is executed every 10 seconds. This period has a direct impact on the detection of change of performance metrics thereby affecting when thread pool size adjustments occur. A high value of this period causes the server to detect changes in performance metrics with a delay. This in turn causes a delay in thread pool size adjustments. Experiments showed that these would result in the server being too slow to respond to changes in performance metrics and workload patterns thereby reducing the ability of the server to optimize performance properly.

Figure 5.20 presented in the previous section showed the variation of throughput over time for the Prime1m application with a varying workload. This was the graph obtained as a result of using a period of 10 seconds in the outer loop of the throughput optimization algorithm.

The corresponding thread pool size over time graph of this experiment was shown in figure 5.21. The reason for the thread pool size remaining at 11 for the period from 00:00:00 to 00:15:00 has been explained in the previous section. At 00:15:00, the number of concurrent users decreased suddenly due to the nature of the varying workload test as explained in Chapter 3. The important observation here is that the system has converged at a thread pool size of one within a short period of time. Similarly, when the number of concurrent users increased at 00:20:00, the system increased the thread pool size to 11 and converged within such a similar short period. This is a positive result because it shows that the proposed algorithm responds and adapts to changes in incoming workloads within a short duration.

Figure 5.30 shows the variation of throughput over time obtained for the same experiment using a period of 1 minute between iterations of the outer loop of the algorithm. Empirical results show how the server has been slow compared to using a period of 10 seconds, to respond to changes in the varying workload. For example, in the experiment with a period of 10 seconds between iterations of the outer loop of the algorithm, the algorithm recorded a sudden decrease in throughput shortly after 00:15:00 which was what actually happened. However, with a period of 1 minute, the

server shows a gradual decrease in throughput after 00:15:00 and takes until almost 00:20:00 to record the lowest value of throughput.

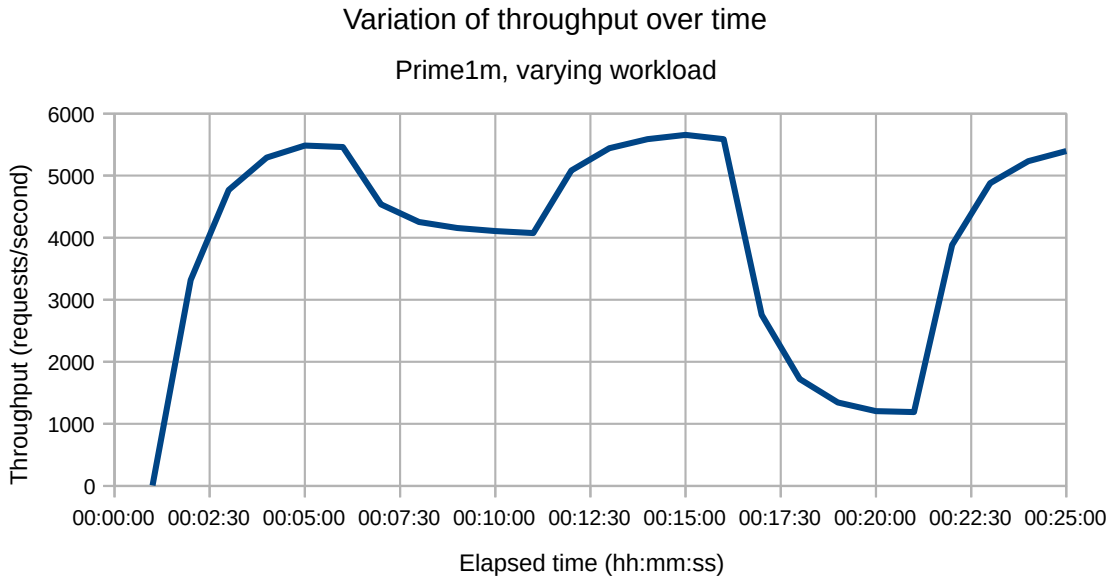

Figure 5.30: Effect of long periods on throughput measurement

This had a negative impact on when increments/decrements of thread pool sizes occurred as seen in the figure 5.31. For example, unlike the proposed algorithm using a period of 10 seconds, the modified algorithm using a period of 1 minute between iterations of the outer loop has not converged at a thread pool size of 1 until almost 00:20:00.

When a period of 5 seconds was used instead, the server detected changes in throughput within a short period of time similar to what was observed when the proposed period of 10 seconds was used. This can be observed figure 5.32.

However, upon examining the variation of thread pool size over time depicted in the figure 5.33, it can be observed that unnecessary thread pool size adjustments have occurred. Although similar throughput values were observed from 00:00:00 to 00:05:00 and 00:10:00 to 00:15:00, the system has unnecessarily increased the thread pool size after 00:00:00, to 21.

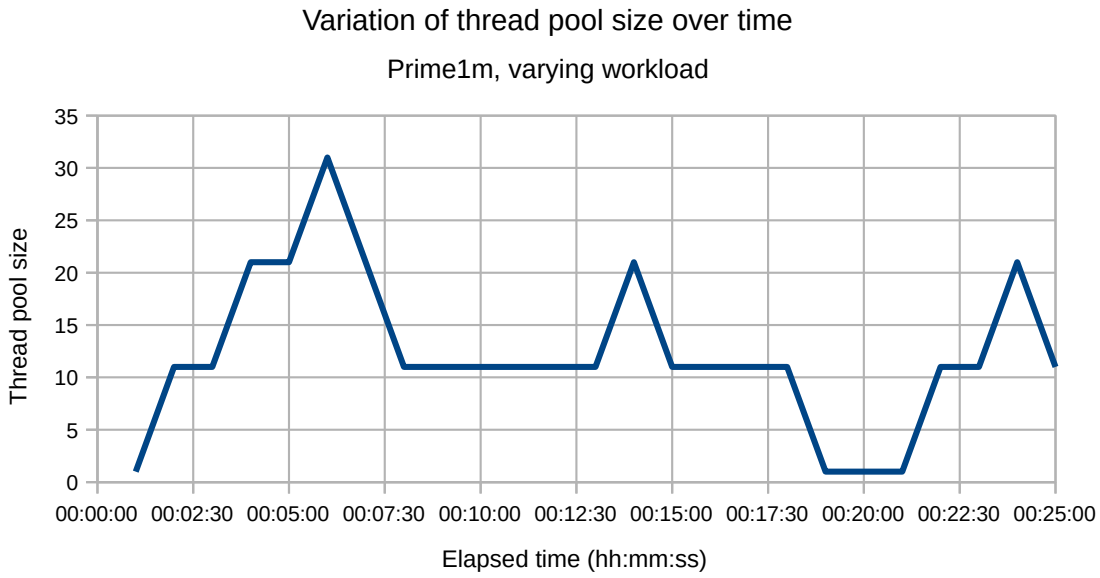

Figure 5.31: Effect on long periods on thread pool size adjustment

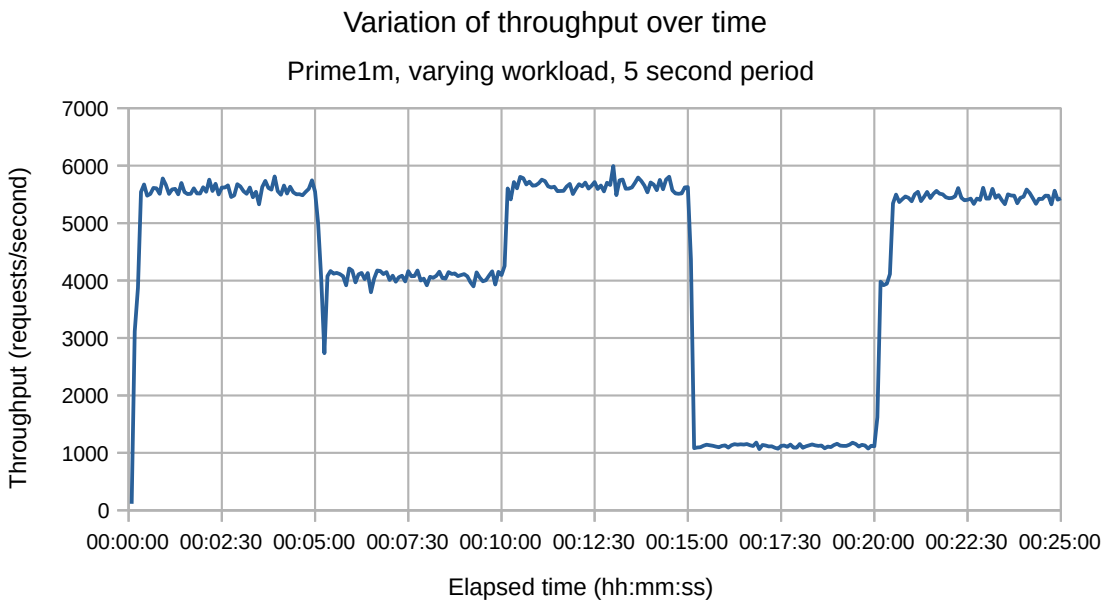

Figure 5.32: Effect of short period on throughout measurement

Furthermore, figure 5.33 also shows the effect of decreasing the thread pool size increment lock duration. This figure shows a lock duration of 30 seconds. A shorter lock duration such as this causes the system to make increments quickly even after they do not provide improved performance. This then causes the thread pool size to fluctuate unnecessarily.

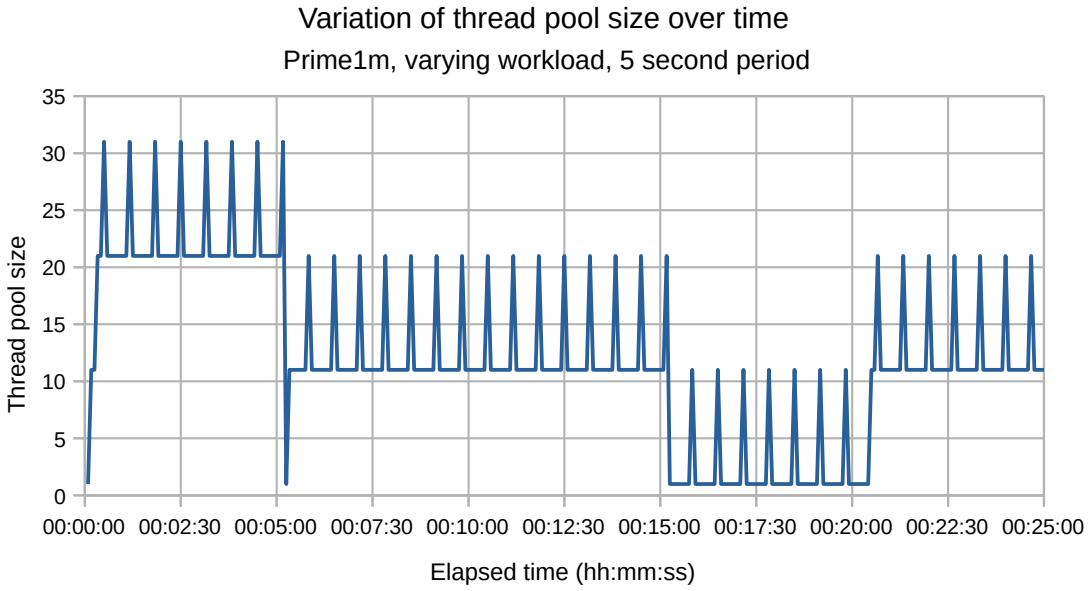

Figure 5.33: Effect of short period on thread pool size adjustment

5.2.2 Thread pool size increment/decrement step size

Another parameter that the algorithms used was the step size of thread pool size increments and decrements. Experimenting with this value showed that using a smaller step size causes delays in convergence of the thread pool size. It was also observed that small changes in thread pool size did not provide significant improvements in the observed performance metrics as well. In contrast, a larger step size of thread pool size increments and decrements caused the system to converge at thread pool sizes unnecessarily large or small respectively.

5.2.3 Minimum acceptable change in performance metrics

The proposed algorithms use 10% as the minimum acceptable change in throughput to cause a change of thread pool size. 5% is used as the minimum acceptable change in latency (mean and 99th percentile) to cause a change of thread pool size.

Fine tuning this parameters showed that lower values for minimum acceptable change in the measured performance metrics caused the system to unnecessarily respond to

minor changes in performance metrics and also to outliers. In contrast, using higher values as the minimum acceptable change caused the system to not be responsive enough to detect changes in the performance metrics measured.

Chapter 6 - Conclusions

6.1 Introduction

The final chapter of the dissertation presents the conclusions about the research. It presents a concluding summary on each research question explaining the answers that were obtained for them during the course of this study. Furthermore, this section also presents implications for future research.

6.2 Conclusions about research questions

6.2.1 Question 1

How does the thread pool size impact the performance of different applications that receive requests under different arrival patterns?

Experiments provided evidence that for a given number of threads, the performance that can be obtained would differ across applications and workloads.

Empirical evidence gathered during testing showed that I/O dominant workloads such as database write operations performed well with larger sized thread pools while CPU dominant workloads performed better when smaller thread pools were used.

Therefore the initial hypothesis that fixed thread pool sizes are inefficient has been proven through the empirical evidence that was gathered during experiments that were conducted. It is can also be concluded that the worker thread pool size has a significant impact on performance.

6.2.2 Question 2

How can an algorithm/s be developed that can dynamically change the thread pool size depending on the changing behavior of application and incoming workload?

Latency and throughput were identified as performance metrics. Therefore three algorithms were developed to adjust the worker thread pool size. These three algorithms developed are a throughput optimization algorithm, a mean latency optimization algorithm and a 99^{th} percentile of latency optimization algorithm. They adjust the worker thread pool size by periodically measuring throughput, mean latency and 99_{th} percentile of latency respectively.

To evaluate these algorithms, two sets of experiments were conducted as follows.

1. Dynamic thread pool size with fixed workload simulations

   In order to test the proposed algorithms, experiments were conducted using the proposed algorithms by keeping the workload from the client side fixed. During these experiments the worker thread pool size started at one and thereafter the proposed algorithms were made to change it as necessary. Testing of the proposed algorithms provided positive results thus proving that it is indeed possible to let the proposed algorithms measure performance metrics and adjust the thread pool size automatically.

2. Dynamic thread pool size with varying workload simulations

   This stage of experiments tested the proposed algorithms by using a varying workload. Similar to the previous stage of experiments, the proposed algorithms were made to adjust the worker thread pool size as necessary. Experiments conducted to evaluate the proposed algorithms showed that they are able to cope up with varying workloads and adjust the thread pool size

accordingly. However, although the effects of fluctuations of performance metrics were minimized, they can have an impact on the ability of the algorithms to function accurately.

Therefore empirical results gathered over these experiments show that the proposed algorithms are able to successfully adjust the worker thread pool size based on performance metrics that are measured.

6.2.3 Question 3

How can the parameters of the adaptive thread pool adjustment algorithms be tuned in order to improve results?

The proposed algorithms utilize several parameters. In order to answer this question, these parameters were varied and their effects on the algorithms were studied.

The proposed algorithms loop continuously. Experiments conducted by varying the period between iterations showed that using a high value for this parameter caused the server to detect changes in performance metrics with a delay. Using a low value for this period caused the algorithm to change the thread pool sizes unnecessarily.

Experiments conducted to study the effect of changing the thread pool increment/decrement step size showed that large step sizes caused the system to converge at unnecessarily large or small thread pool sizes while small step sizes caused delays in convergence of the thread pool size. Furthermore small changes in thread pool size did not provide significant improvements of measured performance metrics.

The proposed algorithms use 10% as the minimum acceptable change in throughput to cause a change of thread pool size and 5% as the minimum acceptable change in latency to cause a change of thread pool size. Lower values caused the system to respond to minor changes in performance metrics unnecessarily and to outliers as well,

High values for this parameter caused the system to not be responsive enough to detect changes in performance metrics.

6.3 Conclusions about research problem

This research study was aimed at finding a solution to the problem of configuring the thread pool size to optimize performance. Experiments and background information showed that manually setting a thread pool size is both difficult and inefficient. It is difficult because it is not possible to accurately predict the thread pool size to be used. It was also found to be inefficient because even if a thread pool size to maximize a performance metric is found experimentally, this may need to be changed over time due to reasons such as change of server hardware, variation of incoming workloads etc. Therefore a much better solution to this problem was to let the server measure performance and adjust the worker thread pool by itself using performance optimization algorithms proposed by this study.

In this dissertation we first investigated the impact of worker thread pool size on the performance of a set of applications and showed that the performance is highly dependent on the worker thread pool size. Our analysis revealed several interesting findings which have been explained in Chapter 5 of this dissertation.
However the main contribution of this dissertation is an adaptive concurrency control mechanism with three algorithms to optimize the performance of a server. These optimize performance in terms of throughput, mean latency or 99th percentile of latency. These algorithms are unaware of the underlying server hardware or the type of application running on the server. This is an advantage as it prevents the proposed algorithms from being associated with a particular set of applications or server hardware. We discussed the details of these three algorithms in detail in Chapter 4. Extensive performance analysis of the proposed algorithms show that they are indeed able to automatically adjust the thread pool size for different applications, different levels of fixed incoming workloads and different levels of varying incoming workloads.

The algorithms presented in Chapter 4 of this dissertation were obtained after tuning the parameters of the algorithms. Experiments show that these parameters have a direct impact on how well the server can respond to changes in performance and on its ability to adjust the worker thread pool size.
Therefore the contributions made through this dissertation provide a practical solution to the cumbersome task of manually adjusting the worker thread pool size of a server to optimize performance.

6.4 Limitations

Although the effects of outliers and fluctuations of performance metrics on the proposed algorithms have been reduced by making increments and decrements during designated iterations of the algorithms, they may still have some impact on the thread pool size adjustment algorithms. This may cause the the thread pool size changes to not be very smooth.

6.5 Implications for further research

This research showed that it is beneficial to let a system adaptively adjust the thread pool size. This work can be improved further.

This can be improved by further reducing the effects of outliers and fluctuations of performance metrics on the thread pool size adjustment algorithms.
Furthermore, from a software engineering standpoint, this concept may be extended to develop an adaptive thread pool size adjustment framework that may be plugged into a web server with little effort.

References

[1] "Stream Processor Documentation - Stream Processor 4.3.0 - WSO2 Documentation." [Online]. Available: https://docs.wso2.com/display/SP430. [Accessed: 08-Jan-2019].

[2] "Netty: Home." [Online]. Available: https://netty.io/. [Accessed: 05-Oct-2018].

[3] "Apache JMeter - Apache JMeter™." [Online]. Available: https://jmeter.apache.org/. [Accessed: 06-Oct-2018].

[4] I. Sun Microsystems, *Multithreaded programming guide.* Place of publication not identified: iUniverse Com, 2005.

[5] B. Goetz, *Java concurrency in practice: Brian Goetz ... [et al.* Upper Saddle River, N.J: Addison-Wesley, 2013.

[6] U. Praphamontripong, S. Gokhale, A. Gokhale, and J. Gray, "Performance Analysis of an Asynchronous Web Server," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, 2006, vol. 2, pp. 22–28.

[7] R. von Behren, J. Condit, and E. Brewer, "Why Events Are a Bad Idea (for High-concurrency Servers)," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, Berkeley, CA, USA, 2003, pp. 4–4.

[8] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, 2011, pp. 116–125.

[9] E. J. Coplien, D. C. Schmidt, and D. C. Schmidt, *Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*. 1995.

[10] I. Pyarali, T. Harrison, D. C. Schmidt, and T. D. Jordan, *Proactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events*. 1997.

[11] K.-Y. Chen, J. M. Chang, and T.-W. Hou, "Multithreading in Java: Performance and Scalability on Multicore Systems," *IEEE Trans. Comput.*, vol. 60, no. 11, pp. 1521–1534, Nov. 2011.

[12] A. ADAMSON, "Specjbb2005-A year in the life of a benchmark," *2007 SPEC Benchmark Workshop*, 2007.

[13] Y. Gu, B. S. Lee, and W. Cai, “Evaluation of Java Thread Performance on Two Different Multithreaded Kernels,” *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 1, pp. 34–46, Jan. 1999.

[14] Wentong Cai, A. Hang, and P. Varman, “Benchmarking IBM SP1 system for SPMD programming,” in *Proceedings of 1996 International Conference on Parallel and Distributed Systems*, 1996, pp. 430–437.

[15] J. C. Hu, I. Pyarali, and D. C. Schmidt, “Measuring the impact of event dispatching and concurrency models on Web server performance over high-speed networks,” in *GLOBECOM 97. IEEE Global Telecommunications Conference. Conference Record*, 1997, vol. 3, pp. 1924–1931 vol.3.

[16] Y. Zheng and X. Zhang, “Static Detection of Resource Contention Problems in Server-side Scripts,” in *Proceedings of the 34th International Conference on Software Engineering*, Piscataway, NJ, USA, 2012, pp. 584–594.

[17] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, “Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores,” *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 209–220, Nov. 2014.

[18] N. Shavit and D. Touitou, “Software transactional memory,” *Distrib Comput*, vol. 10, no. 2, pp. 99–116, Feb. 1997.

[19] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, “Composable Memory Transactions,” in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2005, pp. 48–60.

[20] S. Peyton-Jones and T. Harris, “Programming in the Age of Concurrency: Software Transactional Memory.”

[21] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, “McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime,” in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2006, pp. 187–197.

[22] M. Zhang, J. Huang, M. Cao, and M. D. Bond, “Low-overhead Software Transactional Memory with Progress Guarantees and Strong Semantics,” in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2015, pp. 97–108.

[23] N. Zhou, G. Delaval, B. Robu, É. Rutten, and J.-F. Méhaut, “An autonomic-computing approach on mapping threads to multi-cores for software transactional memory,” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 18, p. e4506, 2018.

[24] “iostat(1) - Linux man page.” [Online]. Available: https://linux.die.net/man/1/iostat. [Accessed: 07-Jan-2019].

[25] “pidstat(1): Report statistics for tasks - Linux man page.” [Online]. Available: https://linux.die.net/man/1/pidstat. [Accessed: 07-Jan-2019].

[26] “Home | Metrics.” [Online]. Available: https://metrics.dropwizard.io/4.0.0/. [Accessed: 07-Oct-2018].

[27] “Java Programming Language.” [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html. [Accessed: 07-Jan-2019].

[28] “Java Software | Oracle.” [Online]. Available: https://www.oracle.com/java/. [Accessed: 07-Jan-2019].

[29] *A High Dynamic Range (HDR) Histogram. Contribute to HdrHistogram/HdrHistogram development by creating an account on GitHub*. HdrHistogram, 2018.

[30] V. Bukhtoyarov, *Collection of advanced monitoring structures with rolling time window semantic for Dropwizard-Metrics library, including integration with HDR Histogram.: vladimir-bukhtoyarov/rolling-metrics*. 2018.

Appendix A: Code Listings

A.1 AdaptiveConcurrencyControl.java

```
 public class AdaptiveConcurrencyControl {

   private static final int THREAD_POOL_MODIFICATION_INITIAL_DELAY = 10;
   private static final int THREAD_POOL_MODIFICATION_PERIOD = 10;
   private static final int PORT = 15000;
   public static Logger LOGGER =
LoggerFactory.getLogger(AdaptiveConcurrencyControl.class);

   public static void main(String[] args) throws Exception {
     if (args.length != 3) {
       LOGGER.error("Arguments not found! Please specify the 3 arguments <TestName>
<initialWorkerPoolCount> <Optimization>");
       System.exit(-1);
     }
     String testName = args[0];
     int initWorkerThreads = Integer.parseInt(args[1]);
     String optimization = args[2];
     ScheduledExecutorService threadPoolSizeModifier=
Executors.newScheduledThreadPool(1);
     CustomThreadPool thirdThreadPool = new CustomThreadPool(initWorkerThreads);
     threadPoolSizeModifier.scheduleAtFixedRate(new
ThreadPoolSizeModifier(thirdThreadPool, optimization),
                         THREAD_POOL_MODIFICATION_INITIAL_DELAY,
THREAD_POOL_MODIFICATION_PERIOD, TimeUnit.SECONDS);
     new NettyServer(PORT, testName, thirdThreadPool).start();

   }
 }
```

A.2 CustomThreadPool.java

```
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class CustomThreadPool {

  private final int KEEP_ALIVE_TIME = 100;
  private TimeUnit timeUnit = TimeUnit.SECONDS;
  private ThreadPoolExecutor executor;

  /**
  * The constructor
  *
  * @param initialPoolSize size of thread pool
  */
  public CustomThreadPool(int initialPoolSize) {
    executor = new ThreadPoolExecutor(initialPoolSize, initialPoolSize,
    KEEP_ALIVE_TIME, timeUnit,new LinkedBlockingQueue<Runnable>(), new
    ThreadPoolExecutor.CallerRunsPolicy());
  }

  /**
   * Submits a task to the thread pool
   *
   * @param task to be executed in the thread pool
   */
  public void submitTask(Runnable worker) {
    executor.execute(worker);
  }

  /**
   * Increments the pool size by n. No upper limit on the pool size
   */
  public void incrementPoolSizeBy(int n) {
    executor.setMaximumPoolSize(executor.getMaximumPoolSize() + n);
    executor.setCorePoolSize(executor.getCorePoolSize() + n);
  }

  /**
   * Decrement the pool size by n. Minimum allowed size is 1
   *
   * @param n the number to increment by
   */
  public void decrementPoolSizeBy(int n) {
    if(executor.getCorePoolSize() - n > 0 && executor.getMaximumPoolSize() - n > 0) {
      executor.setCorePoolSize(executor.getCorePoolSize() - n);
      executor.setMaximumPoolSize(executor.getMaximumPoolSize() - n);
    }
  }

  /*
   * Returns the size of the thread pool
   *
   */
  public int getThreadPoolSize() {
    return executor.getPoolSize();
  }

  public void decrementPoolSizeTo(int n) {
    if (n > 0) {
      executor.setCorePoolSize(n);
      executor.setMaximumPoolSize(n);
    }
  }

  public void incrementPoolTo(int n) {
    executor.setMaximumPoolSize(n);
    executor.setCorePoolSize(n);
  }
}
```

A.3 NettyServer.java

```
public class NettyServer {

  int port;
  String test;
  CustomThreadPool executingPool;
  Timer.Context latencyTimerContext;

  public NettyServer(int portNum, String testName, CustomThreadPool pool) {
    this.port = portNum;
    this.test = testName;
    this.executingPool = pool;
  }

  public void start() throws Exception {
    EventLoopGroup bossGroup = new NioEventLoopGroup();
    EventLoopGroup workerGroup = new NioEventLoopGroup();

  try {
    ServerBootstrap b = new ServerBootstrap();
    b.childOption(ChannelOption.SO_RCVBUF, 2147483647); // Increase receive buffer size
    b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
      .childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch) throws Exception {
          latencyTimerContext = ThreadPoolSizeModifier.LATENCY_TIMER.time();
          ChannelPipeline p = ch.pipeline();
          p.addLast(new HttpServerCodec());
          p.addLast("aggregator", new HttpObjectAggregator(1048576));
          p.addLast(new NettyServerHandler(test, executingPool, latencyTimerContext));
        }
       }).option(ChannelOption.SO_BACKLOG,
1000000).childOption(ChannelOption.SO_KEEPALIVE, true);

    ChannelFuture f = b.bind(port).sync();

    f.channel().closeFuture().sync();
  } finally {
    workerGroup.shutdownGracefully();
    bossGroup.shutdownGracefully();
  }
 }
}
```

A.4 NettyServerHandler.java

```java
public class NettyServerHandler extends SimpleChannelInboundHandler<FullHttpRequest> {

  private String testName;
  private CustomThreadPool executingPool;
  private Timer.Context timerContext;

  public NettyServerHandler(String name,CustomThreadPool pool,Timer.Context tContext) {
    this.testName = name;
    this.executingPool = pool;
    this.timerContext = tContext;
  }

  @Override
  public void channelRead0(ChannelHandlerContext ctx, FullHttpRequest msg) {

    if (testName.equals("Prime1m")) {
      executingPool.submitTask(new Prime1m(ctx, msg, timerContext));
    } else if (testName.equals("Prime10m")) {
      executingPool.submitTask(new Prime10m(ctx, msg, timerContext));
    } else if (testName.equals("DbWrite")) {
      executingPool.submitTask(new DbWrite(ctx, msg, timerContext));
    } else if (testName.equals("DbRead")) {
      executingPool.submitTask(new DbRead(ctx, msg, timerContext));
    }
  }

  @Override
  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
    cause.printStackTrace();
    ctx.close();
  }
}
```

A.5 ThreadPoolSizeModifier.java

```
public class ThreadPoolSizeModifier implements Runnable {

   public static int IN_PROGRESS_COUNT;
   public static MetricRegistry METRICS;
   public static HdrBuilder BUILDER;
   public static Timer LATENCY_TIMER;
   public static MetricRegistry METRICS2;
   public static HdrBuilder BUILDER2;
   public static Timer THROUGHPUT_TIMER;
   private static double oldTenSecondRate;
   private static double oldMeanLatency;
   private static double old99PLatency;
   public static int oldInProgressCount;
   private CustomThreadPool threadPool;
   private String optimizationAlgorithm;
   private static boolean HAS_STARTED, INC_ITR, INC_CHECK_ITR, DEC_ITR, DEC_CHECK_ITR;
   private static boolean INC_IMPROVED;
   private static boolean DEC_IMPROVED;
   int incrementLock, decrementLock;
   int resetMemory;
   private long oldCount;

   private ArrayList<Memory> metricMemory;

   /*
    * Constructor
    *
    * @param The thread pool to be modified
    */
   public ThreadPoolSizeModifier(CustomThreadPool pool, String optimization) {
     this.threadPool = pool;
     this.optimizationAlgorithm = optimization;
     metricMemory = new ArrayList<Memory>();
     METRICS = new MetricRegistry();
     BUILDER = new HdrBuilder();
     BUILDER.resetReservoirOnSnapshot();
     BUILDER.withPredefinedPercentiles(new double[] { 0.99 });
     LATENCY_TIMER = BUILDER.buildAndRegisterTimer(METRICS, "ThroughputAndLatency");
     METRICS2 = new MetricRegistry();
     BUILDER2 = new HdrBuilder();
     THROUGHPUT_TIMER = BUILDER2.buildAndRegisterTimer(METRICS2,
"ThroughputAndLatency2");
     HAS_STARTED = false;
     INC_ITR = false;
     INC_CHECK_ITR = false;
     DEC_ITR = false;
     DEC_CHECK_ITR = false;
     INC_IMPROVED = true;
     DEC_IMPROVED = true;
     AdaptiveConcurrencyControl.LOGGER.info(
                              "Thread pool size, Current 10 Second Throughput,
Throughput Difference, In pogress count, Average Latency, 99th percentile Latency");

   }
   @Override
   public void run() {
     try {
       if (HAS_STARTED == false && (LATENCY_TIMER.getCount() > 0)) {
         INC_ITR = true;
         HAS_STARTED = true;
       }
       int currentThreadPoolSize = threadPool.getThreadPoolSize();
       double currentTenSecondRate = THROUGHPUT_TIMER.getTenSecondRate();
       double rateDifference = (currentTenSecondRate - oldTenSecondRate) * 100 /
oldTenSecondRate;
       int currentInProgressCount = IN_PROGRESS_COUNT;
        Snapshot latencySnapshot = LATENCY_TIMER.getSnapshot();
```

```
        double currentMeanLatency = latencySnapshot.getMean() / 1000000;
        double current99PLatency = latencySnapshot.get99thPercentile() / 1000000;
        AdaptiveConcurrencyControl.LOGGER.info(currentThreadPoolSize + ", " +
currentTenSecondRate + ", " + rateDifference + ", " + currentInProgressCount + ", " +
currentMeanLatency + ", " + current99PLatency);
        if (optimizationAlgorithm.equals("T")) { // If Throughput Optimized
        if ((DEC_ITR == false || (DEC_ITR == true && DEC_IMPROVED == false)) &&
(((oldTenSecondRate - currentTenSecondRate) / oldTenSecondRate) * 100 > 10) &&
resetMemory != 300) {
          Memory current;
          for (int i = 0; i < metricMemory.size(); i++) {
            current = metricMemory.get(i);
          }
          for (int i = 0; i < metricMemory.size() - 1; i++) {
            if (metricMemory.get(i).getThreadPoolSize() == 0) {
              continue;
            }
            if (currentTenSecondRate <= (metricMemory.get(i).getValue()+
metricMemory.get(i + 1).getValue()) / 2) {
              if (currentThreadPoolSize < metricMemory.get(i).getThreadPoolSize()) {
                threadPool.incrementPoolTo(metricMemory.get(i).getThreadPoolSize());
              }else if(currentThreadPoolSize>metricMemory.get(i).getThreadPoolSize()) {
               threadPool.decrementPoolSizeTo(metricMemory.get(i).getThreadPoolSize());
              }
              break;
            }
          }
        }
        if (INC_ITR == true && INC_IMPROVED == true) {
          threadPool.incrementPoolSizeBy(10);
        }
        if (INC_CHECK_ITR == true && INC_IMPROVED == true) {
          if ((((currentTenSecondRate - oldTenSecondRate) / oldTenSecondRate ) * 100 <
10) {
            INC_IMPROVED = false;
            threadPool.decrementPoolSizeBy(10);
            incrementLock = 6; // Prevent increments for the next 8 sets of iterations
          }
        }
        Memory current = new Memory();
        boolean isInAList = false;
        for (int i = 0; i < metricMemory.size(); i++) {
          current = metricMemory.get(i);
          if (current.getThreadPoolSize() == currentThreadPoolSize) {
            isInAList = true;
            current.setValue(((current.getValue() * current.getCount()) +
currentTenSecondRate) / (current.getCount() + 1));
            current.setCount(current.getCount() + 1);
            break;
          }
        }
        if (isInAList == false) {
          metricMemory.add(new Memory(currentThreadPoolSize, 1, currentTenSecondRate));
        }
       }
       if (optimizationAlgorithm.equals("M")) { // If Mean latency Optimized
         if (INC_ITR == true && INC_IMPROVED == true) {
           threadPool.incrementPoolSizeBy(10);
         }
         if (INC_CHECK_ITR == true && INC_IMPROVED == true) {
           if ((((oldMeanLatency - currentMeanLatency) / oldMeanLatency) * 100) < 5) {
             INC_IMPROVED = false;
             threadPool.decrementPoolSizeBy(10);
             incrementLock = 6; // Lock increments
           }
         }
         if (DEC_ITR == true && DEC_IMPROVED == true) {
           threadPool.decrementPoolSizeBy(10);
         }
```

```
 if (DEC_CHECK_ITR == true && DEC_IMPROVED == true) {
   if ((((oldMeanLatency - currentMeanLatency) / oldMeanLatency) * 100) < 5) {
   DEC_IMPROVED = false;
  threadPool.incrementPoolSizeBy(10);
  decrementLock = 6; // Lock decrements
  }
 }
}
if (optimizationAlgorithm.equals("99P")) {
  if (INC_ITR == true && INC_IMPROVED == true) {
    threadPool.incrementPoolSizeBy(10);
  }
    if (INC_CHECK_ITR == true && INC_IMPROVED == true) {
      if ((((old99PLatency - current99PLatency) / old99PLatency) * 100) < 5) {
         INC_IMPROVED = false;
         threadPool.decrementPoolSizeBy(10);
         incrementLock = 6; // Lock increments
      }
    }
  if (DEC_ITR == true && DEC_IMPROVED == true) {
    threadPool.decrementPoolSizeBy(10);
  }
  if (DEC_CHECK_ITR == true && DEC_IMPROVED == true) {
    if ((((old99PLatency - current99PLatency) / old99PLatency) * 100) < 5) {
      DEC_IMPROVED = false;
      threadPool.incrementPoolSizeBy(10);
       decrementLock = 6; // Lock decrements
    }
  }
}
oldTenSecondRate = currentTenSecondRate;
oldMeanLatency = currentMeanLatency;
old99PLatency = current99PLatency;
oldInProgressCount = currentInProgressCount;
if (INC_ITR == true) { //
   INC_ITR = false;
    INC_CHECK_ITR = true;
    DEC_ITR = false;
    DEC_CHECK_ITR = false;
} else if (INC_CHECK_ITR == true) { //
  if (optimizationAlgorithm.equals("T")) {
    INC_ITR = true;
    DEC_ITR = false;
  } else {
    INC_ITR = false;
    DEC_ITR = true;
  }
  INC_CHECK_ITR = false;
  DEC_CHECK_ITR = false;
} else if (DEC_ITR == true) {
  INC_ITR = false;
  INC_CHECK_ITR = false;
  DEC_ITR = false;
  DEC_CHECK_ITR = true;
} else if (DEC_CHECK_ITR == true) {
  INC_ITR = true;
  INC_CHECK_ITR = false;
  DEC_ITR = false;
  DEC_CHECK_ITR = false;
}
if (incrementLock > 0) {
  incrementLock--;
} else if (incrementLock == 0) {
  INC_IMPROVED = true;
}
if (decrementLock > 0) {
  decrementLock--;
} else if (decrementLock == 0) {
DEC_IMPROVED = true;
}
 long currentCount = LATENCY_TIMER.getCount();
```

```
        if ((currentCount - oldCount == 0) && (threadPool.getThreadPoolSize()>1) ) {
          threadPool.decrementPoolSizeTo(1);
        }
        oldCount = currentCount;
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
}
```

A.6 Memory.java

```
public class Memory {
  int threadPoolSize;
  int count;
  double value;

public Memory(int size, int c, double val) {
  this.threadPoolSize = size;
  this.count = c;
  this.value = val;
}

public Memory() {
}

public int getThreadPoolSize() {
  return threadPoolSize;
}

public void setThreadPoolSize(int threadPoolSize) {
  this.threadPoolSize = threadPoolSize;
}

public int getCount() {
  return count;
}

public void setCount(int count) {
  this.count = count;
}

public double getValue() {
  return value;
}

public void setValue(double value) {
  this.value = value;
}

}
```

A.7 Prime1m.java

```
 public class Prime1m implements Runnable {

   private FullHttpRequest msg;
   private ChannelHandlerContext ctx;
   private Timer.Context timerContext;

   public Prime1m(ChannelHandlerContext ctx,FullHttpRequest msg,Timer.Context
timerCtx) {
     this.msg = msg;
     this.ctx = ctx;
     this.timerContext = timerCtx;
   }

   @Override
   public void run() {
   Timer.Context
throughputTimerContext=ThreadPoolSizeModifier.THROUGHPUT_TIMER.time();
   ByteBuf buf = null;
   try {
     ThreadPoolSizeModifier.IN_PROGRESS_COUNT++;
     Random rand = new Random();
     int number = rand.nextInt((1000021) - 1000000 ) + 1000000;
     String resultString = "true";
     for (int i=2; i<number; i++) {
       if (number%i == 0) {
         resultString="false";
          break;
        }
     }
     buf = Unpooled.copiedBuffer(resultString.getBytes());
     ThreadPoolSizeModifier.IN_PROGRESS_COUNT--;
   } catch (Exception e) {
   AdaptiveConcurrencyControl.LOGGER.error("Exception in Prime1m Run method", e);
   }

   boolean keepAlive = HttpUtil.isKeepAlive(msg);
   FullHttpResponse response = null;
   try {
     response = new DefaultFullHttpResponse(HTTP_1_1, OK, buf);
   } catch (Exception e) {
     AdaptiveConcurrencyControl.LOGGER.error("Exception in Netty Handler", e);
   }
   String contentType = msg.headers().get(HttpHeaderNames.CONTENT_TYPE);
   if (contentType != null) {
     response.headers().set(HttpHeaderNames.CONTENT_TYPE, contentType);
   }
   response.headers().setInt(HttpHeaderNames.CONTENT_LENGTH,
response.content().readableBytes());
   if (!keepAlive) {
     ctx.write(response).addListener(ChannelFutureListener.CLOSE);
   } else {
     response.headers().set(HttpHeaderNames.CONNECTION, HttpHeaderValues.KEEP_ALIVE);
     ctx.write(response);
   }
   ctx.flush();
   throughputTimerContext.stop();
   timerContext.stop(); // Stop Dropwizard metrics timer
  }
}
```

A.8 Prime10m.java

```
 public class Prime10m implements Runnable {

   private FullHttpRequest msg;
   private ChannelHandlerContext ctx;
   private Timer.Context timerContext;

   public Prime10m(ChannelHandlerContext ctx,FullHttpRequest msg,Timer.Context
timerCtx) {
     this.msg = msg;
     this.ctx = ctx;
     this.timerContext = timerCtx;
   }

   @Override
   public void run() {
   Timer.Context
throughputTimerContext=ThreadPoolSizeModifier.THROUGHPUT_TIMER.time();
   ByteBuf buf = null;
   try {
     ThreadPoolSizeModifier.IN_PROGRESS_COUNT++;
     Random rand = new Random();
     int number = rand.nextInt((10000021) - 10000000 ) + 10000000;
     String resultString = "true";
     for (int i=2; i<number; i++) {
       if (number%i == 0) {
         resultString="false";
          break;
        }
     }
     buf = Unpooled.copiedBuffer(resultString.getBytes());
     ThreadPoolSizeModifier.IN_PROGRESS_COUNT--;
   } catch (Exception e) {
   AdaptiveConcurrencyControl.LOGGER.error("Exception in Prime10m Run method", e);
   }

   boolean keepAlive = HttpUtil.isKeepAlive(msg);
   FullHttpResponse response = null;
   try {
     response = new DefaultFullHttpResponse(HTTP_1_1, OK, buf);
   } catch (Exception e) {
     AdaptiveConcurrencyControl.LOGGER.error("Exception in Netty Handler", e);
   }
   String contentType = msg.headers().get(HttpHeaderNames.CONTENT_TYPE);
   if (contentType != null) {
     response.headers().set(HttpHeaderNames.CONTENT_TYPE, contentType);
   }
   response.headers().setInt(HttpHeaderNames.CONTENT_LENGTH,
response.content().readableBytes());
   if (!keepAlive) {
     ctx.write(response).addListener(ChannelFutureListener.CLOSE);
   } else {
     response.headers().set(HttpHeaderNames.CONNECTION, HttpHeaderValues.KEEP_ALIVE);
     ctx.write(response);
   }
   ctx.flush();
   throughputTimerContext.stop();
   timerContext.stop(); // Stop Dropwizard metrics timer
  }
}
```

A.9 DbWrite.java

```
 public class DbWrite implements Runnable {

   private FullHttpRequest msg;
   private ChannelHandlerContext ctx;
   private Timer.Context timerContext;

  public DbWrite(ChannelHandlerContext ctx, FullHttpRequest msg, Timer.Context
timerCtx) {
    this.msg = msg;
    this.ctx = ctx;
    this.timerContext = timerCtx;
  }
  @Override
  public void run() {
  Timer.Context throughputTimerContext=
ThreadPoolSizeModifier.THROUGHPUT_TIMER.time();
  ByteBuf buf = null;
  try {
    ThreadPoolSizeModifier.IN_PROGRESS_COUNT++;
    Connection connection = null;
    PreparedStatement stmt = null;
    try {
      Connection =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/echoserver?
useSSL=false&autoReconnect=true&failOverReadOnly=false&maxReconnects=10",
"root", "root");
      Timestamp current = Timestamp.from(Instant.now());
      String sql = "INSERT INTO Timestamp (timestamp) VALUES (?)";
      stmt = connection.prepareStatement(sql);
      stmt.setTimestamp(1, current);
      stmt.executeUpdate();
      buf = Unpooled.copiedBuffer(current.toString().getBytes());
    } catch (Exception e) {
       AdaptiveConcurrencyControl.LOGGER.error("Exception", e);
    } finally {
       if (stmt != null) {
        try {
          stmt.close();
       } catch (Exception e) {
       AdaptiveConcurrencyControl.LOGGER.error("Exception", e);
    }
    }
    if (connection != null) {
      try {
       connection.close();
      } catch (Exception e) {
        AdaptiveConcurrencyControl.LOGGER.error("Exception", e);
      }
     }
    }
    ThreadPoolSizeModifier.IN_PROGRESS_COUNT--;
    } catch (Exception e) {
       AdaptiveConcurrencyControl.LOGGER.error("Exception in DbWrite Run method", e);
    }

   boolean keepAlive = HttpUtil.isKeepAlive(msg);
   FullHttpResponse response = null;
   try {
     response = new DefaultFullHttpResponse(HTTP_1_1, OK, buf);
   } catch (Exception e) {
     AdaptiveConcurrencyControl.LOGGER.error("Exception in Netty Handler", e);
   }
   String contentType = msg.headers().get(HttpHeaderNames.CONTENT_TYPE);
   if (contentType != null) {
     response.headers().set(HttpHeaderNames.CONTENT_TYPE, contentType);
   }
   response.headers().setInt(HttpHeaderNames.CONTENT_LENGTH,
response.content().readableBytes());
```

```
if (!keepAlive) {
     ctx.write(response).addListener(ChannelFutureListener.CLOSE);
   } else {
     response.headers().set(HttpHeaderNames.CONNECTION, HttpHeaderValues.KEEP_ALIVE);
     ctx.write(response);
   }
   ctx.flush();
   throughputTimerContext.stop();
   timerContext.stop(); // Stop Dropwizard metrics timer
  }
}
```

A.10 DbRead.java

```
 public class DbRead implements Runnable {

   private FullHttpRequest msg;
   private ChannelHandlerContext ctx;
   private Timer.Context timerContext;

  public DbRead(ChannelHandlerContext ctx, FullHttpRequest msg, Timer.Context
timerCtx) {
    this.msg = msg;
    this.ctx = ctx;
    this.timerContext = timerCtx;
  }
  @Override
  public void run() {
  Timer.Context throughputTimerContext=
ThreadPoolSizeModifier.THROUGHPUT_TIMER.time();
  ByteBuf buf = null;
  try {
    ThreadPoolSizeModifier.IN_PROGRESS_COUNT++;
    Connection connection = null;
    PreparedStatement stmt = null;
    try {
        Random randId = new Random();
        int toRead = randId.nextInt(50000) + 1;
        connection =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/echoserver?
useSSL=false&autoReconnect=true&failOverReadOnly=false&maxReconnects=10",
                                          "root", "root");
        String sql = "SELECT timestamp FROM Timestamp WHERE id=?";
        stmt = connection.prepareStatement(sql);
        stmt.setInt(1, toRead);
        rs = stmt.executeQuery();
        while (rs.next()) {
          readTimestamp = rs.getTimestamp("timestamp");
        }
    } catch (Exception e) {
        AdaptiveConcurrencyControl.LOGGER.error("Exception", e);
    } finally {
        if (stmt != null) {
         try {
           stmt.close();
        } catch (Exception e) {
        AdaptiveConcurrencyControl.LOGGER.error("Exception", e);
    }
    }
```

```
    if (connection != null) {
      try {
       connection.close();
      } catch (Exception e) {
        AdaptiveConcurrencyControl.LOGGER.error("Exception", e);
      }
     }
    }
     String readTimestampStr = readTimestamp.toString() + "\n";
     buf = Unpooled.copiedBuffer(readTimestampStr.getBytes());
     ThreadPoolSizeModifier.IN_PROGRESS_COUNT--;
    } catch (Exception e) {
      AdaptiveConcurrencyControl.LOGGER.error("Exception in DbRead Run method", e);
    }
   boolean keepAlive = HttpUtil.isKeepAlive(msg);
   FullHttpResponse response = null;
   try {
     response = new DefaultFullHttpResponse(HTTP_1_1, OK, buf);
   } catch (Exception e) {
     AdaptiveConcurrencyControl.LOGGER.error("Exception in Netty Handler", e);
   }
   String contentType = msg.headers().get(HttpHeaderNames.CONTENT_TYPE);
   if (contentType != null) {
     response.headers().set(HttpHeaderNames.CONTENT_TYPE, contentType);
   }
   response.headers().setInt(HttpHeaderNames.CONTENT_LENGTH,
response.content().readableBytes());
   if (!keepAlive) {
     ctx.write(response).addListener(ChannelFutureListener.CLOSE);
   } else {
     response.headers().set(HttpHeaderNames.CONNECTION, HttpHeaderValues.KEEP_ALIVE);
     ctx.write(response);
   }
   ctx.flush();
   throughputTimerContext.stop();
   timerContext.stop(); // Stop Dropwizard metrics timer
  }
}
```