Machine learning approach to fine-tune Ballerina thread pool size

Lakindu Akash

Machine learning approach to fine-tune Ballerina thread pool size

L A S J Gunasekara
Index No : 16000536

Supervisor: Dr. Chamath Keppitiyagama
Co-Supervisor: Dr. Malith Jaysinghe

January 2021

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: L A S J Gunasekara

.....

Signature of Candidate Date: 07/02/2020

This is to certify that this dissertation is based on the work of Mr. L A S J Gunasekara under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Principle Supervisor's Name: Dr. Chamath Keppitiyagama

Co- Supervisor's Name: Dr. Malith Jayasinghe

.....

Signature of Supervisor Date: 07/02/2020

Abstract

Web server optimization is important for businesses since web servers can achieve higher performance consuming less resources. This research present optimization techniques for different program types based IO characteristics. Several web server architectures are implemented in Ballerina and then tuned. Performance of each server architecture and tunes are evaluated against different types of program where CPU and IO intensive features are prominent in each program. For further tuning thread pool size is selected based on experimental results. Then optimal thread pool size is identified for each set of programming features where features are different types of remote IO calls. Since IO calls are part of the native representation of ballerina, a parser is implemented to extract these features automatically from Abstract Syntax Tree of the Ballerina source code. Finally, a machine learning model is proposed where optimal thread pool size is the output of given set of programming features. Finally, it could be observers that for different program types, the latency when the optimal thread pool size suggested by the ML approach is used is lower than the latency when Ballerina's default thread pool size is used.

Preface

The purpose of this dissertation is to present the work from the research named "Machine learning approach to fine-tune Ballerina thread pool size".

Acknowledgment

Foremost, I would like to express my sincere gratitude to my supervisor **Dr. Chamath Keppetiyagama** and my co-supervisor **Dr. Malith Jayasinghe** for the continues support of my study for whole one year,their patience, motivation and immense knowledge. Their guidance allowed me to improve my thinking,writing and presenting skills.

Besides my supervisors I would like to thank **Ms. Duneesha Fernando** for reviewing my work, supporting to write this thesis and organizing meetings in order to bring this study up to this level.

Also, my sincere thanks goes for my batch-mates who helped to succeed this project by allowing me to discuss problems arisen while continuing this research.

Last but not least, I would like to thank my family members who supported me in all other ways in hard times of me.

Contents

List of Figures

List of Tables

List of Acronyms

AST Abstract Syntax Tree. 12

NIO Non Blocking IO. 9

OIO Old IO. 33

SEDA Staged Event Deriven Architecture. 7, 8

SYMPED SYmmetric Multi-Processor Event Driven. 8

VM Virtual Machine. 29

Chapter 1

Introduction

In this research we explored various optimization techniques based on programming features for web servers. Initially this research started with idea of finding optimal web server architecture based on programming features of Ballerina language. Several architectural changes were made to original Ballerina server architecture. Thereafter, performance of each server architecture is evaluated with respect to different programs. Furthermore, server architectures were fine-tuned and it was able to identify that varying the thread pool size of the existing architecture is the best way maximize the performance for given set of programming features. Finally, machine learning model was proposed to output the optimal thread pool size for given set of programming features.

1.1 Background to the Research

Web server optimization is prominent branch in the enterprise level software industry as well as e-businesses.Then services able to facilitate greater performance consuming less computation resources. This research came along long way from comparing performance of different server architectures to thread pool optimization. Prior determination of web sever architecture with best performance is difficult due to various factors such as type of operations that execute, workload, working deployment environment etc. There are several researches conducted for both performance of server architectures and thread pool tuning. Pariag at el. [1] have compared performance of different server architectures under different work-

loads. Afterward, server architectures are extensively fine-tuned to optimize performance. Furthermore, performance of architectures were maximized for workloads that perform IO. Several other studies have also discussed [2, 3, 4] advantages and drawbacks of each sever architectures. Apart from that, several studies had pointed out that thread pool tuning able to maximize the performance of web servers. Several mathematical models [5, 6, 7, 8, 9] and dynamic thread pool optimization techniques [10, 11, 12] had been proposed in these studies.

1.2 Research Problem and Research Questions

This research addresses the following questions, In order to understand following research questions it is important to have idea about what is reffered to web sever architecture explained in subsection 1.6

- Is it possible to find a server architecture or parameters that would result in the best performance for a given Ballerina program?
- Is it possible to tune the thread pool size of a given server architecture in order to optimize the performance (i.e. throughput and latency)? If so what is the impact of the thread pool size on the performance for different Ballerina programs?
- Is it possible devise a model that can estimate the thread pool size that would result in the best performance for a given Ballerina program?

1.3 Justification for the research

Debate of state-of-the-art web server architecture is progressing and researches try to compare the performance of web server architectures introducing various optimization techniques. In previous researches, new architectures were proposed by fine-tuning existing architectures in order to increase the overall performance and increase the performance for different workloads as well. Nevertheless, performance of server architectures were not compared against different types of programming features specially for various CPU intensive and IO intensive applications. This research addresses the following areas,

- Implement new server architectures in Ballerina based on popular server architecture models, then performance is compared against different types of program
- Identify parameters further in order to optimize the performance for different types of programming features

- A machine learning model is proposed for predicting the optimal parameters (In this research it is thread pool size) given set of programming features

Initial idea of the research was providing solid statement on one server architecture perform well for one type of program while performance of another server architecture is better for another type of program by extensive tuning the server architectures. Based on results gathered focus was later concentrated on tuning the thread pool size.

None of previous researches have proposed a methodology for identification of programming features automatically and propose the web server architecture or optimal thread pool size for given set of programming features. Furthermore, Ballerina is a better candidate for the research because identification of the programming features such as IO operation is more convenient in Ballerina. Features that are difficult to extract in other languages are part of the Ballerina language.

1.4 Methodology

This research started with implementing new web server architectures in Ballerina and then evaluating the performance of each including the existing architecture (Refer figure 3.1). These architectures are inspired from Thread per connection model and event based models(Ex: SEDA). This step includes modifying the Ballerina source code. Proper implementation requires deep analysis of Ballerina run time and optimization for the implementations, otherwise bugs can affect the performance.

Performance are evaluated with test programs which consist of different programming features. As an example one program may consists of several database calls and another program only contain some arithmetic operations. Then performance are evaluated according to program features. The target is to provide best performant architecture for given features. Fine-tunes are carried out based on results of each phase. That led the research to focus on thread pool tuning after number of experiments.

Final model is consisted of a machine learning model that provide optimal thread pool size for given program. Figure 1.1 shows the overview of the research.

Results are dependent of the deployment environment of the service. Hence, first user need to train the machine learning model with predefined set of programs. The user able to provide own program and the model predict the optimal thread pool size for that program. These steps are explained more detail in Chapter 3.

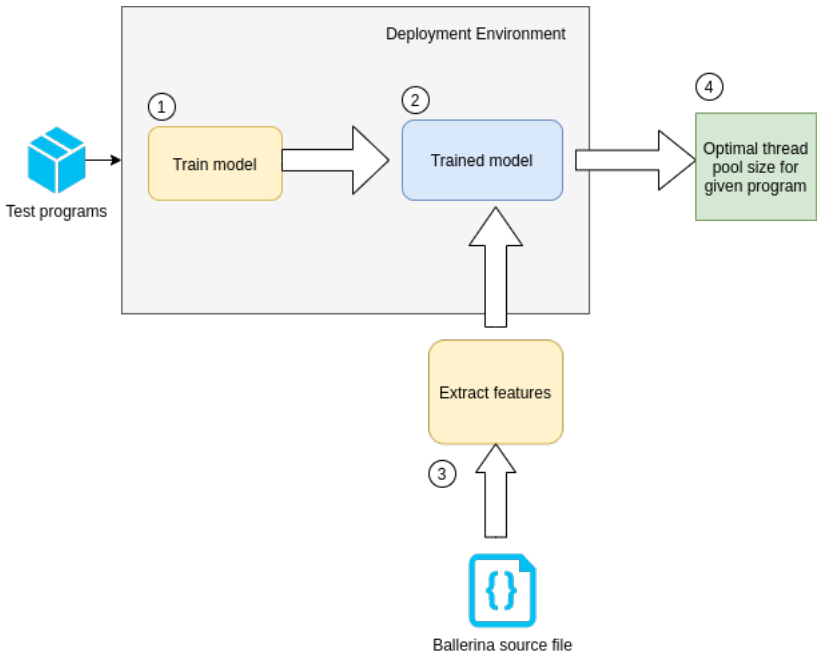

Figure 1.1: High-level overview of the research

1.5 Outline of the Dissertation

This dissertation outlines 6 chapters as follows.

1. Introduction — Contains small background study, research problems, justification and overview of methodology.
2. Literature Review — Present related studies and theories for this research.
3. Research Design — Design of the research with respect to each stage of the research.
4. Implementation — Provide important implication carried out in order to conduct experiments and obtain results.
5. Results & Evaluation — Contain the results obtained in each phase and in depth discussion of the results.

6. Conclusion — Discuss how the results mentioned in Chapter 5 has addressed the research questions. An emphasis will be given in highlighting the contribution of this research work to the scientific world. Implications of further research will also be discussed in this chapter.

1.6 Definitions

Concurrency level

In this research Concurrency level and Concurrent users are used interchangeably referred to concurrent users who are connected the web server. This value can be simulated in load testing software. A good web server must be able to perform well in higher number of concurrency level.

Deployment environment

Deployment environment is where web server is deployed and running. Configuration of the environment includes the Operating system, Number of CPU core, Speed of the CPU, Amount of RAM and Network stability etc.

Web Server architecture

Web server architecture provide clear abstraction of what are the components of web server, how the components are interacting each other and behavior of each component when client request is processed. Thread per connection model [1] and SEDA [2] is such two popular server architectures. Thread per connection model spawn new thread per each client request while SEDA reuse the threads by pooling techniques. New server architectures can be implemented by combining existing model or changing parameters such as size of thread pool. Major changes such as adding new components, combining them can be considered as a new architecture while changing parameters such as defining minimum and maximum number of threads can be expressed as fine-tuning. Chapter 2 provide in-depth explanation thread pools.

1.7 Delimitations of Scope

1.7.1 In-Scope

- Implement new server architectures in Ballerina by modifying the existing implementation. This includes adding and removing new thread pools. Then performance is evaluated using relevant metrics .
- Fine tune and optimize the new architectures by identifying bugs and extensive debugging — these tuning includes the changing thread pool size.
- Implementing different types of program which have CPU intensive and IO intensive features. Standard ballerina benchmark tools are also referred [32].
- Justification of used tools and accuracy of
- Identify different program features that affect the performance of web sever.
- Compare relevant metrics of several machine learning models and hyper parameter tuning of each model in order to get accurate model.
- Compare performance of proposed model and current Ballerina sever architecture.

1.7.2 Out-Scope

- Generalizing the findings for other languages and frameworks.
- Evaluate and compare the results for different deployment environments which have different hardware.
- Design and implementation of analytical (a.k.a white box model in this research) model (queuing theory approach etc.) to predict optimal thread pool size.

1.8 Conclusion

This chapter provides base principles and presents a foundation to the dissertation. At the beginning, the research problem is identified, formulated research questions and research is justified. Then the definitions is presented, dissertation is outlined, research design and implementation is briefly discussed laying the path to proceed with detailed explanation of the research.

Chapter 2

Literature Review

2.1 Overview

This section provides a broader context of this area with regard to other researches and studies. Literature review consist of following areas.

1. Characteristics of IO
2. Performance of web server architectures
3. Importance of thread pool tuning
4. Appropriateness of Ballerina for this research

2.2 Background and related theories

2.2.1 About IO

Basically every program instruction can be divided as IO and non IO operations. IO operations are typically operations which do not use the CPU. When there is an API call that requests data from IO, that call won't immediately return, but with some delay. [13] This delay can be very small for requesting a file on a hard drive, and much longer when requesting data from a network. This is because the data that is requested from IO devices has to travel longer to the caller. For example, call to an external HTTP server has to wait more rather than requesting data from server where hosted in same machine. There are two types of IO,

1. Blocking IO

2. Non-blocking IO

Calling an operation that requests data from IO will cause the running thread to "block", i.e. it is waiting until the requested data has returned to the caller. When a thread is blocked in Linux, it will be put in a Sleep state by the kernel until data has returned to the caller. Threads in sleep state immediately give up its access to the CPU, so as to not waste CPU time. After IO is ready, the thread is taken out from the Sleep state and put in a Runnable state. Threads in this state are eligible to be executed on the CPU again. The thread scheduler will put the thread on a CPU when one is available. The process of taking threads on and off the CPU is called context switching where this process is done in OS level. Context switching is quite expensive because it has to save the data related to its state called context and when it is ready to run, data must be restored again into CPU registers and stacks. Allocated memory size of thread in java is 1 Mb by default. A web sever receiving 1000 concurrent request may result of 1000Mb usage of RAM only to maintain thread. Moreover, when there are large number of context switching is happening larger overhead is added to the system. This is where disadvantage of thread per connection model is apparent.

Non-blocking IO reduce the context switching by reusing the threads. The main benefit of non-blocking IO is that we need fewer threads to handle the same amount of IO requests. When there is a IO operation need to be performed the task is delegated to OS and that call is kept in a queue, then thread is released for other operations. When IO is completed, queue is notified and further instructions of the task are carried out afterward. Non-blocking IO is tightly coupled with thread pools. Fewer number of threads are allocated in advance and tasks executed by those threads by reusing. Therefore, context switching is minimized.

2.2.2 Event loops

To facilitate the non-blocking IO most programming languages implement an event loop. The concept of event can be any operation including IO operations needed to be executed. The event loop polls (constantly check) if data is returned from

IO. An event loop is basically a while(true) loop that in each iteration will check if data is ready to read from a File Descriptor in Unix systems [14]. The list of File descriptors that we want to check for ready data is called the interest list. At glance, we can think of the event loop as an expensive task. Although there are several optimization techniques that have been implemented at kernel level to reduce the cost of event loop. Some examples are epoll, io_uring, kqueue [15, 16].

2.2.3 Web Server architecture performance

There are several web server architectures that have existed for a long time. In the early 90s thread per connection model was popular because it is relatively easier to implement. But when the internet is getting popular web servers get more hits. Thread per connection model has serious flaws when there are higher numbers of concurrency reaches. Since each thread needs a considerable amount of memory and overhead of context switching [2]. Having thousands of connections to such a server cannot handle large requests. In order to overcome this problem SEDA (Staged event driven architecture) [2] were proposed. It limits the thread count using concept of using thread pool, thus it reduces the context switching and memory usage. SEDA is a very successful mechanism to handle large concurrencies. Although the question had arisen "do SEDA perform well in every situation?". [17, 3, 18] This debate has a long history. To answer this there were many researches conducted and proposed several fine-tuned architectures for refined situations.

Bharti et al. [19] proposed a fine Grained SEDA Architecture for Service Oriented Network Management Systems in order to improve the performance architecture for specific functions. They are fine-tuning SEDA by breaking the stage in SEDA into sub-stages. Each sub-stage represents a SEDA implementation of a critical functionality of the parent SEDA stage which each stage consist of single thread pool. In those stages, performance has been improved by micro-managing the functionalities of each sub stage. Hence, the application, instead of having a single stage with competing threads from their functionalities, will now have fine-grained sub-stages. Each of the fine-grained sub-stages will now host threads performing a much smaller set of tasks. They showed that design has significant performance improvement of the application.

Pariag et al. [1] showed extensive tuning of web server architectures able to significantly improve the performance of specific types of workloads. Workload refers to what type of operation expect to carried out in the web server, specially IO operations. They have considered 3 different server libraries that have different server architectures, the μserver utilizes an event-driven architecture, Knot that uses the highly-efficient Capriccio thread library for implementing a thread per connection model, and WatPipe that uses a hybrid of events and threads to implement a pipeline based server that is similar to staged event-driven architecture Staged Event Deriven Architecture (SEDA) server like Haboob [2]. They have modified the Capriccio thread library to use Linux's zero-copy sendfile interface. Then they have introduced the SYmmetric Multi-Processor Event Driven (SYMPED) architecture in which relatively minor modifications are made to a single process event-driven (SPED) [20] server (the μserver) to allow it to continue processing requests in the presence of blocking IO due to disk accesses. They finally describe a C++ implementation of WatPipe, which although utilizing a pipeline-based architecture, excludes the dynamic controls over event queues and thread pools used in SEDA.

They have compared all three architectures and the conclusion is slightly different from the previous studies. One important conclusion of them related to this research is they have observed that when using blocking sockets to send data to clients, the performance obtained with some architectures is quite good and in one case is noticeably better than when using non-blocking sockets. They have shown that a proper combination of the number of connections and kernel-level worker threads (or processes) is required to get the best performance results for each workload type.

In their conclusion they have stated "this work could be done after or in conjunction with work that enables servers to automatically and dynamically tune them-selves to efficiently execute the offered workload"

More recent similar works [21, 22] can be found regarding the improving the performance of web server architectures. Thus, we can conclude performance of web server architecture is dependent on type of workload or type of operations which web server executes. Tuning parameters such as thread pool size of web

server also able to improve the performance.

Behren et al. [3] claiming that "Events Are A Bad Idea (for high-concurrency servers)". They have shown comparison of benchmark results of threaded and event based architectures. In their conclusion stated "Although event systems have been used to obtain good performance in high concurrency systems, we have shown that similar or even higher performance can be achieved with threads". But there is a question left on whether it hold true only for the Java NIO library or for the non-blocking approach in general.

2.2.4 Ballerina language

As growing needs of the distributed computing rapid development of web services, new programming languages and paradigms were emerged. Ballerina [23] is such a successful language that handles network programming in a developer friendly way. Here I quoted a small paragraph from the ballerina website. "For decades, programming languages have treated networks simply as I/O sources. Ballerina introduces fundamental, new abstractions of client objects, services, resource functions, and listeners to bring networking into the language so that programmers can directly address the Fallacies [24] of Distributed Computing as part of their application logic. This facilitates resilient, secure, performant network applications to be within every programmer's reach."

Apart from the above quotation ,Ballerina is best suited for this research since Ballerina language has certain capabilities extracting performance critical features especially IO calls. In Ballerina, they are called connector calls. In order to answer the second research question this provides huge benefits as network calls are first class citizens in Ballerina languages. Chapter 3 explains more about these features.

While other languages provide network operations as library functions Ballerina provides in native way. If we try to extract such features from other languages we need to know exactly this function call do an IO operation including library functions. That is very difficult task to extract such features just exploring the source code in other languages like C,Java etc.

Network services are first order objects in ballerina language like functions. [25] Such features can be used to identify whether a given program contain network

services or not. In other languages we need to have prior knowledge about which libraries are used for such implementations and which part of the code implement the network services.

There are some tools already developed [26, 27] using these features in order to represent the code as sequence diagrams. Figure 2.1 shows a generated sequence diagram for a code which includes a remote database call.

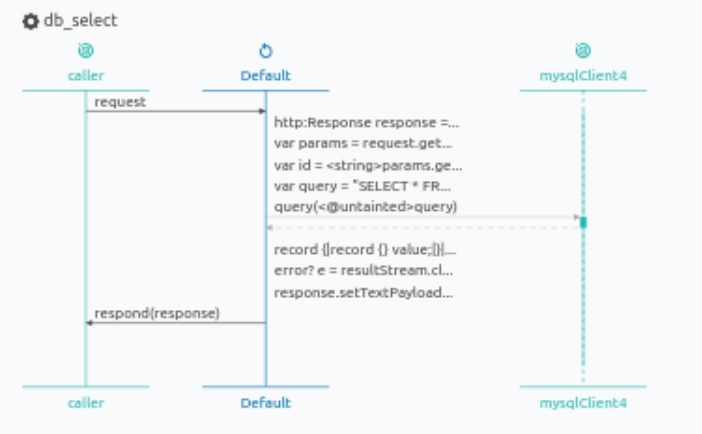

Figure 2.1: Sequence diagram representation of code

It is clear that we can use the network awareness properties of the Ballerina language to identify performance critical features directly from source code unlike other languages. Examples of network awareness features are creating RESTful clients and servers ,making database queries , completing asynchronous calls ,producing and consuming streaming data message-passing etc [25].

2.2.5 Thread pool tuning

Thread pools are very important component when implementing server architectures. In SEDA every stage has its own thread pool. Creating and destroying a thread is resource intensive task for the OS [6]. Instead of creating and destroying threads on demand, thread pool reuses the allocated number of threads. Size of the thread pool is almost fixed, although there are many studies to predict the number of threads in the pool dynamically using various approaches in order to gain the performance. Some approaches propose analytical models and schemes [5, 6, 7, 8, 9] and other approaches use run-time information [10, 11, 12] to predict size of the pool. Most server implementation decide the number of thread

in the pool heuristically and based on empirical knowledge [6, 7] to get balanced performance for every type of operations including IO since integrating analytical modeling is complex and calculation of run-time information can add an overhead to the server.

When discussing some heuristic and experience based approaches to decide the thread pool size, one rule is that the size of thread pool should be two times of the number of CPUs (processors). Ballerina scheduler is also following this rule. Another rule, used by Microsoft's IIS, initially allocates 10 threads in the thread pool per CPU, and allows the size of the thread pool to increase based on the number of client requests [6]. The thread pool's maximum size is heuristically set to be two times of the number of megabytes of RAM in the host machine. Those heuristics are based on memory and CPU capabilities and there is no theoretical justification.

Yibei Ling et al. [6] propose a mathematical model to predict optimal thread pool size by calculating cost of spawning, destroying and maintaining threads in a thread pool. However, this approach has some flaws because of assumptions they initially made. First assumption is that each thread in the pool has the same execution priority and receive uniform treatment. This is not always true because OS has limited number of processors and there are other OS level processes also using those resources. Typical modern web servers are deployed with resource control and monitoring tools as well. Thus threads may not get equal access to CPUs. The next assumption is one web request is very much like another and that the memory and I/O resources required by different threads are minimal. This research's major focus is to optimize the sever architectures for different types of programming features specially IO.

D.L. Freire et al.[7] also did a similar study to calculate expected gain of web servers under different thread pool sizes using a analytical model. The optimal size of the pool depends on cost of creating threads,the cost of maintenance of the pool and the workload of the system as a probability density function. The workload can be thought as the rate of receiving the client request to the web server. Experiments were evaluated under fore different probability density function which are uniform,exponential,density of Pareto and gamma density. Then optimal thread

pool size is decided for each density function. However, this study has similler assumption of above study.

There are a number of parameters that can be tuned in a thread pool such as number of threads in the pool, maximum number of threads allowed in a pool, maximum time interval that a thread will be idle waiting for a new task [7]. However, most of the studies are focused only to tune the size of thread pool size since that parameter affect most when evaluating the performance of web servers.

2.2.6 Importance of optimization

Optimization for web server is important for a business, so services able to deliver better performance using minimum resources. Today, larger portion of businesses are dependent on web service. Growing needs demand more resources that cost for the business. Although the word "optimization" shares the same root as "optimal", it is rare for the process of optimization to produce a truly optimal system. It is hard to define optimal system where system is optimized under certain quality metrics, which may be in contrast with other possible metrics. Therefor, the optimized system will typically only be optimal in one set of applications, not for all. One might reduce the CPU usage, while one is optimizing memory usage. Often there is no "one size fits all" design which works well in all cases, thus there are trade-offs to optimize the attributes of the greatest interest.

When come into web server optimization, it can be achieved under different ways. One way is to write clever code that use better data structures and algorithms, place code in perfect place in order not to waste CPU for IO. Programming patterns like asynchronous programming patterns provide some shortcuts for developers in odder to overcome blocking scenarios. Still, optimization is hard for every aspect because interrelation of trade-off between techniques [28, 29]. Another way is to optimize web servers where business logic is executing. Then developers able to focus more on business logic rather than spending time on code optimization.

2.3 Conclusion

It is clear that performance of a web server can be depended on the type of operation that is carried out by the web server. Moreover, IO features affect the performance of web servers and some web server architectures perform well for certain use cases. Also, careful tuning of web servers was able to achieve significant performance improvement. Furthermore, thread pool tuning is dominant where optimizing the performance of server architectures. It is understandable why analytical models are sometimes not practical in certain situations. Finally ,no research could be found that program classification based on how programming features affect the performance of web servers.

Chapter 3

Design

This chapter undergoes the design steps that are taken in each stage of the research. Initially this research is started as **Compile time server architectures for Ballerina**. After analyzing the results of the initial set of experiments, the focus area is narrowed down to finding optimal thread pool size for different programming features. A machine learning approach is used to predict the thread pool size for a given set of programming features. Research questions remain unchanged from the beginning since thread pools are a crucial part in web server architectures and it is a tuning parameter for further improving the performance of server architecture. Literature review indicates that performance of single web server architecture differs according to type of operations it executes. Type of the operations can be extracted by analyzing the source code. In simple terms different web server architecture perform differently for different types of programs. Then based on the results further improvements and fine-tuning to the architecture are carried out in order to identify what parameters are required to change for maximizing the performance for each type of program.

Formal research methodology is based on Design science research approach [30]. Design science research approach requires the creation of an innovative, purposeful artifact for a special problem domain. The artifact must be evaluated in order to ensure its utility for the specified problem. In order to form a novel research contribution, the artifact must either solve a problem that has not yet been solved, or provide a more effective solution. Both the construction and evaluation of the artifact must be done rigorously, and the results of the research presented effectively

both to technology-oriented and management-oriented audiences.

Design of the research is divided into four stages. First off existing Ballerina server architecture is carefully analysed and designed how new architectures can be derived from it. New architectures are implemented by modifying the source code. Secondly, identified parameters from the first phase are tuned in order to maximize the performance in server architectures. In this phase, the size of the thread pool is tuned. Thirdly, differentiating the IO intensive features in program and machine learning model is designed to predict size of the thread pool. Finally, AST is parsed for extracting the identified features to feed into the machine learning model that is designed.

This research consists of the following areas. Below sections span the design steps considered answering the research questions mentioned in section .

Before diving into the design steps, it is important to specify the internal architecture of Ballerina run time. Next section briefly explains the internals of the Ballerina.

3.1 Ballerina

This section provides an overview of current Ballerina architecture and explanation on how this research expects to use network awareness features in Ballerina language.

In Ballerina connector calls that basically perform IO operations are explicitly presented in the source code as " - >". Abstract Syntax Tree (AST) parser extracts this information from the source to identify whether this call is a connector call and type of the connector call (HTTP, Database etc.).

Following is one example how the database calls are represented in ballerina code, note the -> symbol.

stream<record, error> resultStream = mysqlClient4 - > query(¡@untainted¿query);

Algorithm used for this operation is presented in section 3.2.5

3.1.1 Ballerina internal architecture

This section explains the main components of Ballerina run time.

Netty Layer

This layer is implemented using a library called Netty [31]. Primary task of the Netty layer is to listen for Http client connections and manage the session. It has its own thread pools to handle those connections. In original ballerina architecture this layer simply handed over the incoming connection to the ballerina scheduler. Then the scheduler invokes relevant tasks for the client and returns the results to Netty layer again for submitting back to the client. Inside the Netty layer three is two types of threads.(1) **Boss threads** — Each bonded port has its own boss thread. For example, if the server listens on ports 80 and 443, then there are two boss threads. Main functions of these threads are accepting the client requests. (2) **Worker threads** — Boss thread passes the accepted request to the worker thread. Worker thread then carries out the execution of the task. In Ballerina, these worker threads pass the request to Ballerina scheduler.

Scheduler

Ballerina scheduler is the main component for every program/operation which executes. The Ballerina scheduler executes the client request using its own thread pool. The default thread pool size is two times the available processor count of the operating system. All the tasks which need to be executed are held in a queue. Then threads in the pool access the queue and execute the task. This is where major turnings are happening at the final stage of the research.

Resource function

In ballerina each API endpoint can be implemented as a resource function. That function includes all the implementation required to fulfill the task that client is requested. More details about resource function is stated in Chapter 4. In this research each resource function is considered as programs.

Figure 3.1 shows high level view of Ballerina architecture.

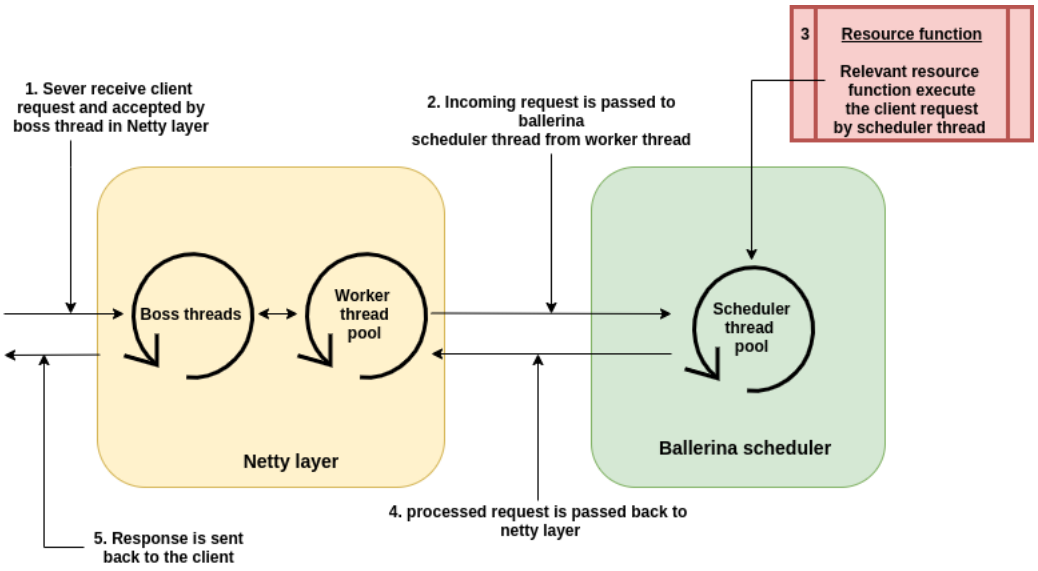

Figure 3.1: Ballerina's internal architecture

3.2 Design steps

The initial idea is coupled with the bound type of the program (CPU bound and IO bound). Objectives were, (1) for a given Ballerina program identify its bound type, then (2) switch the server architecture which best suits the bound type. To determine the best server architecture, experiments are conducted. Following steps are taken to conduct the experiments. Below process is iterative.

- Implement server architectures
- Use existing benchmark programs and implement new programs as necessary which have CPU intensive task and IO intensive tasks.
- Examine results.
- Debug and fine tune architectures based on results.

3.2.1 Testing process

Each testing program is hosted as HTTP endpoints. Calling that endpoint invokes the relevant program. Jmeter act as a HTTP client. Typical web server able to process concurrent HTTP requests. Jmeter can be configured to model this behavior. Jmeter continuously calls the given endpoint for certain period under

given configurations such as concurrency level. Then the following metrics are recorded,

- Average latency — client HTTP requests are continuously sent to the web server and measure response time of each request. Then average is calculated.
- Standard deviation of latency — Standard deviation of above latency results.
- Throughput — Number of successful responses received per second.
- Error rate - Number of erroneous request as percentage of all request sent.
- Median, 75th percentile, 99th percentile of latency results.

Then the performance is evaluated using above metrics. As an example when average latency is low and throughput is high for a given endpoint, that architecture's performance is good relative to others.

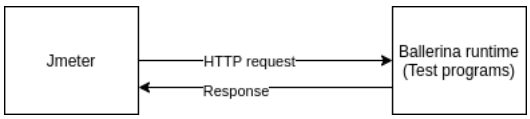

Figure 3.2: Jmeter and Ballerina run time

3.2.2 Phase I - Implement and evaluate performance of different server architectures

In this phase following server architectures are implemented.

- Original Ballerina architecture
- Netty OIO
- Removed Ballerina scheduler thread pool
- Changing Ballerina scheduler thread pool size

The goal of implementing Netty OIO is to have similar architecture of thread per connection model. In this Netty's worker thread pool is configured to spawn new thread per each client request. These threads block the execution until the

task is finished. However, the Ballerina scheduler continues the execution in non-blocking fashion.

The expectation of removing the Ballerina scheduler is to inspect the behavior and performance when one thread pool is removed. The Netty layer's worker thread will carry out the execution of client tasks instead of the Ballerina scheduler thread. When there are multiple thread pools, it adds some overhead to the system. Expected behavior of removing thread pools is to improve the performance.

Despite that, the purpose of having multiple thread pools is to avoid starvation where a task is keeping thread pool busy so that another task is waiting and not getting a chance to execute for some time. Therefore that task is starved. instead of having a single thread pool for all functions,if other services have their own thread pool, then it is assured of having a certain number of threads at its disposal, hence it's not as sensitive to demands made by other services. The idea is when there are multiple thread pools, while one partition of the application is busy, but other parts of the application continue to work normally regardless of the rest of the system. This will add more stable characteristics to the system when the load is high.

Finally, some variation to the number of threads in the Ballerina scheduler is performed. **Default thread pool size of Ballerina scheduler** is $\mathbf{2 \times Number\ of\ CPU\ core}$ Size of the thread pool is increased by $\mathbf{2 \times Number\ of\ CPU\ cores}$ and $\mathbf{4 \times Number\ of\ CPU\ cores}$. In Ballerina connector calls such as database calls block the execution thread. Increasing the number of threads can be beneficial in such situations because while other threads are blocking there are more threads to handle other tasks.

For each architectural change, a number of benchmark programs were run which consist of CPU intensive and IO intensive features. After conducting experiments with above architectures, following conclusions were made,

- Netty OIO performance is worse in every situation — thus this architecture was no longer considered
- Removing Ballerina scheduler performs well for both IO and CPU intensive programs.

Results and explanations are discussed in-depth in Chapter 5. Based on these results, experiments were conducted with changing the size of the thread pool in the Ballerina scheduler.

Test programs

Following programs were implemented to evaluate the performance of each architectural change,

- Check whether given number is prime - 3 primes are checked as prime small (521), prime medium(7919) and prime large(1000003)
- Merge sort — 1000 random numbers are sorted
- Read File from disk
- Database test - Execute 'select' query on mysql database.

Above programs cover the CPU intensive cases and IO intensive cases.

Sample result of a single experiment is shown in figure 3.3. Four metrics (throughput, slandered divination, mean — average latency, 99th percentile) are shown in the plot. X-axis represents the concurrent users. Y-axis represents the relevant metric.

Then each metric is evaluated against each test program and each architectural change mentioned above.

Architectural changes to existing Ballerina run time involved extensive debugging because bugs in the implementation may incur wrong results. Therefore, this phase is conducted in an iterative manner to verify the results. At the end of this stage no architecture was performed well for IO use cases rather than CPU intensive cases and vice versa. Although changing thread pool size in the Ballerina scheduler started to show some significant results. Increasing thread pool size was affected differently in IO use cases. Then the phase 2 is designed to analyze the variation of thread pool comprehensively.

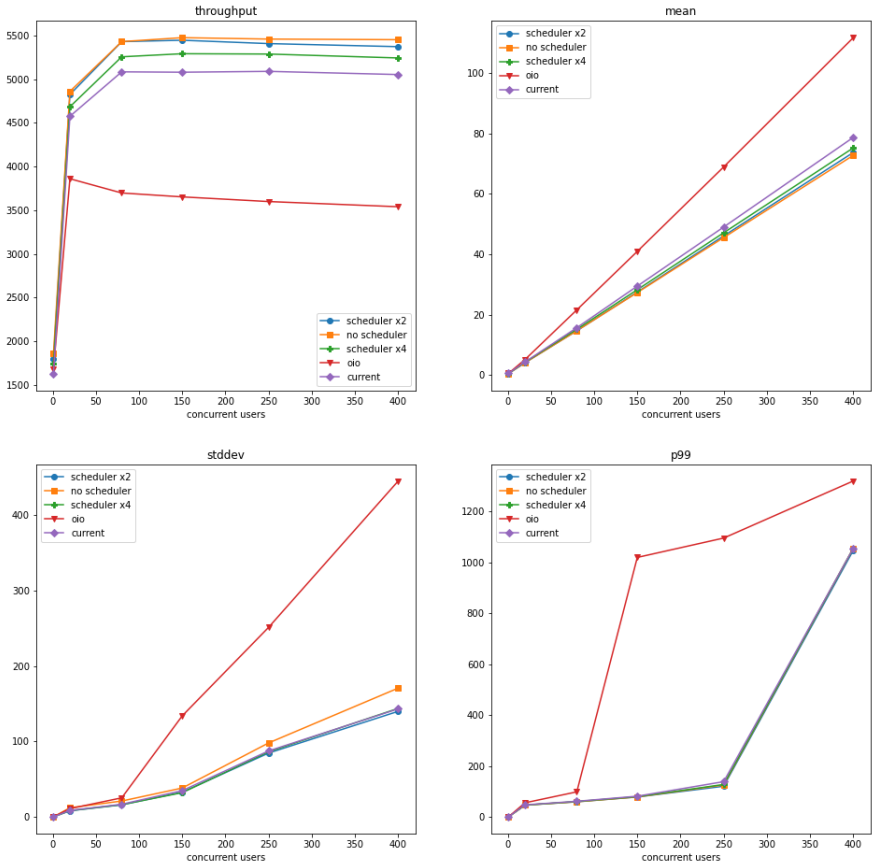

Figure 3.3: Sample experiment result

3.2.3 Phase II - Tuning thread pool size

Thread pool tuning can be approached in two ways, **(1) White box system** — This requires complex mathematical modeling with queuing theories [7]. Also, assumptions made at the beginning make it difficult to apply those modeling techniques for practical scenarios. **(2) Black box system** — In this approach the thread pool system is considered as a black box where in depth analysis of the thread pool system is not performed. Experiments are performed heuristically by changing parameters such as size of the thread pool. Figure 3.4 shows the boundary of the black box.

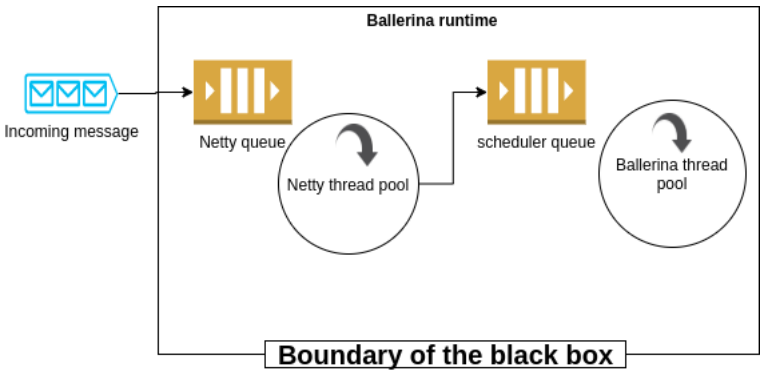

Figure 3.4: Boundary of the black box

This phase is preceded by considering the system as black box. This model allows executing different programs under different sizes of thread pool. Following the first phase of experiments, it is identified that changing thread pool size is affected differently for programs that consist of IO features. In the previous phase, experiments are conducted under different concurrency levels, but in this phase fixed concurrency level is selected. Concurrency level was selected where a number near the point in the throughput curve which curve is starting to get flatten.

New sets of programs are implemented referring to multiple sources [32, 33] in order to support programs that have rich IO call variations. Addition to the database calls in previous phase programs which have remote HTTP calls, GRPC calls and loops that contain these features are implemented. Subsection 3.2.4 provide more information. CPU intensive programs are no longer considered since it is difficult to extract CPU intensive operations directly by parsing AST. More IO oriented approach is used where AST parser able to extract this information

Table 3.1: Program vs. Thread pool size - Latency results

Program	Pool size					
	1	2	3	...	19	...
Program 1	Ave. latency	...	...	...	...	
Program 2	...	...	...	...	...	
Program 3	...	...	...	...	...	
...	...	...	...	...	...	
Program 50	...	...	...	...	...	
...	...	...	...	...	...	

directly from the source code.

Experimental design

As a primary metric of evaluation average latency is used. Then each program is run under different thread pool sizes. Afterward metrics are recorded. Table 3.1 shows summary of the results. Empty cells represent the average latency. Minimum latency values provide the optimal thread pool size for a given program. Instead of declaring thread pool size as a multiplier of number of CPU, explicit numbering is used for in-depth analysis of every thread pool size from 1 to 22. This range is selected because latency is always getting higher when increasing the number further.

3.2.4 Phase III - Building machine learning model to predict optimal thread pool size for given program

At first two models are considered addition to predicting optimal thread pool. One model directly predict the latency with given program features and given thread pool size. This model can be expressed as following function.

$$Latency\,(ms) = f(\,Program\,features,\,Threadpool\,size\,)$$

Weakness of this model is it heavily depend on nature of IO call. As an example latency is dependent on where the database is hosted if the given program has database call. Also, latency is affected by the network connection also. Then the

Table 3.2: Programming features selected for machine learning model

	Feature
F1	Number of HTTP connector calls
F2	Number of Database connector calls
F3	Number of non-blocking gRPC connector calls
F4	Number of loops containing HTTP connector calls
F5	Number of loops containing Database connector calls
F6	Number of loops containing non-blocking gRPC connector calls

generalization of result is very difficult.

The next model predict optimal thread pool size for the given program. That model can be expressed as follows. That is the hypothesis that try to resolve using machine learning model.

$$Optimal\ thread\ pool\ size = f(\ Program\ features)$$

In order to build this model it is required find the minimum thread pool value from the result obtained by Design phase II. This step can be depicted as in figure 3.5

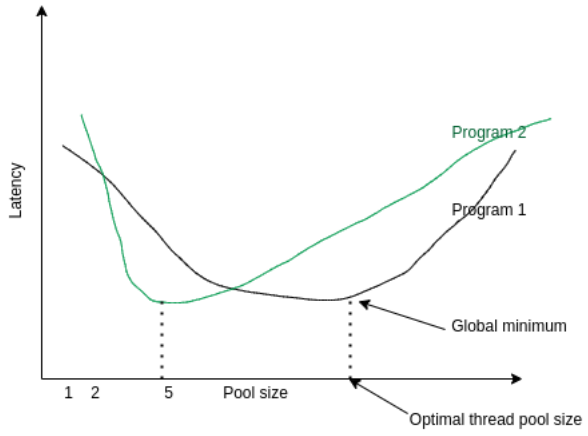

Figure 3.5: Obtaining optimal thread pool size

After some feature engineering steps, Features in table 3.2 are selected to train the machine learning model. Hyper parameters of each machine learning model is tuned for better accuracy. Implementaion of hyper parameter tuning is presented in chapter 4.

Then a training data set is created where input features are above programming features and output is optimal thread pool size. Table 3.3 shows the shape of the

Table 3.3: Data frame shape

Program	Features				Thread pool size
	F 1	F2	..	Fn	
Program 1					
...					

data frame.

Then following machine learning models are selected and evaluated the accuracy of each model.

- XGBoost
- Support Vector Machine
- Decision Tree
- Random Forest

For regression models Mean Absolute Percentage Error (MAPE) and Mean Squred Error (MSE) are evaluated. For classification models F1 score and accuracy are evaluated.

The problem is originally a regression problem because output of the model is a number. As this research provide a proof of concept, the problem is formulated as classification model as well. Mapping the problem into a classification is possible because, in the result of optimal thread pool sizes, clear clusters can be recognized. Furthermore, distribution of optimal thread pool is not uniform and only has several values.

3.2.5 Phase IV - Parsing Ballerina AST to obtain features

First off, it is better have some idea about what is Abstract Syntax Tree (AST). It is a tree representation of stricture of the source code.AST does not include all the details such as semicolon, parenthesis etc. AST is used in semantic analysis of a code during compilation process. Traversal of the AST verify the correctness of the program against rules provided for that language. Moreover, information of

the code can be extracted by traversing the AST. Example of an AST is shown in figure 3.7 for code segment shown in figure 3.6.

```
while(b!=0){
    if a>b{
        stream<record{}, error>
 resultStream = mysqlClient4->query(<@untainted>query);
    }
    return 10;
}
```

Figure 3.6: Ballerina code

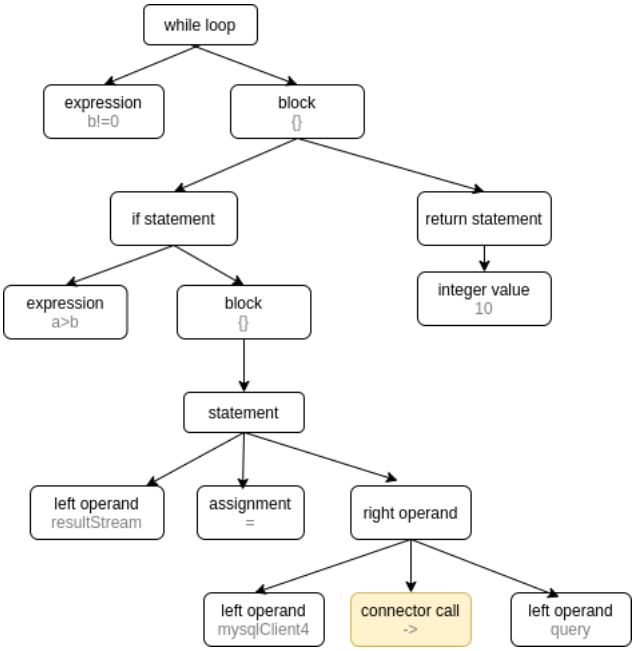

Figure 3.7: Example AST

In this phase a module is designed to parse Ballerina source code to feed the features in to machine learning model. Manual feature extraction is erroneous and harder when programs are large. First this module accepts the Ballerina source code and output the array of features which is fed to machine learning model. This is where Ballerina specific features are used that other language are lacks. Services are first order functions in the Ballerina language. This able to decide that the given program contain the web service. Then the subsequent parsing are happened inside these service definitions. Also network calls are part of the language. In the other languages they are mostly library calls which is difficult to identify the

given line of code contain the network call. In Ballerina this is called as connector call. Recalling section 3.1, other languages provide network operations as library functions. If we try to extract such features from other languages, we need to know exactly this function call do an IO operation. In Ballerina, it is possible because connector calls can be differentiated naively. Algorithm 1 is implemented to extract features.

Finally, combing the above design steps, overall design of the research is shown in figure 1.1.

3.3 Limitations

There are several limitations can be reorganized in the design. In design phase II, fixed concurrency level is chosen. It is not explored how different concurrency levels are affected to the results. Another limitation is that hardware variables like number of CPU, memory are not considered when constructing the hypothesis. Finally, when constructing the machine learning model it is only considered IO features. CPU extensive programs cannot be identified by parsing the source code.

Algorithm 1: Extract features using Ballerina AST

```
Input: Source code (Ballerina.toml)
Output: Feature Array
Start parsing;
while End program do
    Traverse node;
    if Right Arrow Found then
        if Left operand is Kind(Database call) then
            while Backtrack do
                if Loop start found then
                    Update feature array (Database call inside loop + 1);
                    Stop backtracking and continue parsing;
                end
                Update feature array (Direct Database call + 1);
                Continue parsing;
            end
        end
        if left operand is Kind(Http call) then
            while Backtrack do
                if Loop start found then
                    Update feature array (HTTP call inside loop + 1);
                    Stop backtracking and continue parsing;
                end
                Update feature array (Direct HTTP call + 1);
                Continue parsing;
            end
        end
        if left operand is Kind(GRPC-non-blocking call) then
            while Backtrack do
                if Loop start found then
                    Update feature array (GRPC call inside loop + 1);
                    Stop backtracking and continue parsing;
                end
                Update feature array (Direct GRPC call + 1);
                Continue parsing;
            end
        end
    end
end
```

Chapter 4

Implementation

This chapter explain the steps taken to implement the research design and how it is addressed the issues occurred during implementation. Implementation can be divided as following sub topics.

- Implementing server architectures in Ballerina
- Implementing test cases for performance evaluation
- Designing automated pipeline to obtain results
- Creating Machine learning models to classify programs

4.1 Implementing server architectures in Ballerina

Ballerina source code written Java and Ballerina itself.

This involves deeper analysis existing Ballerina run time. Ballerina has three main non-blocking thread pools. One thread pool (Worker thread) lies in the scheduler. Other two thread pools lies in the Netty layer. Ballerina's internal architecture is explained in-detail in chapter 3. Initial step is making Netty thread pool to have blocking IO. The Netty layer integrated as separate library (trasport-http)[34]. Following code fragment shows the implemented changes. *workerGroup = new OioEventLoopGroup(200000);* specify the use of blocking thread in the tread pool and allowing up to 20,000 threads in the thread pool. This thread pool

hand over the client workload to Ballerina Scheduler. This class is referred in the Ballerina run-time.

```
1
2  public class DefaultHttpWsConnectorFactory implements
    HttpWsConnectorFactory {
3
4  private final EventLoopGroup bossGroup;
5  private final EventLoopGroup workerGroup;
6  private final EventLoopGroup clientGroup;
7  private EventExecutorGroup pipeliningGroup;
8
9  private final ChannelGroup allChannels = new DefaultChannelGroup
    (GlobalEventExecutor.INSTANCE);
10
11 public DefaultHttpWsConnectorFactory(int serverSocketThreads,
    int childSocketThreads, int clientThreads) {
12 bossGroup = new OioEventLoopGroup(4);
13 workerGroup = new OioEventLoopGroup(200000);
14 clientGroup = new NioEventLoopGroup(clientThreads);
15 }
16
17 ...
```

The next architectural change is **remove Ballerina scheduler** and use only Netty threads to complete client request. Then the Ballerina scheduler is not involved in executing client request. Normal procedure is incoming request is handed over to scheduler thread pool. Then scheduler thread pool complete the execution using it's own threads and again handed over to Netty layer to send the client response.

This implementation should be carried out carefully. If the implementation is erroneous or having bugs, it would produce wrong results. Thus extensive debugging was used while implementation .This was difficult due to tight coupling of Netty layer and Ballerina scheduler. In order to implement that I created separate **Task Executor** as *MyExecutor*. Execution of client request is defined in that class. The *submit()* method accept the incoming request and execute the instructions using incoming thread which is proceeded from Netty thread pool.

```
1
2 public class BallerinaHTTPConnectorListener implements
    HttpConnectorListener {
3 ...
4
5 @Override
6 public void onMessage(HttpCarbonMessage inboundMessage) {
7
8 ...
9
10 MyExecutor.submit( service, httpResource.getName(), callback,
    properties, signatureParams);
11
12 ...
```

Addition to major architectural changes, fine tuning to implemented architecture and existing architecture is carried out continuously. Those fine tunes assisted to narrow down to research to thread pool optimization. In the final phase existing Ballerina architecture is selected and tested with different thread pool numbers. Number of threads in the thread pool is set through environment variable. Following code snippet shows that.

```
1 public class Scheduler {
2  ...
3
4 // Setting Ballerina thread pool size
5 private static String poolSizeConf = System.getenv(BLangConstants.
    BALLERINA_MAX_POOL_SIZE_ENV_VAR);
6
7 ...
8
9 public Scheduler(boolean immortal) {
10   try {
11     if (poolSizeConf != null) {
12         poolSize = Integer.parseInt(poolSizeConf);
13     }
14   } catch (Throwable t) {
15
16     // Log and continue with default
```

```
17      err.println("ballerina: error occurred in scheduler while
        reading system variable:" +
18      BLangConstants.BALLERINA_MAX_POOL_SIZE_ENV_VAR + ", " + t.
        getMessage());
19    }
20    this.numThreads = poolSize;
21    this.immortal = immortal;
22
23    console.println("Pool size: "+poolSize);
24  }
```

4.2 Implementing test cases for performance evaluation

Test cases were implemented in Ballerina language. This implementation is very crucial since all the results is depend on the designing of test programs. In the initial phases all the test programs are designed to have IO calls and CPU intensive instructions. Some of test cases were designed by Ballerina team [32] which was originally designed to evaluate performance of Ballerina run time. Those tests included external service calls (http requests). This research able to re-use them and reproduce the results.

Since Ballerina is service oriented language implementation is well structured. In Ballerina a service is packed to single unit. Following example represent the service called *myservice*. That service is exposed outside via port number 9090. All the relevant implementation to this service goes inside the braces. There can be multiple services exposed via different ports. Structure of this is well suited with micro-service architecture.

```
1  service myservice on new http:Listener(9090) {
2
3    }
```

Inside a service it can be implemented resource function. Single resource function can be considered as single endpoint in RESTful API. Following resource function represent the endpoint *http://hostname:9090/book/someBookId*

```
1
2  service myservice on new http:Listener(9090) {
3
4
5  @http:ResourceConfig {
6  methods: ["GET"], path: "/book/{bookId}"
7  }
8  resource function getById(http:Caller caller, http:Request req,
       string bookId) {
9    json? payload = booksMap[bookId];
10   http:Response response = new;
11   if (payload == null) {
12     response.statusCode = 404;
13     payload = "Item Not Found";
14   }
15   response.setJsonPayload(untaint payload);
16   var result = caller->respond(response);
17   if (result is error) {
18   log:printError("Error sending response", err = result);
19   }
20 }
```

Similar to above resource funcnion all the test cases are enclosed as resource functions. Then the test cases are invoked by http request. Following code segment shows one test case used for testing. It includes single blocking IO call (Databse call).

```
1  resource function db_select(http:Caller caller, http:Request
       request) returns error? {
2    http:Response response = new;
3    var params = request.getQueryParams();
4    var id = <string>params.get("id")[0];//<string>params.id
5    var query = "SELECT * FROM emp where id = "+id;
6
7    stream<record{}, error> resultStream = mysqlClient4->query(<
       @untainted>query);
8
9    record {|record {} value;|}|error? result = resultStream.next();
10
11   error? e = resultStream.close();
```

```
12
13  response.setTextPayload(<@untainted> io:sprintf("%s", result));
14  check caller->respond(response);
15
16  }
```

100 of programs similar to above are implemented to evalute test cases.

4.3 Designing automated pipeline to obtain results

This section explains how the results were obtained and what are the actions taken into account to ensure the accuracy.

Proper testing environment is very important since results may inaccurate without proper isolation. Testing environments are isolated with used Docker [35] containers. It minimizes the interference occurred by other processes. Also it limits the resource usage by specifying the upper bound.

Performance metrics are obtained primarily by Jmeter [36]. In order to minimize the instrumentation errors results are validated with load testing tool called Tsung [37].

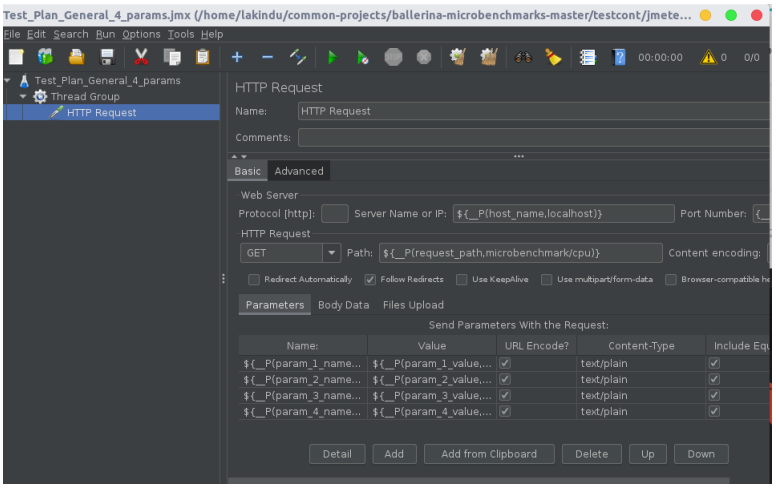

Figure 4.1: Jmeter interface

Jmeter have both command line and graphical user interface. Tsung configurations only available with command line and XML files. Configuring tests is easier

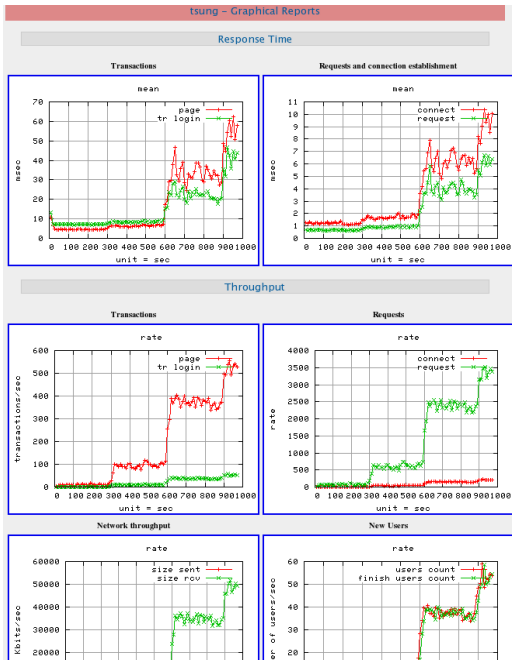

Figure 4.2: Tsung stat report

in Jmeter than Tsung. Thus Tsung is only used to validate the results in early phases. Several tests are executed using both tools and obtained the latency results to validate the tools. Some results are shown in table 4.1. Average variance is *0.29 ms*.

Test	Jmeter	Tsung	Description
test 1	27.42ms	24.71ms	Database call - concurrency level 80
test 2	57.31ms	57.22ms	External http call - concurrency level 80
test 3	44.97ms	45.18ms	Merger sort - concurrency level 60
test 4	34.76ms	34.42ms	database call - concurrency level 100
test 5	28.44ms	28.32ms	Prime number calculation - concurrency level 80

Table 4.1: Latency results comparison using Jmeter and Tsung

Testing pipeline is consited of two parts, which are,

1. Ballerina run time where tests were implemented

2. Jmeter/Tsung clients to measure the performance metrics

Addition to above, database server, external HTTP server and grep server is designed to run tests. Figure 4.3 depict the architecture of testing environment. 3 Virtual Machine (VM) that are hosted in Google cloud is used. VM that running Ballerina run time and Jmeter client has 6 core CPU with 12 GB of memory. Other two VM are 2 core CPU with 4 GB of memory. In order to eliminate bottlenecks of external services CPU usage is monitored to keep the CPU usage under 80%. Hence results are only depend on Ballerina run time. Since Ballerina run time, Jmeter client and Database server is hosted in same VM using docker Maximum CPU usage is limited to 2 cores for Jmeter and Ballerina run time. Database server is allocated with 1.2 CPU cores. With maximum concurrency level Database server use only 83% CPU.

Continuous monitoring ensured no bottleneck due to resource limitation. Testing pipeline is automated since duration of test is very long. Some tests were running for days and failing tests caused running them again. Implementation of pipeline is included in code listings.

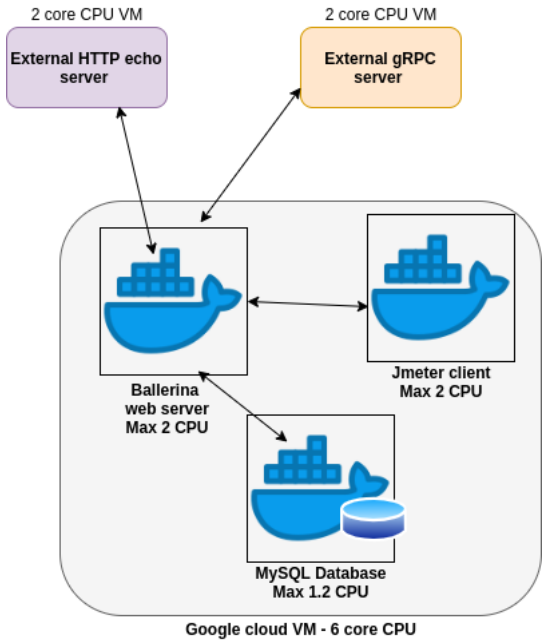

Figure 4.3: Testing environment

4.4 Creating Machine learning models to classify programs

Final implementation of this research is build machine learning model to predict architecture based on programming features. Based on early results, the research is narrowed down to find the optimal thread pool size instead of predicting architecture. The final model predict the optimal thread pool size for the given program. Two approaches were used to obtain results. One is making the problem as classification problem and other is making it as regression problem. Then relevant metrics are compared to evaluate the best approach. Following machine learning models are used,

- XGBoost
- Support Vector Machine
- Decision Tree
- Random Forest

Following code shows implementation of XGBoost regression model.

```
1
2 from numpy import loadtxt
3 from xgboost import XGBRegressor
4 from sklearn.model_selection import train_test_split
5 from sklearn.metrics import accuracy_score
6 from sklearn.metrics import mean_absolute_percentage_error
7 from sklearn.metrics import mean_squared_error
8
9 def cross_validation_r(model,X,Y):
10
11   cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1)
12
13   scores = cross_validate(model, X, Y, cv=cv , scoring=['
     neg_mean_squared_error','neg_mean_absolute_percentage_error'])
14
15
```

```
  cv_mape=scores['test_neg_mean_absolute_percentage_error']
  cv_mse=scores['test_neg_mean_squared_error']

  print("MAPEs: ",cv_mape)
  print("MSEs: ",cv_mse)

  print("MAPE: %0.2f%% with a standard deviation of %0.2f" % (abs(
    cv_mape.mean()*100), cv_mape.std()))
  print("MSE: %0.2f  with a standard deviation of %0.2f" % (abs(
    cv_mse.mean()), cv_mse.std()))

# load data
dataset = loadtxt('g_dataset_class_2.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:5]
Y = dataset[:,6]
# split data into train and test sets
seed = 6
test_size = 0.3

X_train, X_test, y_train, y_test = train_test_split(X, Y,
    test_size=test_size, random_state=seed)
# fit model no training data
model = XGBRegressor()

model.fit(X_train, y_train)
# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]


# Cross validation
print("# cross validated evaluations")
print(cross_validation_r(model,X,Y))

# cross validated evaluations
MAPE: 7.92% with a standard deviation of 0.01
MSE: 1.46  with a standard deviation of 0.15
```

In order to tune hyper parameters of the machine learning model, following implementation is used. Then the machine learning model is retrained with optimal parameters rather than default parameters.

```
1
2 def hyperparameter_tuning(tr_features, tr_labels):
3   # Number of trees in random forest
4   n_estimators = [int(x) for x in np.linspace(start = 200, stop =
     2000, num = 1000)]
5   # Number of features to consider at every split
6   max_features = ['auto', 'sqrt']
7   # Maximum number of levels in tree
8   max_depth = [int(x) for x in np.linspace(5, 150, num = 50)]
9   max_depth.append(None)
10  # Minimum number of samples required to split a node
11  min_samples_split = [x for x in range(1,51)]
12  # Minimum number of samples required at each leaf node
13  min_samples_leaf = [x for x in range(1,51)]
14  # Method of selecting samples for training each tree
15  bootstrap = [True, False]
16  # Create the random grid
17  random_grid = {
18  'max_features': max_features,
19  'max_depth': max_depth,
20  'min_samples_split': min_samples_split,
21  'min_samples_leaf': min_samples_leaf,
22  }
23
24  rf_hyper = tree.DecisionTreeRegressor()
25  rf_random = model_selection.RandomizedSearchCV(estimator =
     rf_hyper, param_distributions = random_grid, n_iter = 100, cv =
      3, verbose=2, random_state=42, n_jobs = -1)
26  # Fit the random search model
27  rf_random.fit(tr_features, tr_labels)
28  return rf_random.best_params_
29
30  a =hyperparameter_tuning(X_train,y_train)
31  return a
```

Chapter 5

Results and Evaluation

This chapter compare the results obtained in each stage and how those results are supported to narrow down the research to find number of optimal thread pool for given program. In the early phases broader measurement metrics (average latency, throughput, Standard deviation, average latency, Median latency, 99th percentile of latency, error rate) were analyzed to compare architectures. Some metrics In later phases average latency (in milliseconds) is used as main metric to compare performance. Chapter 3 explains the these metrics in detail.

In the first phase following architectures were implemented,

- Ballerina current architecture
- Increasing Ballerina scheduler thread pool size into 2 times and 4 times
- Removing Ballerina Scheduler thread pool
- Changing Netty layer worker thread pool as thread per connection model (Netty OIO)

5.1 Results comparison of different architectures

Metrics such as error rate is not shown in the plots since those metrics are used to verify the validity of obtained results in intermediate steps. Performance tests were started with Current ballerina architecture, Removing ballerina scheduler thread pool and Netty OIO architecture. Since Netty OIO performance (High latency and

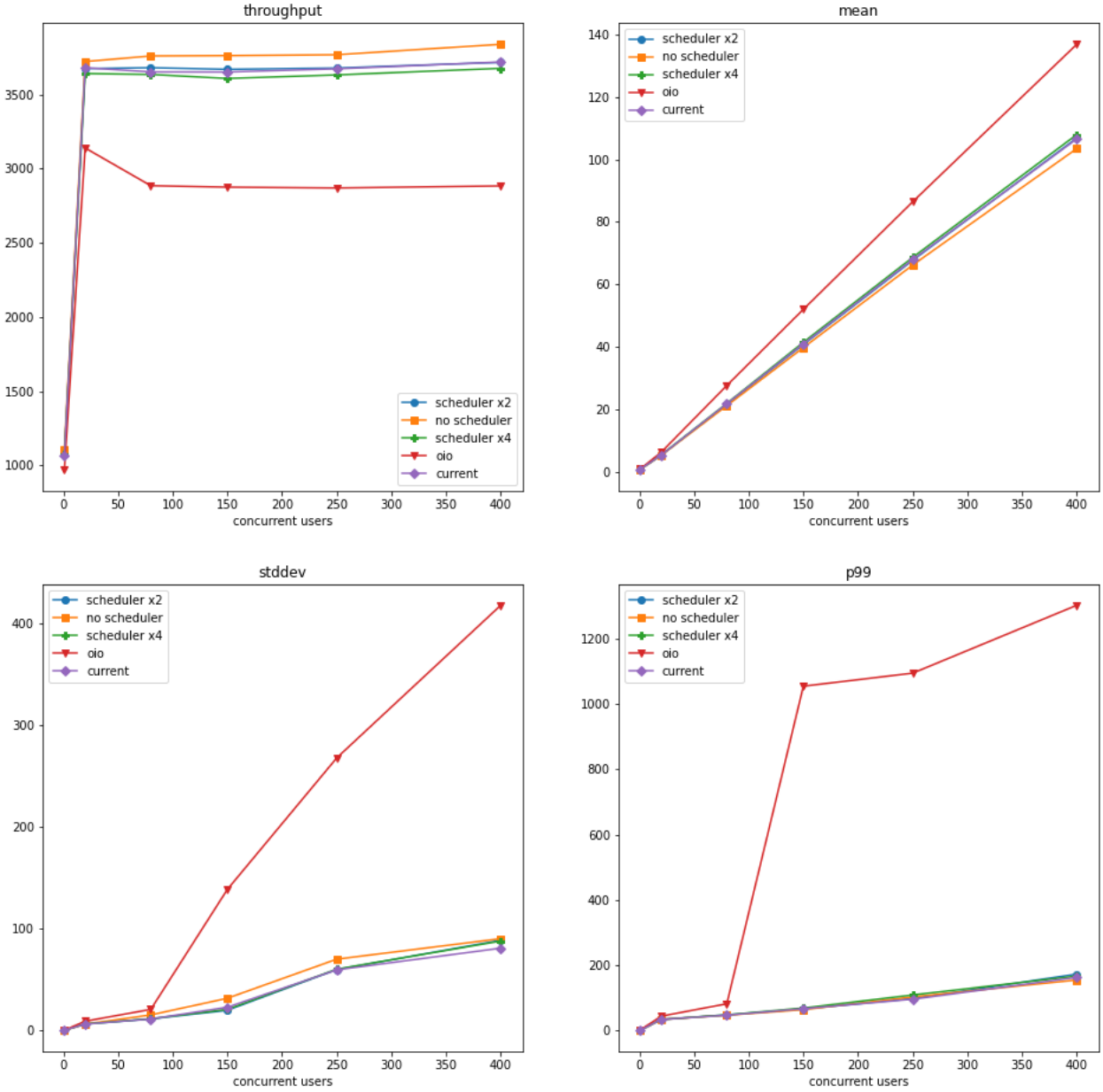

Figure 5.1: Throughput, Avg. Latency(Mean), Standard deviation(stddev) and 99th percentile of latency of Database call test for 1. Make twice the thread pool size of scheduler 2. Ballerina removing scheduler thread pool 3. Make 4 times the thread pool size of scheduler 4. Netty OIO architecture 5. Ballerina current architecture

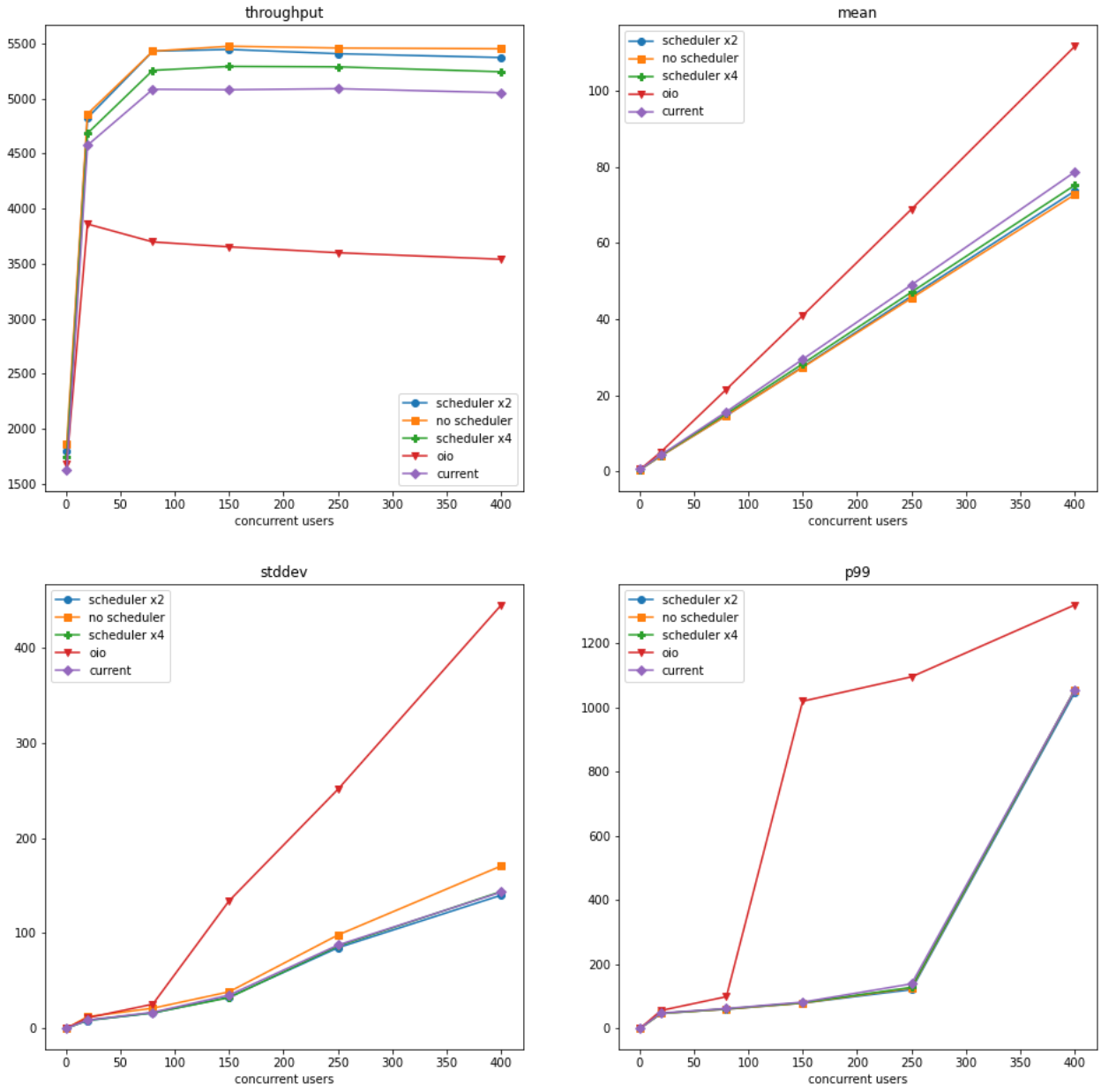

Figure 5.2: Throughput, Avg. Latency(Mean), Standard deviation(stddev) and 99th percentile of latency for Prime small test

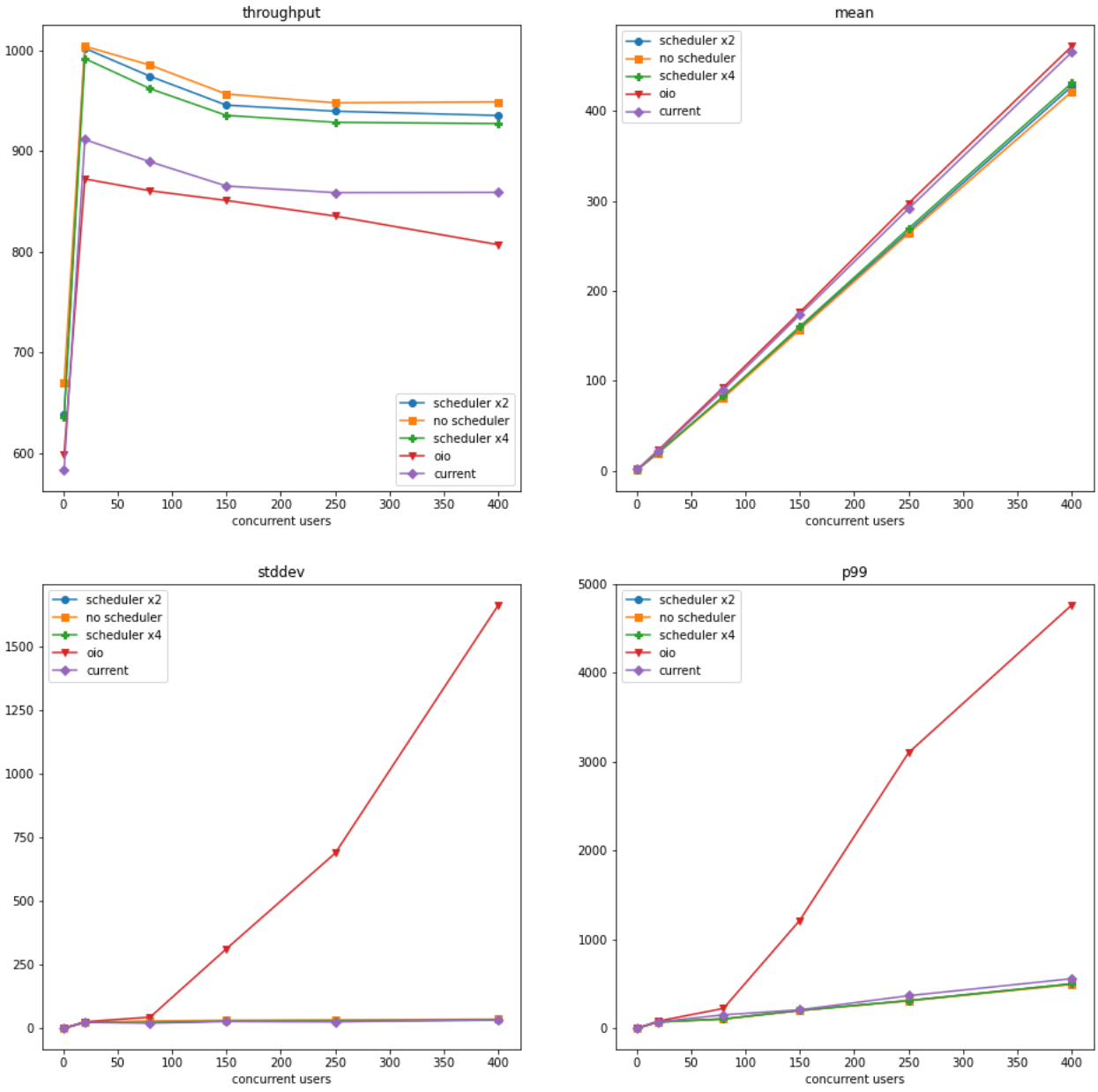

Figure 5.3: Throughput, Avg. Latency(Mean), Standard deviation(stddev) and 99th percentile of latency for Prime medium test

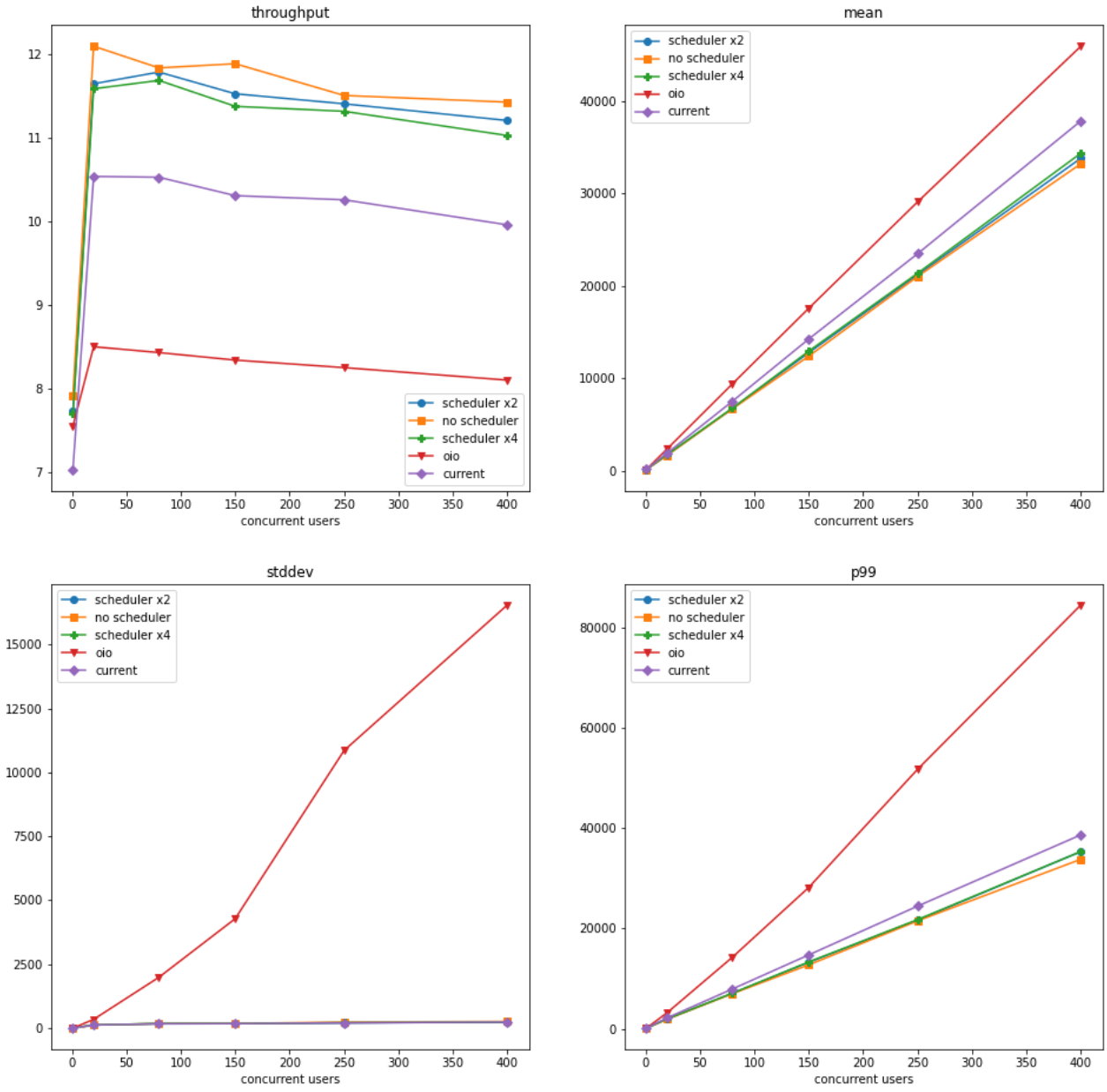

Figure 5.4: Throughput, Avg. Latency(Mean), Standard deviation(stddev) and 99th percentile of latency for prime large test

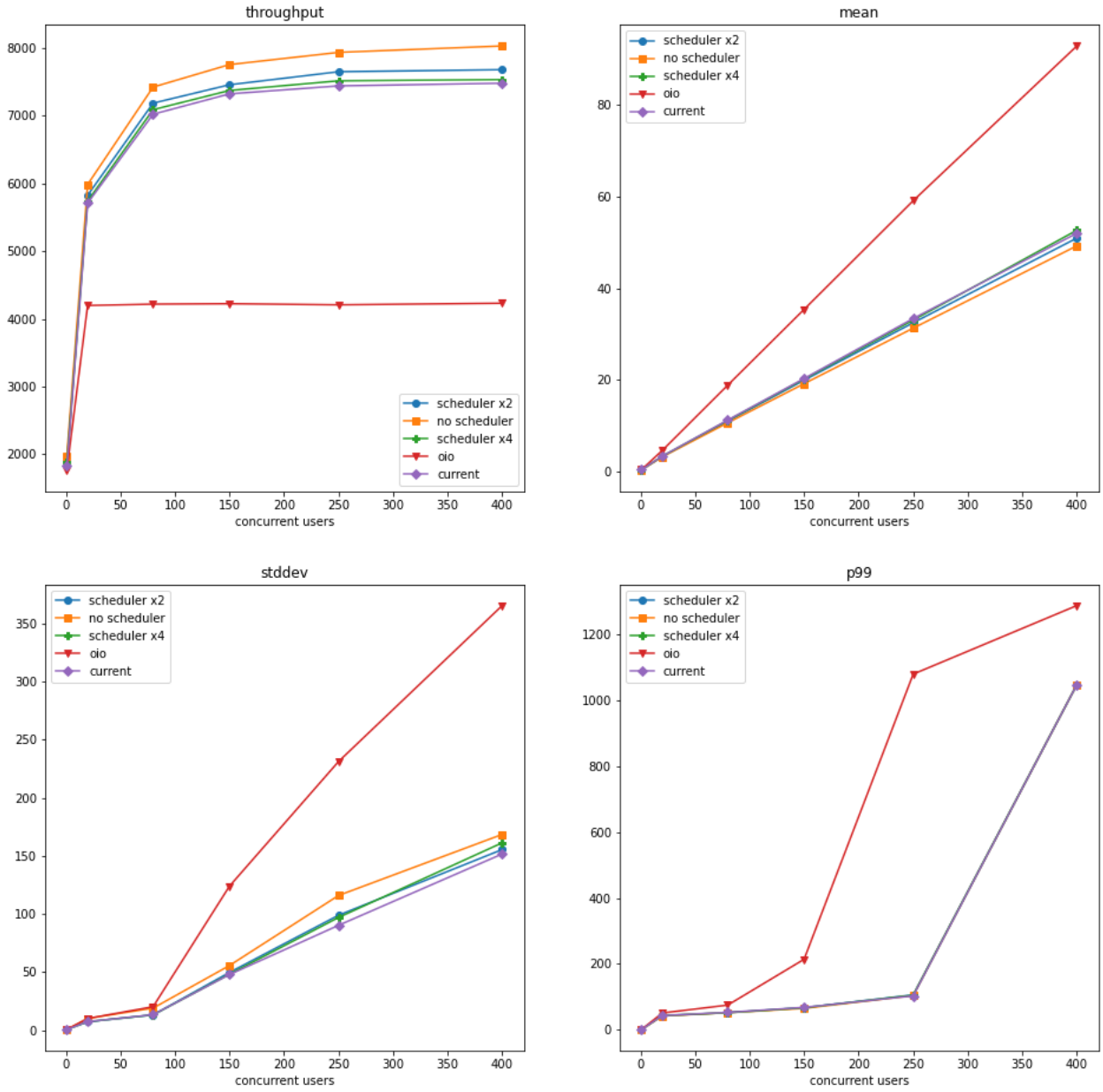

Figure 5.5: Throughput, Avg. Latency(Mean), Standard deviation(stddev) and 99th percentile of latency for file read test

low throughput) is lowest in every case, focus is moved to tread pools. Then the behavior is analyzed with adding more threads to scheduler thread pool. Figures 5.1,5.2,5.3,5.4,5.5, shows the throughput and latency results for Database, Prime small,Prime medium,Prime large and File read. In mentioned graphs it can be seen that OIO architecture has the lowest throughput, the highest latency and higher standard deviation among all the other architectural changes for almost every concurrency level. After initial tests OIO architecture was not considered for future tests.

Another significant indication that removing ballerina scheduler able to gain significant performance(Higher throughput and lower latency). That performance gain is independent form the program type. Thus,it was not possible to find distinct situations where a certain architecture would perform well for one program type while another architecture performs well for another program type. Although changing thread pool size affected differently in CPU bound(Prime test) and IO cases (Database test). Then the direction of server architecture tuning is focused on tread pool tuning. In this phase it is only changed two values of thread pool size. That is not sufficient to bring any conclusion.

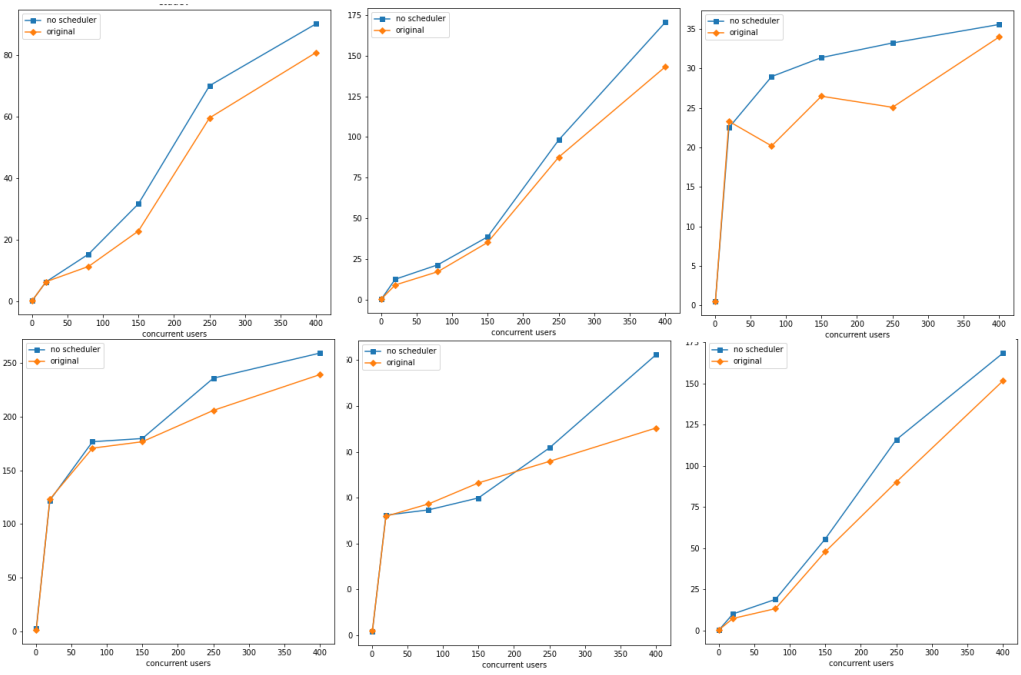

Figure 5.6: Standard deviation comparison of Database select test, Prime small test, Prime medium test, Prime large test, Prime medium test, Merge sort, File test

In order to evaluate the performance of sever architecture in depth standard deviation is another important metric. Although removing ballerina scheduler perform well by throughput and latency, standard deviation is higher than current ballerina architecture for every program type. Since OIO architecture has greater standard deviations, values are not clear in the above figures. To analyze more clearly Figure 5.6 shows standard deviation values for current architecture and removed scheduler architecture. It is clear that having Ballerina's scheduler thread pool provide more stable results. Also, it is noticeable that increasing thread pool size of the Ballerina scheduler thread pool able to decrease the average latency closer as removed Ballerina scheduler thread pool. There is a trade off between stability and latency when scheduler thread pool is removed. In order to maintain stability it is chosen to keep ballerina scheduler thread pool.

Test programs were designed to have CPU bound and IO bound features in order to evaluate which server architecture is better for IO bound programs and which architecture is better for CPU bound program. Results obtained in this phase does not able to differentiate performance of architecture based on program type. However, it is clear that changing the thread pool size affected differently in IO bound and CPU bound programs. When comparing the throughput of Database test case (Figure 5.1) and Prime small test case (Figure 5.2), changing thread size affected most in the prime small test. In Database test, increasing thread pool size doesn't change the throughput and latency values significantly. The reason behind that scenario can be explained as, since database call is a blocking call which blocks the execution thread, thread pool size variation (2 times and 4 times the default scheduler thread pool size) is not sufficient to gain significant performance different. Next experiments results are based on introducing more thread pool sizes to the ballerina scheduler considering results on this phase. Also, there is lack of programs based on IO features in the initial experiments even though there are many CPU intensive programs. Thus new IO programs are added to the experiments.

5.2 Results comparison of different thread pool size

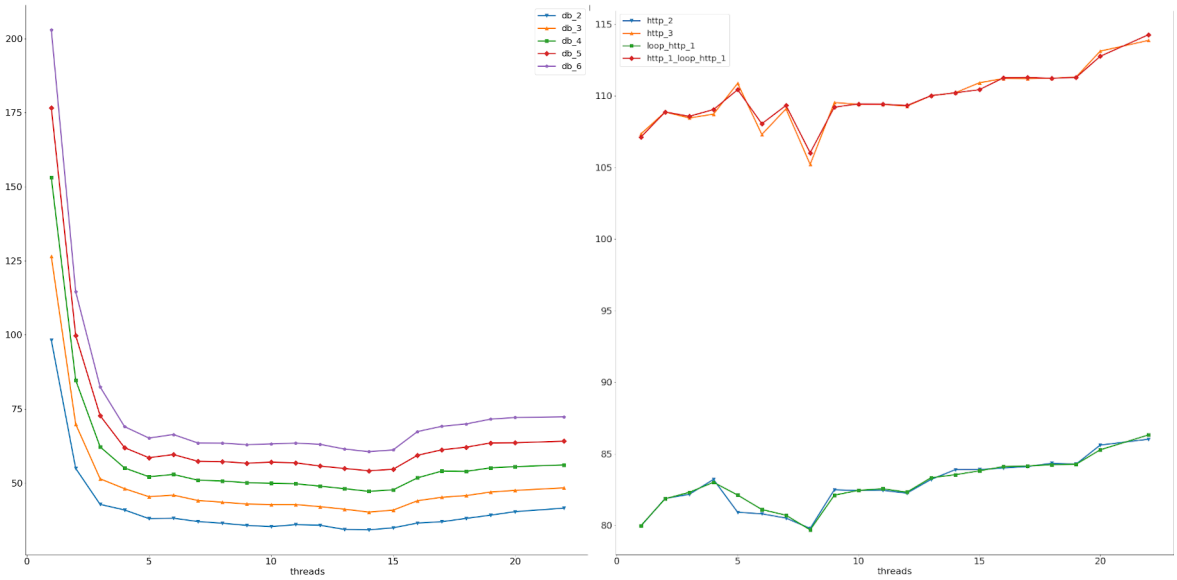

Figure 5.7: Left plot shows Thread pool size vs Average latency (millisecond) for number of database calls. **db_2** represent program has 2 database calls. Right plot shows Thread pool size vs Average latency (millisecond) for number of http calls and number of loops which contains http calls

Based on previous experiment set, performance is evaluated for different thread pool sizes. Programs which contains different type of IO features are analyzed. In previous experiment sets for IO features, there were only database call test. Programs with following IO features are evaluated,

- gRPC calls
- Database calls
- Http calls

Implemented programs contains above IO features. This experiment is aimed to obtain to get thread pool size which gives the minimum latency. More details of the design is explained in chapter 3. Results are taken from 82 programs.

Figure 5.7 and figure 5.8 shows some results for thread pool size of 1 to 22. This range is selected because minimum of the average latency is always laid between

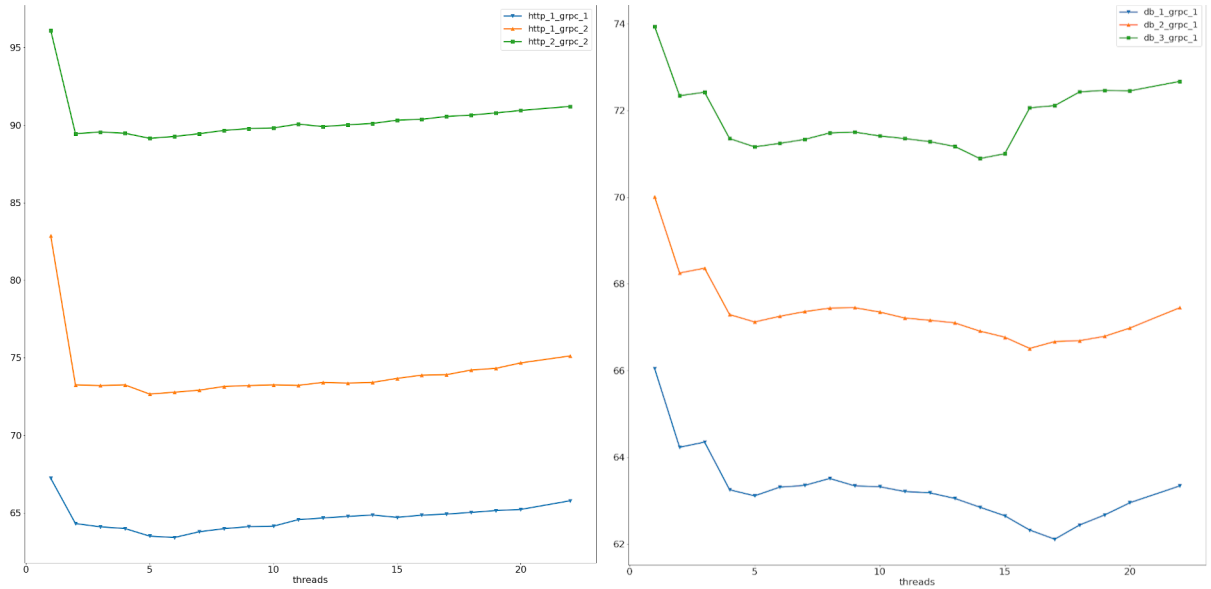

Figure 5.8: Left plot shows Thread pool size vs Average latency (millisecond) for number of database calls and grpc calls. **http_1_grpc_2** represent that program has 1 http call and 2 grpc calls. Right plot shows Thread pool size vs Average latency (millisecond) for number of database calls and number of grpc calls.

this range. In order to verify the selected range, experiments are conducted for higher number of thread pool sizes. Those results showed only increase of average latency.

From these experiments, thread pool size which gives minimum latency is obtained in order to feed the machine learning model.

5.3 Results comparison of machine learning models

This section presents the results of machine learning models. The machine learning model can be expressed as follows,

$$Optimal\ thread\ pool\ size = f(\ Program\ features)$$

After feature engineering steps, following features are selected to train the machine learning model.

- Number of HTTP connector calls

- Number of Database connector calls
- Number of non-blocking gRPC connector calls
- Whether each type of connector calls lies inside a loop
- Whether loops contain HTTP,Database or gRPCS calls (As separate features)

Features are obtained by parsing the AST of Ballerina programs. Input of the training data set is above programming features and output is the optimal thread pool size which gives the minimum latency. Table 5.1 shows the fragment of the data set.

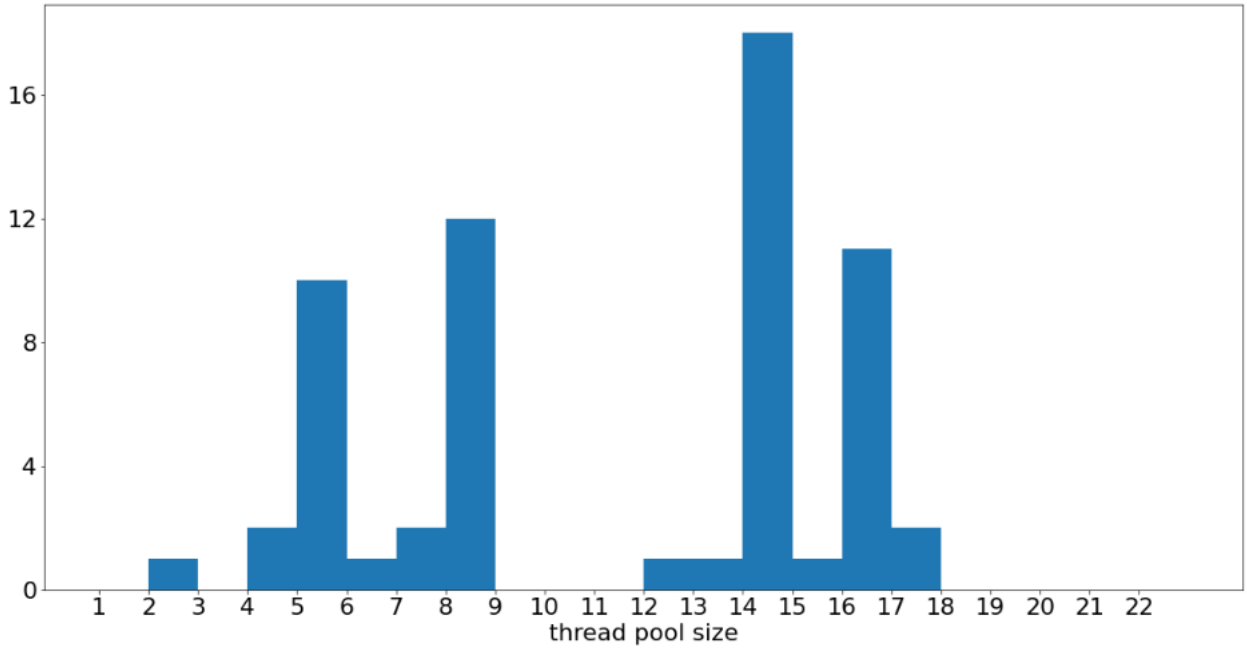

Figure 5.9: No. of occurrences of optimal thread pool sizes.

Then each following machine learning model is trained with the above data-set. In order to avoid over-fitting because of data set is small 5-fold cross validation is performed. For evaluation the performance of machine learning model 30% of data is used as test data. Then following machine learning models are selected and evaluated the accuracy of each model.

This problem is originally a regression problem. But when it is analyzed the optimal thread pool size for each program, the distribution is very narrowed. Figure 5.9 shows number of occurrences of each thread pool size. High frequency of optimal

Table 5.1: Fragment of training data set

Program	**Features**					**Optimal thread pool size**
	No. of database calls	**No. of HTTP calls**	**No. of gRPC calls**	**No. loops that contains database calls**	**More features...**	
Program 1	1	0	0	0	...	14
Program 2	1	2	0	0	...	5
...	...	...	...	...	...	...

thread pool values can be seen around values of 5,8,14 and 16. Thus four classes can be created and reduce the machine learning model to classification problem as well. But there are some low frequency values around high frequency values also. For example around the value of 14, there are values of 12,13 and 15. Frequency of them is one. Those programs can be assigned to nearby class by observing the data set. Considering this fact results are shown for both classification and regression models. However, the accuracy of classification model is low compared to regression model. Results is presented in table 5.2.

- XGBoost
- Support Vector Machine
- Decision Tree
- Random Forest

MAPE	Interpretation
<10	Highly accurate forecasting
10-20	Good forecasting
20-50	Reasonable forecasting
>50	Inaccurate forecasting

Source: Lewis (1982, p. 40)

Figure 5.10: Mean Absolute Percentage Error (MAPE) Value interpretation for regression models

For regression models Mean Absolute Percentage Error (MAPE) and Mean Squred Error (MSE) are evaluated. For classification models F1 score and accuracy

Table 5.2: Evaluation of different machine learning model

Machine Learning Model	**Classification**			**Regression**	
	Accuracy	**F1 score macro**	**F1 score weighted**	**MAPE**	**MSE**
XGBoost	62.73%	59.38%	62.29%	7.92%	0.91
SVM	59.09%	45.08%	52.97%	30.06%	13.41
Decision Tree	66.36%	62.93%	67%	**6.79%**	**0.91**
Random Forest	68.59%	61.58%	69%	9.17%	1.55%

are evaluated. When comparing models, it is clear that Decision Tree Regression model has the lowest Mean Absolute Percentage Error (MAPE) and Mean Squred Error (MSE) values. Lewis and Colin David [38] suggest ranges of MAPE values that provide insights on how good the forecasting. Figure 5.10 shows the summary of MAPE value interpretation. Thus, Decision tree model has highly accurate forecasting of thread pool size for given set of programming features. Despite MAPE value is lower than Decision tree, XGboost and Random forest model falls under highly accurate forecasting.

Chapter 6

Conclusions

6.1 Introduction

This chapter discusses the conclusion of the research project with regard to research questions defined in section 1.2. Furthermore, Limitations of the research and further implication of this research is discussed.

6.2 Conclusions about research questions (aims/objectives)

The initial objective of the research was finding a server architecture that would result in the best performance for a given Ballerina program. Initially, three server architectures were implemented in Ballerina run-time by removing,adding and modifying existing components. Server architectures were chosen based on previous studies [21, 1, 20, 2]. Number of thread pools and configuration of thread pools were major points when differentiating the server architectures. Performance of Current Ballerina architecture,Netty OIO and Removed Ballerina scheduler (specifically scheduler thread pool) were evaluated against different types of programs. Additionally, two variations of original Ballerina architecture were implemented by changing the size of the scheduler thread pool. Programs consisted of either CPU intensive or IO intensive features. Metrics such as throughput, average latency, standard deviation are evaluated. Experiments in the first phase showed that removing the scheduler thread pool resulted in higher throughput and lower latency

for all programs. Furthermore, Netty OIO performance is the worst for every program. Therefore we could not identify an architecture that would result in best performance for a given program among selected architectures.

Since changing thread pool size affected differently for IO and CPU intensive programs, we tried to answer the second research question which is optimizing performance of server architecture by tuning the thread pool size. There are several studies which try to provide analytical models for thread pool optimization. However none of them have addressed the thread pool optimization based on different program characteristics. Scheduler thread pool in current ballerina architecture is selected over removed scheduler architecture in order to tune because of stability issues when load to the web server (concurrency level) is increased. New programs were added for testing which have different IO characteristics (Database calls, HTTP calls, gRPC calls). Experiments were designed to measure impact of thread pool size for different IO features in programs. We were able to determine the impact of thread pool size for different programs which consisted of different IO features. We were able to find thread pool size which gives minimum average latency for each program.

The final objective was deriving a model to predict optimal thread pool size based on a given set of programming features. A machine learning model was trained in order to answer the last research question. The model takes input as features in the given ballerina program (i.e Number of database calls, HTTP calls, gRPC calls, loops) and estimates optimal thread pool size as output. Among several machine learning models, the Decision Tree regression model was chosen based on evaluation metrics. Hence, we were able to devise a model that estimates the thread pool size for a given Ballerina program.

6.3 Conclusions about research problem

In this research, we are able to propose a model to estimate optimal thread pool size for a given Ballerina program. Ballerina's native representation of remote IO calls a.k.a connector calls and other web service oriented features supported to extract these features directly from source code by parsing AST tree. The combination

of all modules makes it easier to decide the optimal thread pool size for Ballerina scheduler prior to execution of the program. This process can be integrated into the compilation process of the Ballerina program resulting in automatic thread pool size configuration in compiled code. The result of this study can be used as proof of concept for performance estimation based on programming features.

Furthermore, heuristic rules such as deciding the number of thread pool size as twice the number of CPU [6] is not resulting in optimal performance has been proven when analyzing the results of the first phase. In figures 5.1,5.2,5.3,5.4,5.5 , (Note that *scheduler x2* represent 4 times the number of CPUs and *scheduler x4* results 8 times the number of CPUs) we can see that performance is better when the number of threads are 4 times the number of CPUs than the current ballerina architecture where default thread pool size is 2 times the number of CPUs for all tested program types.

6.4 Limitations

There are a number of limitations that occurred while progressing the research. In the first phase of the research we analyzed both CPU intensive and IO intensive applications. However, when building the machine learning model it was difficult to extract CPU intensive features directly from the source code.Thus, only IO characteristics were considered which were able to parse from source code. As an example, we can model the CPU intensive behaviors as number of variable assignment, comparisons and arithmetic operations then it is possible to extract this information from source code. But when 3rd party library functions calls are happening, it is not able to derive this information directly from the source code. This requires, analysis of compiled byte code than source code. Thus, this study did not explore that level.

Furthermore, results of the experiment can be subject to the environment of the machine where the server is deployed. The effect of the amount of RAM number of the CPU is not analyzed in this research.

Also, performance results can be subject to run time variables as well. This study only provides static analysis. As an example, the number of times that

particular loop is executing can be depended on a variable. Some paths of the program may never get executed. Thus, for a given program features that are affected at run time may differ from the static information.

6.5 Implications for further research

For future studies this proof of concept can be made as the ground with combination of above limitations. We are able to prove that performance of web servers can be modeled based on program features. In this research we only considered the Ballerina language. Same set of experiments can be conducted for other frameworks and languages. If we can extract information from compiled code (byte-code in java) we can integrate total number of expressions, assignment, comparisons, loops and if clauses including 3rd party libraries as well. Then we can build an accurate model which includes both IO and CPU intensive features. Moreover, we can get relation among Number of CPU, Amount of RAM etc by running this model in multiple environments.

Appendix A

Publications

Appendix B

Diagrams

Appendix C

Code Listings

Test programs and AST parser implementation can be found at `https://github.com/lakinduakash/ballerina-microbenchmarks` [39]. Web sever architecture implementation can be found at `https://github.com/lakinduakash/ballerina-lang` [40]

```
1
2 package org.wso2.transport.http.netty.contractimpl;
3
4 import io.netty.channel.EventLoopGroup;
5 import io.netty.channel.group.ChannelGroup;
6 import io.netty.channel.group.DefaultChannelGroup;
7 import io.netty.channel.nio.NioEventLoopGroup;
8 import io.netty.channel.oio.OioEventLoopGroup;
9 import io.netty.util.concurrent.DefaultEventExecutorGroup;
10 import io.netty.util.concurrent.DefaultThreadFactory;
11 import io.netty.util.concurrent.EventExecutorGroup;
12 import io.netty.util.concurrent.GlobalEventExecutor;
13 import org.wso2.transport.http.netty.contract.Constants;
14 import org.wso2.transport.http.netty.contract.HttpClientConnector;
15 import org.wso2.transport.http.netty.contract.
    HttpWsConnectorFactory;
16 import org.wso2.transport.http.netty.contract.ServerConnector;
17 import org.wso2.transport.http.netty.contract.config.
    ListenerConfiguration;
18 import org.wso2.transport.http.netty.contract.config.
    SenderConfiguration;
```

```
19 import org.wso2.transport.http.netty.contract.config.
      ServerBootstrapConfiguration;
20 import org.wso2.transport.http.netty.contract.websocket.
      WebSocketClientConnector;
21 import org.wso2.transport.http.netty.contract.websocket.
      WebSocketClientConnectorConfig;
22 import org.wso2.transport.http.netty.contractimpl.common.ssl.
      SSLConfig;
23 import org.wso2.transport.http.netty.contractimpl.common.ssl.
      SSLHandlerFactory;
24 import org.wso2.transport.http.netty.contractimpl.listener.
      ServerConnectorBootstrap;
25 import org.wso2.transport.http.netty.contractimpl.sender.channel.
      BootstrapConfiguration;
26 import org.wso2.transport.http.netty.contractimpl.sender.channel.
      pool.ConnectionManager;
27 import org.wso2.transport.http.netty.contractimpl.websocket.
      DefaultWebSocketClientConnector;
28
29 import java.util.Map;
30 import javax.net.ssl.SSLException;
31
32 import static org.wso2.transport.http.netty.contract.Constants.
      PIPELINING_THREAD_COUNT;
33 import static org.wso2.transport.http.netty.contract.Constants.
      PIPELINING_THREAD_POOL_NAME;
34
35 /**
36  * Implementation of HttpWsConnectorFactory interface.
37  */
38 public class DefaultHttpWsConnectorFactory implements
      HttpWsConnectorFactory {
39
40     private final EventLoopGroup bossGroup;
41     private final EventLoopGroup workerGroup;
42     private final EventLoopGroup clientGroup;
43     private EventExecutorGroup pipeliningGroup;
44
```

```
45      private final ChannelGroup allChannels = new
    DefaultChannelGroup(GlobalEventExecutor.INSTANCE);
46
47      public DefaultHttpWsConnectorFactory() {
48          bossGroup = new OioEventLoopGroup(4);
49          workerGroup = new OioEventLoopGroup(200000);
50          clientGroup = new NioEventLoopGroup(Runtime.getRuntime().
    availableProcessors() * 2);
51      }
52
53      public DefaultHttpWsConnectorFactory(int serverSocketThreads,
    int childSocketThreads, int clientThreads) {
54          bossGroup = new OioEventLoopGroup(4);
55          workerGroup = new OioEventLoopGroup(200000);
56          clientGroup = new NioEventLoopGroup(clientThreads);
57      }
58
59      @Override
60      public ServerConnector createServerConnector(
    ServerBootstrapConfiguration serverBootstrapConfiguration,
61              ListenerConfiguration listenerConfig) {
62          ServerConnectorBootstrap serverConnectorBootstrap = new
    ServerConnectorBootstrap(allChannels);
63          serverConnectorBootstrap.addSocketConfiguration(
    serverBootstrapConfiguration);
64          SSLConfig sslConfig = listenerConfig.getListenerSSLConfig
    ();
65          serverConnectorBootstrap.addSecurity(sslConfig);
66          if (sslConfig != null) {
67              setSslContext(serverConnectorBootstrap, sslConfig,
    listenerConfig);
68          }
69          serverConnectorBootstrap.addIdleTimeout(listenerConfig.
    getSocketIdleTimeout());
70          if (Constants.HTTP_2_0.equals(listenerConfig.getVersion())
    ) {
71              serverConnectorBootstrap.setHttp2Enabled(true);
72          }
```

```
73      serverConnectorBootstrap.addHttpTraceLogHandler(
    listenerConfig.isHttpTraceLogEnabled());
74      serverConnectorBootstrap.addHttpAccessLogHandler(
    listenerConfig.isHttpAccessLogEnabled());
75      serverConnectorBootstrap.addThreadPools(bossGroup,
    workerGroup);
76      serverConnectorBootstrap.addHeaderAndEntitySizeValidation(
    listenerConfig.getRequestSizeValidationConfig());
77      serverConnectorBootstrap.addChunkingBehaviour(
    listenerConfig.getChunkConfig());
78      serverConnectorBootstrap.addKeepAliveBehaviour(
    listenerConfig.getKeepAliveConfig());
79      serverConnectorBootstrap.addServerHeader(listenerConfig.
    getServerHeader());
80
81      serverConnectorBootstrap.setPipeliningEnabled(
    listenerConfig.isPipeliningEnabled());
82      serverConnectorBootstrap.setWebSocketCompressionEnabled(
    listenerConfig.isWebSocketCompressionEnabled());
83      serverConnectorBootstrap.setPipeliningLimit(listenerConfig
    .getPipeliningLimit());
84
85      if (listenerConfig.isPipeliningEnabled()) {
86          pipeliningGroup = new DefaultEventExecutorGroup(
    PIPELINING_THREAD_COUNT, new DefaultThreadFactory(
87                  PIPELINING_THREAD_POOL_NAME));
88          serverConnectorBootstrap.setPipeliningThreadGroup(
    pipeliningGroup);
89      }
90
91      return serverConnectorBootstrap.getServerConnector(
    listenerConfig.getHost(), listenerConfig.getPort());
92   }
93
94   private void setSslContext(ServerConnectorBootstrap
    serverConnectorBootstrap, SSLConfig sslConfig,
95          ListenerConfiguration listenerConfig) {
96      try {
```

```
97          SSLHandlerFactory sslHandlerFactory = new
    SSLHandlerFactory(sslConfig);
98          serverConnectorBootstrap.
    addcertificateRevocationVerifier(sslConfig.
    isValidateCertEnabled());
99          serverConnectorBootstrap.addCacheDelay(sslConfig.
    getCacheValidityPeriod());
100         serverConnectorBootstrap.addCacheSize(sslConfig.
    getCacheSize());
101         serverConnectorBootstrap.addOcspStapling(sslConfig.
    isOcspStaplingEnabled());
102         serverConnectorBootstrap.addSslHandlerFactory(
    sslHandlerFactory);
103         if (sslConfig.getKeyStore() != null) {
104             if (Constants.HTTP_2_0.equals(listenerConfig.
    getVersion())) {
105                 serverConnectorBootstrap
106                         .addHttp2SslContext(sslHandlerFactory.
    createHttp2TLSContextForServer(sslConfig));
107             } else {
108                 serverConnectorBootstrap
109                         .addKeystoreSslContext(
    sslHandlerFactory.createSSLContextFromKeystores(true));
110             }
111         } else {
112             if (Constants.HTTP_2_0.equals(listenerConfig.
    getVersion())) {
113                 serverConnectorBootstrap
114                         .addHttp2SslContext(sslHandlerFactory.
    createHttp2TLSContextForServer(sslConfig));
115             } else {
116                 serverConnectorBootstrap.
    addCertAndKeySslContext(sslHandlerFactory.
    createHttpTLSContextForServer());
117             }
118         }
119     } catch (SSLException e) {
120         throw new RuntimeException("Failed to create ssl
    context from given certs and key", e);
```

```
121         }
122     }
123
124     @Override
125     public HttpClientConnector createHttpClientConnector(
126             Map<String, Object> transportProperties,
    SenderConfiguration senderConfiguration) {
127         BootstrapConfiguration bootstrapConfig = new
    BootstrapConfiguration(transportProperties);
128         ConnectionManager connectionManager = new
    ConnectionManager(senderConfiguration.getPoolConfiguration());
129         return new DefaultHttpClientConnector(connectionManager,
    senderConfiguration, bootstrapConfig, clientGroup);
130     }
131
132     @Override
133     public HttpClientConnector createHttpClientConnector(
134         Map<String, Object> transportProperties,
    SenderConfiguration senderConfiguration,
135         ConnectionManager connectionManager) {
136         BootstrapConfiguration bootstrapConfig = new
    BootstrapConfiguration(transportProperties);
137         return new DefaultHttpClientConnector(connectionManager,
    senderConfiguration, bootstrapConfig, clientGroup);
138     }
139
140     @Override
141     public WebSocketClientConnector createWsClientConnector(
    WebSocketClientConnectorConfig clientConnectorConfig) {
142         return new DefaultWebSocketClientConnector(
    clientConnectorConfig, clientGroup);
143     }
144
145     @Override
146     public void shutdown() throws InterruptedException {
147         allChannels.close().sync();
148         workerGroup.shutdownGracefully().sync();
149         bossGroup.shutdownGracefully().sync();
150         clientGroup.shutdownGracefully().sync();
```

```
151         if (pipeliningGroup != null) {
152             pipeliningGroup.shutdownGracefully().sync();
153         }
154     }
155 
156     /**
157      * This method is for shutting down the connectors without a
    delay.
158      **/
159     public void shutdownNow() {
160         allChannels.close();
161         workerGroup.shutdownGracefully();
162         bossGroup.shutdownGracefully();
163         clientGroup.shutdownGracefully();
164         if (pipeliningGroup != null) {
165             pipeliningGroup.shutdownGracefully();
166         }
167     }
168 }
```

Listing C.1: Netty OIO Implementaion

```
1  #!/usr/bin/env bash
2  
3  # Start mysql-server
4  docker stop my-mysql-benchmark 2>/dev/null
5  docker rm my-mysql-benchmark 2>/dev/null
6  docker run --name my-mysql-benchmark -e MYSQL_ROOT_PASSWORD=
       root@123 -p 3306:3306 --cpus="1.7" -d mysql
7  
8  echo "mysql server starting..."
9  
10 sql_query="create database testdb;
11 create table testdb.emp
12 (
13     id         int auto_increment
14         primary key,
15     firstName varchar(100) null
16 );
17 
```

```
INSERT INTO testdb.emp (id, firstName) VALUES (1, 'Lakindu Akash')
    ;"

while ! docker exec -it my-mysql-benchmark mysql -u root -
    proot@123  -e "${sql_query}" &>/dev/null; do
        sleep 0.1
done

echo "mysql server started"

# Download mysql-conector
if [ ! -f new_tests/benchmark/jar/mysql-connector-java-8.0.19.jar
    ]; then
     wget https://repo1.maven.org/maven2/mysql/mysql-connector-java
    /8.0.19/mysql-connector-java-8.0.19.jar -O new_tests/benchmark/
    jar/mysql-connector-java-8.0.19.jar
fi


docker stop bal-benchmark-cont 2>/dev/null
docker rm bal-benchmark-cont 2>/dev/null

rm -rf testcont/balruntime/benchmark
cp -rf new_tests/benchmark testcont/balruntime/benchmark

curdir=$(pwd)
cd testcont/balruntime
docker build . -t bal-benchmark -f bal.Dockerfile
cd $curdir
```

Listing C.2: Building Ballerina environment

```
#!/usr/bin/env bash

docker stop bal-benchmark-cont 2>/dev/null
docker rm bal-benchmark-cont 2>/dev/null

docker run -e "BALLERINA_MAX_POOL_SIZE=${1}" --name bal-benchmark-
    cont --net="host" -p 9090:9090 --cpus="2" -d bal-benchmark
```

Listing C.3: Running Ballerina tests

```
#!/bin/bash

pool_sizes=(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 22
    )

#Remove old results
rm -f testcont/jmeter/tests/results/*.json
rm -f testcont/jmeter/tests/results/*.jtl
rm -f testcont/jmeter/tests/results_*

#keys for localssh access
rm -f localssh.pub localssh
ssh-keygen -b 2048 -t rsa -f localssh -q -N ""
cat localssh.pub >> ~/.ssh/authorized_keys
chmod og-wx ~/.ssh/authorized_keys

cp localssh testcont/jmeter/tests/localssh
#End keys


bash jmeter-build-only.sh
bash ballerina-build-only.sh


for t in ${pool_sizes[@]}; do

echo "Setting pool size ${t}"

bash ballerina-run-only.sh $t
sleep 8

echo "Running tests..."

bash jmeter-run-only.sh $t

echo "finished benchmark with pool size ${t}"
sleep 1

```

```
done;
```

Listing C.4: Running Ballerina with different thread pool sizes

```
#!/usr/bin/env bash

docker stop jmeter-cont 2>/dev/null
docker rm jmeter-cont 2>/dev/null

curdir=$(pwd)
cd testcont/jmeter


# Removed deamon flag in order to run one by one
docker run -e "POOL_SIZE=${1}" --mount type=bind,source="$(pwd)"/
    tests,target=/usr/tests --network host --name jmeter-cont --
    cpus="2" jmeter

cd $curdir
```

Listing C.5: Running Jmeter

```
FROM openjdk:8
COPY jballerina-tools-2.0.0-SNAPSHOT /usr/ballerina/jballerina-
    tools-2.0.0-SNAPSHOT
COPY benchmark /usr/benchmark
WORKDIR /usr/benchmark
RUN ["/usr/ballerina/jballerina-tools-2.0.0-SNAPSHOT/bin/ballerina
    ","build", "--skip-tests","-a"]
CMD ["/usr/ballerina/jballerina-tools-2.0.0-SNAPSHOT/bin/ballerina
    ","run", "target/bin/main.jar"]
EXPOSE 9090
```

Listing C.6: Running Ballerina tests

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.3">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname
    ="Test_Plan_General_4_params" enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
```

```
6       <boolProp name="TestPlan.functional_mode">false</boolProp>
7       <boolProp name="TestPlan.tearDown_on_shutdown">true</
    boolProp>
8       <boolProp name="TestPlan.serialize_threadgroups">false</
    boolProp>
9       <elementProp name="TestPlan.user_defined_variables"
    elementType="Arguments" guiclass="ArgumentsPanel" testclass="
    Arguments" testname="User Defined Variables" enabled="true">
10        <collectionProp name="Arguments.arguments"/>
11      </elementProp>
12      <stringProp name="TestPlan.user_define_classpath"></
    stringProp>
13    </TestPlan>
14    <hashTree>
15      <ThreadGroup guiclass="ThreadGroupGui" testclass="
    ThreadGroup" testname="Thread Group" enabled="true">
16        <stringProp name="ThreadGroup.on_sample_error">continue</
    stringProp>
17        <elementProp name="ThreadGroup.main_controller"
    elementType="LoopController" guiclass="LoopControlPanel"
    testclass="LoopController" testname="Loop Controller" enabled="
    true">
18          <boolProp name="LoopController.continue_forever">false</
    boolProp>
19          <intProp name="LoopController.loops">-1</intProp>
20        </elementProp>
21        <stringProp name="ThreadGroup.num_threads">${__P(threads
    ,5)}</stringProp>
22        <stringProp name="ThreadGroup.ramp_time">${__P(ramp_time
    ,1)}</stringProp>
23        <boolProp name="ThreadGroup.scheduler">true</boolProp>
24        <stringProp name="ThreadGroup.duration">${__P(duration,60)
    }</stringProp>
25        <stringProp name="ThreadGroup.delay"></stringProp>
26        <boolProp name="ThreadGroup.same_user_on_next_iteration">
    false</boolProp>
27      </ThreadGroup>
28      <hashTree>
29        <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="
```

```
HTTPSamplerProxy" testname="HTTP Request" enabled="true">
30          <elementProp name="HTTPsampler.Arguments" elementType="
Arguments" guiclass="HTTPArgumentsPanel" testclass="Arguments"
testname="User Defined Variables" enabled="true">
31            <collectionProp name="Arguments.arguments">
32              <elementProp name="${__P(param_1_name,param_1)}"
elementType="HTTPArgument">
33                <boolProp name="HTTPArgument.always_encode">true</
boolProp>
34                <stringProp name="Argument.value">${__P(
param_1_value,0)}</stringProp>
35                <stringProp name="Argument.metadata">=</stringProp
>
36                <boolProp name="HTTPArgument.use_equals">true</
boolProp>
37                <stringProp name="Argument.name">${__P(
param_1_name,param_1)}</stringProp>
38              </elementProp>
39              <elementProp name="${__P(param_2_name,param_2)}"
elementType="HTTPArgument">
40                <boolProp name="HTTPArgument.always_encode">true</
boolProp>
41                <stringProp name="Argument.value">${__P(
param_2_value,0)}</stringProp>
42                <stringProp name="Argument.metadata">=</stringProp
>
43                <boolProp name="HTTPArgument.use_equals">true</
boolProp>
44                <stringProp name="Argument.name">${__P(
param_2_name,param_2)}</stringProp>
45              </elementProp>
46              <elementProp name="${__P(param_3_name,param_3)}"
elementType="HTTPArgument">
47                <boolProp name="HTTPArgument.always_encode">true</
boolProp>
48                <stringProp name="Argument.value">${__P(
param_3_value,0)}</stringProp>
49                <stringProp name="Argument.metadata">=</stringProp
>
```

```
50              <boolProp name="HTTPArgument.use_equals">true</
    boolProp>
51              <stringProp name="Argument.name">${__P(
    param_3_name,param_3)}</stringProp>
52            </elementProp>
53            <elementProp name="${__P(param_4_name,param_4)}"
    elementType="HTTPArgument">
54              <boolProp name="HTTPArgument.always_encode">true</
    boolProp>
55              <stringProp name="Argument.value">${__P(
    param_4_value,0)}</stringProp>
56              <stringProp name="Argument.metadata">=</stringProp
    >
57              <boolProp name="HTTPArgument.use_equals">true</
    boolProp>
58              <stringProp name="Argument.name">${__P(
    param_4_name,param_4)}</stringProp>
59            </elementProp>
60          </collectionProp>
61        </elementProp>
62        <stringProp name="HTTPSampler.domain">${__P(host_name,
    localhost)}</stringProp>
63        <stringProp name="HTTPSampler.port">${__P(host_port
    ,9090)}</stringProp>
64        <stringProp name="HTTPSampler.protocol"></stringProp>
65        <stringProp name="HTTPSampler.contentEncoding"></
    stringProp>
66        <stringProp name="HTTPSampler.path">${__P(request_path,
    microbenchmark/cpu)}</stringProp>
67        <stringProp name="HTTPSampler.method">GET</stringProp>
68        <boolProp name="HTTPSampler.follow_redirects">true</
    boolProp>
69        <boolProp name="HTTPSampler.auto_redirects">false</
    boolProp>
70        <boolProp name="HTTPSampler.use_keepalive">false</
    boolProp>
71        <boolProp name="HTTPSampler.DO_MULTIPART_POST">false</
    boolProp>
72        <stringProp name="HTTPSampler.embedded_url_re">></
```

```
   stringProp>
73          <stringProp name="HTTPSampler.connect_timeout"></
   stringProp>
74          <stringProp name="HTTPSampler.response_timeout"></
   stringProp>
75        </HTTPSamplerProxy>
76        <hashTree/>
77      </hashTree>
78    </hashTree>
79  </hashTree>
80 </jmeterTestPlan>
```

Listing C.7: Jmeter Test plan

References

[1] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, "Comparing the performance of web server architectures," *SIGOPS Oper. Syst. Rev.*, vol. 41, p. 231–243, Mar. 2007.

[2] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, (New York, NY, USA), p. 230–243, Association for Computing Machinery, 2001.

[3] J. Behren, J. Condit, and E. Brewer, "Why events are a bad idea," pp. 19–24, 05 2003.

[4] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, (New York, NY, USA), p. 186–189, Association for Computing Machinery, 2002.

[5] D. Xu, "Performance study and dynamic optimization design for thread pool systems," 2004.

[6] Y. Ling, T. Mullen, and X. Lin, "Analysis of optimal thread pool size," *SIGOPS Oper. Syst. Rev.*, vol. 34, p. 42–55, Apr. 2000.

[7] D. Freire, R. Z. Frantz, F. Roos-Frantz, and S. Sawicki, "Optimization of the size of thread pool in runtime systems to enterprise application integration: A mathematical modelling approach," vol. 20, pp. 169–188, 04 2019.

[8] M. D. Syer, B. Adams, and A. E. Hassan, "Identifying performance deviations in thread pools," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 83–92, IEEE, 2011.

[9] H. Linfeng, G. Yuhai, and W. Juyuan, "Design and implementation of high-speed server based on dynamic thread pool," in *2017 13th IEEE International Conference on Electronic Measurement & Instruments (ICEMI)*, pp. 442–445, IEEE, 2017.

[10] A. F. Lorenzon, M. C. Cera, and A. C. S. Beck, "Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy," *Journal of Parallel and Distributed Computing*, vol. 95, pp. 107–123, 2016.

[11] J. Nieplocha, A. Márquez, J. Feo, D. Chavarría-Miranda, G. Chin, C. Scherrer, and N. Beagley, "Evaluating the potential of multithreaded platforms for irregular scientific computations," in *Proceedings of the 4th International Conference on Computing frontiers*, pp. 47–58, 2007.

[12] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, "Adaptive scheduling with parallelism feedback," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 100–109, 2006.

[13] Kyungtae Woo, Chansu Yu, Dongman Lee, Hee Yong Youn, and B. Lee, "Non-blocking, localized routing algorithm for balanced energy consumption in mobile ad hoc networks," in *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 117–124, 2001.

[14] C. D. Saether and P. S. Jr., "System of device independent file directories using a tag between the directories and file descriptors that migrate with the files.," U.S. Patent No. 5,333,315. 26 Jul., 07 1994.

[15] N.-M. Yao, M.-Y. Zheng, and J.-B. Ju, "High performance web server based on pipeline," vol. 14, pp. 1127–1133, 06 2003.

[16] axboe, "Io uring." Available at `https://github.com/axboe/liburing`.

[17] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), p. 268–281, Association for Computing Machinery, 2003.

[18] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, (New York, NY, USA), p. 186–189, Association for Computing Machinery, 2002.

[19] S. Bharti, V. Kaulgud, S. Padmanabhuni, V. Krishnamoorthy, and N. Krishnan, "Fine grained seda architecture for service oriented network management systems," *International Journal of Web Services Practices*, vol. 1, pp. 158–166, 01 2005.

[20] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An efficient and portable web server," 1999.

[21] A. S. Harji, P. A. Buhr, and T. Brecht, "Comparing high-performance multi-core web-server architectures," in *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[22] Harji, Ashif, "Performance comparison of uniprocessor and multiprocessor web server architectures," 2010.

[23] S. Weerawarana, C. Ekanayake, S. Perera, and F. Leymann, "Bringing middleware to everyday programmers with ballerina," in *Business Process Management* (M. Weske, M. Montali, I. Weber, and J. vom Brocke, eds.), (Cham), pp. 12–27, Springer International Publishing, 2018.

[24] "The eight fallacies of distributed computing - tech talk." `https://web.archive.org/web/20171107014323/http://blog.fogcreek.com/eight-fallacies-of-distributed-computing-tech-talk/`.

[25] A. Oram, *Ballerina: A Language for Network-Distributed Applications*. O'Reilly Media, Incorporated.

[26] "Ballerina plugin for vs code." `https://github.com/ballerina-platform/plugin-vscode`.

[27] "Ballerina plugin for intelij ide." `https://github.com/ballerina-platform/plugin-intellij`.

[28] B. Wescott, *Every computer performance book: how to avoid and solve performance problems on the computers you work with*. CreateSpace, 2013.

[29] T. P. Adewumi, "Inner loop program construct: A faster way for program execution," *Open Computer Science*, vol. 8, no. 1, pp. 115–122, 2018.

[30] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Q.*, vol. 28, p. 75–105, Mar. 2004.

[31] M. W. Norman Maurer, *Netty in Action*. Manning Publications.

[32] Wso2, "Ballerina performance test suite - source code." `https://github.com/ballerina-platform/ballerina-performance`.

[33] Wso2, "Ballerina official website." `https://ballerina.io/`.

[34] Wso2, "Http trasnport library." `https://github.com/wso2/transport-http`.

[35] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[36] E. H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[37] "Tsung - a load testing tool using erlang language." `http://tsung.erlang-projects.org/`.

[38] C. D. Lewis, *Industrial and business forecasting methods: A practical guide to exponential smoothing and curve fitting*. Butterworth-Heinemann, 1982.

[39] "Test program implementation, ast parser implementations and other scripts used for conducting experiments." https://github.com/lakinduakash/ballerina-microbenchmarks.

[40] "Implementation of server architectures in ballerina." https://github.com/lakinduakash/ballerina-lang.