Prediction and Frequency Based Dynamic Thread Pool System A Hybrid Model

Faisal Bahadur
Hazara University
5 PUBLICATIONS **1** CITATION
SEE PROFILE

Abdul Hakeem
Hazara University
10 PUBLICATIONS **9** CITATIONS
SEE PROFILE

Miraj Gul
Hazara University
2 PUBLICATIONS **1** CITATION
SEE PROFILE

Arif Iqbal Umar
Hazara University
76 PUBLICATIONS **496** CITATIONS
SEE PROFILE

Prediction and Frequency Based Dynamic Thread Pool System A Hybrid Model

Sumat Nazeer [#], [*]*Faisal Bahadur* [#], Mohammad Abrar Khan [+], Abdul Hakeem [#], Miraj Gul [#], Arif Iqbal Umar [#]

[#] Department of Information Technology
Hazara University
Mansehra, Pakistan

[+] Institute of Information Technology
Kohat University of Science and Technology
Kohat, Pakistan

Abstract— ***As Internet is continuously expending, web servers receive hundreds of thousands hits per day, to cope with these increasing demand we must have some high performance web servers. Multithreading is one of the most fundamental approaches used in web servers to achieve high performance. To handle high number of requests from remote servers, thread-pool based multithreading model is used, where worker threads are responsible for serving these requests. Multiple threads work in a pool, which need to be synchronized. In TEMA model of thread pool, synchronization overhead is one of the main reasons that lead to wrong predictions of thread pool size and delayed response time which results in performance degradation. In our work we have proposed a Hybrid scheme to dynamically optimize a thread pool that is based on predicting incoming request frequencies. The scheme removes synchronization overhead of TEMA in order to achieve better predictions in pool size. Simulation results ensure that the Hybrid model outperforms TEMA by achieving 94% prediction accuracy as compared to the 76% claimed by TEMA. The gained prediction accuracy comes out to be 23%. The prediction accuracy of Hybrid scheme ultimately resulted in lessening the response time from 213ms to 203ms has also been achieved.***

Keywords- Multithreading , Dynamic Thread Pool, Concurrency , Web server , application server

I. INTRODUCTION

In real world, at a particular time, different event keep occurring together at the same time. So when we plan a product to watch and control true frameworks, we should try to manage this simultaneity. In short, we can say that Concurrency is the inclination for various occurrences to happen at the same time me in a framework. In other words, we can say that Concurrency is the propensity for various occasions to happen at the same time in a system. Whereas if we go through the dictionary meaning of concurrency or concurrent than it comes out to mean as "working in the meantime" or "running parallel." It additionally is characterized as "meeting or tending to meet at the same point," or "focalized”. Concurrency is helpful in multi core, multiprocessor and distributed computer systems as it increases performance and reliability, and provides fault tolerance in most of applications, like email that is inherently distributed.

In order to achieve better and improved performance of server side programs, many different concurrency models have been introduced till date. Figure 1.1 shows the concurrent execution of different jobs.

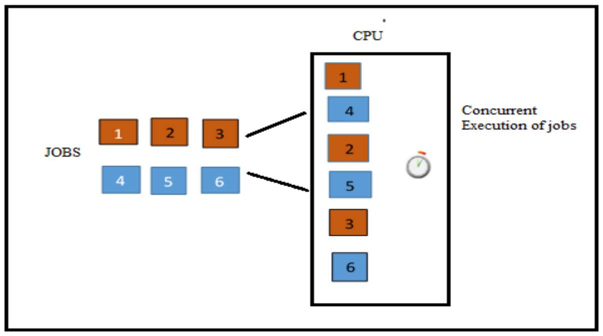

Figure 1.1.Concurrency model

. Multithreading is most widely used model for concurrency. The goal of multithreading is to improve overall performance using multiple threads. Threads are one of a few advances that make it conceivable to execute different code ways simultaneously inside a solitary application.

A. *Multithreaded Server Architecture*

There are two types of Multithreading architectures i.e. thread-per-request and thread-pool architectures.

Thread-Per-Request: The thread-per-request design provides responsiveness to clients which single thread is not able to provide. It will maximize the responsiveness towards user requests and is also easy to implement. In this architecture when a client's request is submitted, a thread is created to serve that request. Or in other words, threads are created as per request and after completing that request, the thread is killed. The main disadvantage of thread per request is that it causes processing overhead due to continuous creation and destruction of threads. Besides, there is also risk of server crash, if the number of request crosses the tolerance threshold of the server.

Thread Pool: Thread pools are most important as they are used to enhance the throughput and responsiveness of server side application. A thread pool is a" container of available threads" when a client's request arrives, an available thread is assigned to serve that request and after serving the request, the thread is returned back to the pool, and so on. The thread pool architecture consists of a Task queue, worker thread, and a pool of thread, that manages the worker threads. Thread pool is not that much affected by overhead as is thread per request and allocates limited number of operating system resources.

*B. **Moving Averages***

Moving averages are the most popular and most used statistical indicators that are commonly employed in stock exchange servers, for predicting/forecasting the direction of future trends. Moving average is defined as a mathematical proof which is determined by taking an average number of past data points, and then the results are plotted using these determined points in the form of graph, so that traders can look at processed data instead of focusing on the day-to-day price variations.

Exponential Moving Average: Exponential moving average is a kind of Moving average that gives more weight to recent prices instead of focusing on all past prices. So, it is more responsive because the recent trend is generally more informative as an investor will apply a stronger weight on the more recent data. In EMA the level of forecasting lies between the currently observed value and last noted values. As EMA is applied for Predicting/forecasting future trends in price market, in the same way it can also be applied on thread pool so that it predicts the number of threads in advance, that are required for request serving by calculating the exponential average of incoming Requests frequencies. So when a client request arrives it is served by the available thread in the pool without any delay, as the demanded threads are already available in pool.

II. RELATED WORK

In this section different types of thread pool models are discussed. Also, it overviews how thread pooling affects the performance of server side applications. Finally, it discusses the motivation and how this work is implemented for improving the system performance.

2.1. Thread Pool Based on CORBA Implementation

The idea of thread pool was first presented by in [1]. After checking the thread pool concurrency model and illustrating it, [1] shows how multi-threaded servers can be developed for the use of distributed stock quote application. An I/O string initially puts every in-coming demand at the tail of demand line inside this thread pool and a worker thread from the pool queues the demand from the head of the demand line. This technique diminishes the cost of thread creation and bounds the resources of operating system to be used in an efficient manner. Client requests are executed simultaneously until the quantity of concurrent demands surpasses the quantity of threads in the pool. The major shortcomings of this model are context switching overhead and synchronization overhead at the level of request queuing that commonly results in the absence of dynamic optimization strategy.

2.2. Thread Pool Model Based On ORB Implementation

This model is presented by [2]. They displayed a detail plan and usage of multi-threaded object request broker (ORB).

This model was implemented by using Windows NT and underlying TCP protocol. When a client submits a request to server, using this model, the request is straightforwardly executed by free accessible thread in the pool, without disturbing the demand line, whereas ORB provides an interface for client-server communication. This model also uses a daemon process at server side that is responsible for receiving client request, storing and retrieving information from request massage, and also for activating and deactivating client requests. This model is more appropriate for those applications in which low-rated in-coming demand is involved as compared to those applications in which the measurement of thread pool is gradually streamlined because of heavy load of in-coming demands.

2.3. Thread Pool Model Based On Shared-Memory Pool Techniques

This model is presented in [3]. He presented a model in which he targeted the multi-threaded authorization system; the model shows performance improvement by introducing shared memory pool.

For card validation it applies a divide and conquer approach. The process of validation occurs in two steps, the first one of which is card restriction validation and the second one is online fraud validation. They contribute to improve the system performance and response time, and also involves less expensive I/O operation. The problem with this model is poor response time on high request rate and unavailability of dynamic tuning for proposed thread pool.

2.4. Thread Pool Model Based On Real Time(RT) ORB Implementation

This model is presented in [4]. They presented the watermark thread pool model for RT CORBA servers, for the purpose of dynamic thread pooling. When a thread pool is initialized, it has some fixed number of statically allowed threads known as low watermark; the size of this pool dynamically grows to a specified value in order to handle the bursts of client requests known as high watermark. In this model Server applications at first indicate the quantity of static threads, the quantity of maximum threads required in future and the priorities of threads in pool. When a client submits request to server and the available servers are too busy to serve the request then in that case a new thread is created to serve the client request. However, if the number of threads in a pool reaches the highest watermark value, then no more additional threads will be added to the pool. The new incoming request resides in the available space inside request queue until a thread is free to serve the request. If there is no space available in request queue, then the ORB generates a TRANSIENT exception message, showing a short time resource limitation, enabling the client to submit the request at some other time.

2.5. Thread Pool Model Based On Heuristic Approach

Thread Pool model is based on Heuristic approach and is presented in [5]. It is also known as Dynamic Thread Pool model and is used for thread pool size adjustment. This heuristic based algorithm first calculates the average idle time of five completed jobs, and then calculates the percentage difference between the Average Idle Time (AIT) of currently completed jobs and the AIT of previous jobs.
On the basis of these calculations, if the difference of AITs exceeds by 1% then the thread pool size is either increased or decreased.
This model suffers from delay in response time causing slow performance which is due to the use of too many thresholds in dynamic tuning algorithm.

2.6. Thread Pool Based on Optimal Size Analysis

Thread Pool based on optimal size analysis model is presented in [6]. In Contrast to [5] that uses numerous thresholds, it uses only two variables namely C1 & C2. C1 represents single thread creation/destruction overhead whereas C2 represents the maintenance overhead of every thread.

C1 and C2 are used as key elements for dynamic tuning of thread pool. These overheads are measured by calculating the value of probability distribution time taken by each event. The formula developed in [6] indicates a useful relationship among pool size, load given to system and overhead involved. Thread pool based on optimal size analysis however fails to model the actual performance of the thread-pool. The reason behind [6] failure in modeling the performance of the thread-pool is due to the calculation overhead in thread creation/destruction overhead as well as practical difficulty in maintenance overhead.
This model is affected by overhead in the form of making additional threads.

2.7. Thread Pool Based on DAM Driven By Heuristic Approach

This model is proposed by [7] which is about Web application server and its connection with thread pool model. It takes into account a vibrant system which contains heuristic components for mirroring the setting at run-time to improve changing the size of thread pool. In this model Tc and Ta are used in term of heuristic for determination of the amount of the requests and the average wait time. These variables properly indicates running-time position of thread pool efficiently due to which the thread pool size is characterized dynamically and powerfully.
The only shortcoming of this model is delay in response time as Request Queue must be locked until AIT is calculated.

2.8. Thread Pool Based On Thread Adaption Mechanism

This model is presented by [8]**.** He proposed a thread pool by utilizing framework of thread adaption. According to his model, thread pools that run in an application server inside various administrations adjust/absorb the coming threads. AIT of requests is observed by the over-load monitor in which they are in wait state inside the request queue after every 20 milliseconds and distinguishes it with previous recorded AIT. When the sensitivity threshold value is low and change in AIT is high then it forwards the change to the controller for redistribution of the strings among thread pools and obtains

strings from those pools which have smaller number of change in AIT. This model significantly enhances response times for element segments on a bustling all around tuned string pool.
The inadequacy in this model is the occasional registering of AIT from request line that mars the implementation, as backlog needs to be matched or obstructed till the completion of the calculation of AIT.

2.9. Thread Pool Model Based on Model Fuzzing Approach

The proposed model [9] configures Resource Manager Parameters, known as model fuzzing, according to which for describing the numerous variations of the managed framework, the parameters of the framework model will change. In this model, parameters of pool management are noted at continues intervals and if system tends to degrade, then the parameters are set back to those configurations at which the system shows high performance. The main drawback of this model is that new load of work on the server and the request arrival pattern possibly has changed dimensions in different times. Due to which the past design settings would not be most appropriate to the new work load. Thread pool model based on model fuzzing is described in [9].

2.10. *Thread Pool For Distributed Shared Memory (DSM)*

In [10], the authors propose DSM in which non-blocking queues are used for Configuration and Implementation of Hierarchical Thread Pool Executor (HTPE). The sophistication of this model is its scalability i.e. adding more machines and its freedom from requiring tuning for adding more machines.

2.11. Thread Pool For Balancing Mobile Agent In Distributed Environment

This model is presented in [11] which presents a distributed thread pool model, using Executor service that can dynamically maximize the size and nodes and minimize them automatically without using specific node adjuster. In this model Load balancing is done by a central load balancer node. DAI (Distributed Artificial Intelligence) is based upon this model. In DAI frameworks there is no need to combine all the pertinent information in a solitary area, as in monolithic and centralized Artificial Intelligence frameworks which have topographically close handling hubs and are firmly coupled.

2.12. Thread Pool Based on Prediction Scheme Using Gaussian Distribution

This model is proposed by [12]. They proposed a scheme in which they use Gaussian distribution where anticipation is obtained from the modelling change for the required threads in the pool for maximizing resource utilization. In this approach threads are pre-created and pre-deleted. When client requests increase then more threads are made in advance. Similarly, when client requests decrease then it doesn't diminish the quantity of threads rapidly to keep away from superfluous overhead of thread creation/cancellation. Limitation of the model lies in its setting the coefficient of the linear mathematical statement which entirely is based on the most recent and present numbers of threads.

2.13. Thread Pool Based on Prediction Scheme using Exponential Moving Averages

This model is proposed by [13]. For the implementation of a prediction-based dynamic thread pool, it uses the Exponential moving averages. This model consists of a main program, a watcher thread, a worker thread, and a thread pool. Watcher thread determines the volume base progress of pool size by keeping dynamic number of threads. If there is a conflict in thread creation then it requires synchronization on each thread creation, thread blocking and thread destruction.
Using exponential average, the predicted value is calculated using the formula T^{n+1}. The outcome is then compared with previously recorded rate t^n. In case the result of T^{n+1} is greater than the currently predicted value of threads t^n, then the required worker thread are formed and are added to the pool. Due to frequent changes in request entry, execution can be affected on account of watcher's synchronization overhead that leads to wrong predictions; another problem with this plan is redundant threads.

2.14. Thread Pool Based of Prediction Scheme using Trendy Exponential Moving Averages (TEMA)

This model is proposed by [14]. The authors proposed a prediction based thread pool for running of server and server side application system. By introducing the trendy time series, greater throughput is achieved by using exponential moving average (EMA) model [12]. In TEMA, a mathematical model is proposed which shows the relation between idle/free timeout period and threshold of thread pool parameters. TEMA is capable of dynamically adjusting these two parameters. This model overcomes the problem of redundant threads but suffers from lacking threads which causes delay in response time. Another limitation of TEMA is synchronization overhead that causes wrong predictions.

2.15. *Thread Pool Based On A Frequency Based Optimization* Strategy(FBOS)

This model is proposed by [15]. They present a dynamically tuned model for thread pooling. In this scheme, pool size dynamically increases, as the frequency of incoming requests increase. More threads are created in the pool if the amount of incoming request frequency increases. The only shortcoming of FBOS is delay in response time due to the abrupt increase in request frequency.

III. MOTIVATION

In literature review, several thread pool models are presented for achieving better concurrency in web servers. Most of the techniques discussed earlier suffer from delay in response time, low performance, synchronization overhead, wrong predictions, context switching etc. In [15], thread creation time overhead is of short duration only i.e. increase in request frequency leads to high response time for some of the clients which consequently demands threads creation in advance, as in prediction based approach. Whereas in prediction based approach [14, 13] synchronization overhead is one of the reasons that leads to wrong prediction. These limitations make the technique less effective which has driven us to devise a solution for overcoming these limitations.

IV. PROBLEM SPECIFICATION

This section considers and reviews the implementation of prediction model based on TEMA in detail.

A. Prediction Model Based on TEMA

It is very hard to implement TEMA because of frequent changes in request arrival results in excessive synchronization overhead and the more the increase in synchronization overhead results in the more is context switch, slowing down the system. Besides, synchronization overhead is one of the reasons that results in wrong prediction. This scheme suffers from lacking threads which can increase latency for some of the requests.

In this model Watcher thread is responsible to calculate the changing rate in pool size t^n after every 0.1 sec, which is done by counting active threads in pool. It is a synchronized counter that is updated after every 0.1 sec in every situation when whenever a thread is created, destroyed, blocked or activated. So this t^n is updated upon every operation, in which case when watcher is taking value for measuring the current number of thread in worker pool, suddenly the threads whose idle time exceed are deleted, resulting in wrong prediction e.g. watcher thread reads that there are 4 threads in pool and the threads required are 6. So, when the watcher creates 2 more threads, then suddenly the threads in the worker pool, whose idle time exceed are deleted and the value of t^n abruptly gets changed, resulting in wrong prediction i.e. the required number are 6 whereas the watcher predicts only 2.

Figure 4.1identifies the point where the synchronization overhead exists.

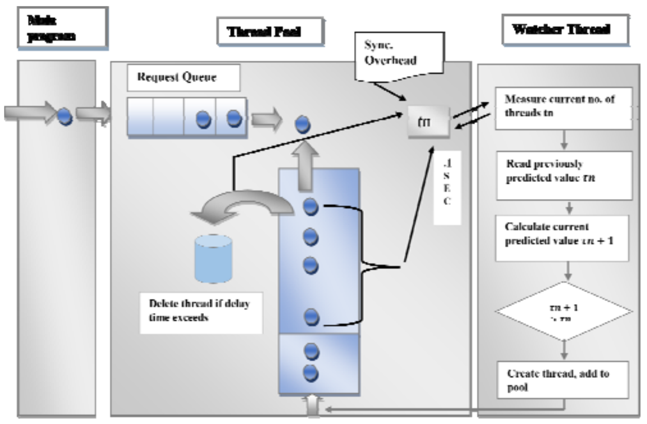
Main program
Thread Pool
Sync. Overhead
Watcher Thread
Request Queue
tn
Measure current no. of threads tn
Read previously predicted value τn
.1 S E C
Calculate current predicted value τn + 1
Delete thread if delay time exceeds
τn + 1 > τn
Create thread, add to pool

Figure 4.1.Problem Identification in TEMA Model

V. PROPOSED SOLUTION

This section provides description of the design of prediction and frequency based dynamic thread pool system and also the collection of components used in system model.

A. Design Components

- **Dynamic Thread Pool:** This is an object class that shows the complete system. Also this object class is a start-up class as it will start the coordinating components. On a system start-up, thread pool class will create an object of Request Queue and Worker thread Queue.

- **Request Queue:** Request Queue is the place holding the given number of incoming clients solicitation. These solicitations are submitted to queue, using First In First Out (FIFO) based data structure i.e. solicitation are submitted form one side of queue and exit for further processing from other side of the request queue.

- **Worker thread Pool:** Worker Threads Queue is responsible for servicing incoming client's requests. The size of the pool will increase and decrease dynamically in advance, depending upon the frequency of incomings requests.

- **Worker thread Queue:** This object class is responsible for managing the requests taken for execution. This is responsible for assigning the available thread to incoming request. Also it contains a timer class that is responsible for destroying a thread if the idle wait time increases by 3sec.
- **Frequency Detector:** This object class has control over incoming request rate i.e. to notice the rate of arriving client requests. There is clock inside the detector that gets invoked after a uniform interval of .2 sec to detect the rate of incoming request frequency and then store the rate in Frequency Holder Object. After storing frequency rate the Frequency Detector updates the status.
- **Frequency Holder:** This object class of Frequency Holder is responsible for storing frequency rate. Frequency Detector will update this object after every .2 seconds.
- **Watcher thread:** An object of this class is responsible for making prediction on the basis of frequency rate, noted by Frequency Holder. The Watcher thread makes predictions on the basis of the frequency rate of incoming requests and creates desired amount of threads for the incoming stage. The future predictions for thread creation are done by using the exponential average, by observing the currently read request frequency from Frequency Holder. Afterward, the value of currently recorded frequency CPF is contrasted with the previous predicted value i.e. PPF. If CPF is greater than PPF then additional number of new threads are generated and put in to the worker thread queue.
- **Dust Bin Pool:** This is an object of Worker Pool that is activated when the average idle time or wait time of a thread in a pool increases by 3sec. So the extra threads from thread pool are deleted to avoid redundant threads in worker pool.

B. *System Design(Hybrid Model)*

This passage introduces the design of Hybrid thread pool system which is sorted out by gathering of components that we have specified above.

Principle of Hybrid thread pool system is to enhance the Quality of administration i.e. each solicitation that is submitted to the system ought to get a reasonable and brisk reaction and for accomplishing this objective we require the thread pool system to be convenient, effectively quantifiable and tentatively verifiable.

Hybrid thread pool systems at start up time the system have some initial amount of threads that are equal to the number of processor in the machine. When client's requests arrive the frequency of incoming requests are detected by frequency detector. The Hybrid thread Pool System maintains a counter, named frequency Counter, that is updated after every arrival of request, the value of this counter is noted repeatedly by Frequency Detector. The detector first notes counter value and set it back to zero value so that the counter can sustain the frequency of next execution times. At the same time the requests are placed in the request queue in FIFO order.

When the requests are in request queue, the frequency detector forwards the detected frequencies to frequency holder, from where the watcher thread reads the requests frequencies and predicts the desired figures of threads in advance. If current frequency is greater than the previous frequency than the watcher adds more threads to worker pool. The worker pool manages the required number of threads and assigns threads to requests. Every thread in the worker pool has a ticker that counts the number of time a thread goes into inactive state, without executing any request. When thread is initializing for executing a request and transits to busy state then its timer stops. The ticker starts again if the thread goes back into the inactive state; if the inactive time of thread is up to two seconds, it will be automatically destroyed and the pool size will be managed. Figure 5.1 shows the Design and working of hybrid thread pool system.

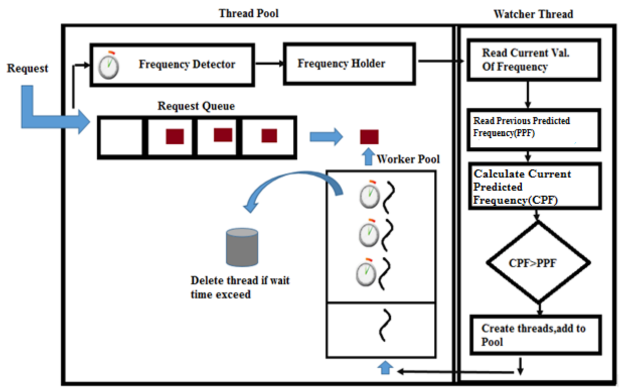
Thread Pool
Watcher Thread
Request
Frequency Detector
Frequency Holder
Request Queue
Worker Pool
Delete thread if wait time exceed
Read Current Val. Of Frequency
Read Previous Predicted Frequency(PPF)
Calculate Current Predicted Frequency(CPF)
CPF>PPF
Create threads,add to Pool

Figure 5.1 Hybrid Thread Pool Model

C. *Flow Diagram*

Figure 5.2 is a flow diagram that shows how Frequencies are detected and predictions are made on the basis of the recorded incoming requests frequencies.

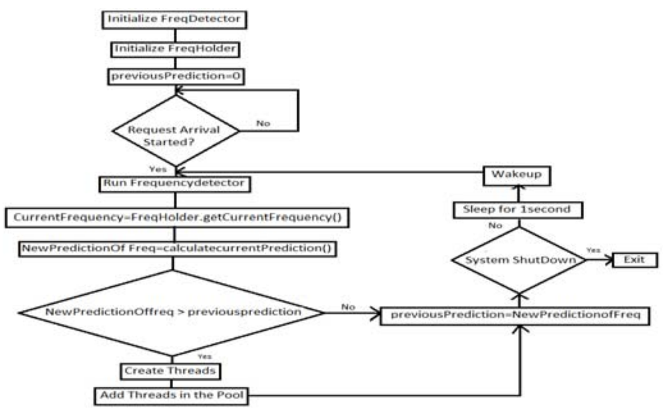

Figure 5.2.Working of Hybrid Model

VI. SIMULATION SETUP

In order to determine the performance of Hybrid system, we have used a simulator written in Java, which analyzes the pattern of incoming request frequencies. We have uploaded our scheme i.e. Prediction and Frequency based dynamic thread pool on a simulator named as Thread-Pool-Tester [16]. Thread-Pool-Tester is also known as TPT, as it is a combination of two separate tiers i.e. a virtual client side and a virtual server side. The machines on both client and server side have specifications of Intel core i3 and the processor is Intel (R) Core (TM) i3-4010U 1.70 GHz with 4 GB RAM. We have performed simulations by generating different loads on the server using poison distribution and with workload of 200ms. The results produced by performing simulation are plotted by TPT in the form of a graph. The results are discussed in the section below:

VII. ANALYSIS AND RESULTS

In this section, performance refinement strategy is presented and the results obtained are compared with those of previous strategy. The comparison is made between the Hybrid and TEMA models on the basis of their pool size, number of predictions made and the response time of different requests submitted to them. The Hybrid Model is based on the rate of arrival of current frequencies rather than on the number of threads in pool. In this way, synchronization overhead is avoided. In our strategy, we analyze the performance of system after the completion of jobs, and the simulator generates graphs for pool sizes, response times, and predicted number of threads. The generated graphs are used for making comparisons of both strategies. Figure 6.1 presents Dynamic request frequencies, in which the horizontal axis represents the time in seconds and the vertical axis represents the Number of requests submitted in the given time.

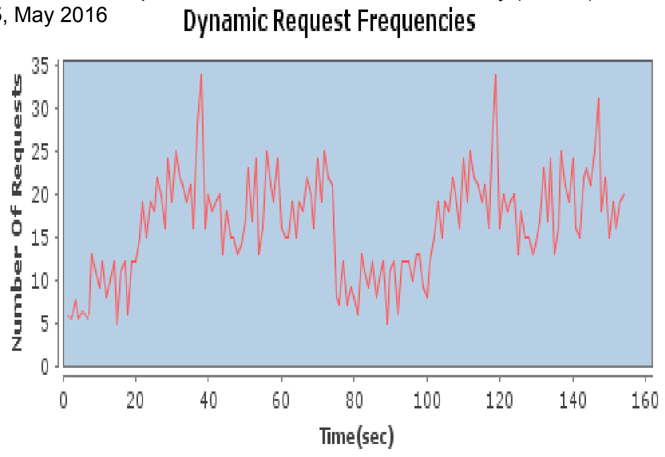

Figure 6.1: Dynamic Request Frequencies

On the basis of the above graph we calculate average request frequencies. Table 1 illustrates that from 1 to 20sec, the average request frequency is 10, and from 20 to 80sec, the average request frequency is 20. Similarly, from 80 to 100sec, the average request frequency is 10, and from 100 to 160sec, the average request frequency is also 20.

Time(sec)	Average Request Frequency
1-20	10
20-40	20
40-60	20
60-80	20
80-100	10
100-120	20
120-140	20
140-160	20

Table 1: Average Request frequency in Time

On the basis of above average request frequencies, we have calculated the average load i.e.
Average Load=(10+20+20+20+10+20+20+20)/8
Average Load=140/8=17
The average workload comes out to be 17 requests per second.

Figure 6.2 shows the pool size comparison of both the strategies, wherein horizontal axis represents Time in milliseconds and vertical axis represents pool size. On the basis of the graphical results we have applied Exponential moving average formula.

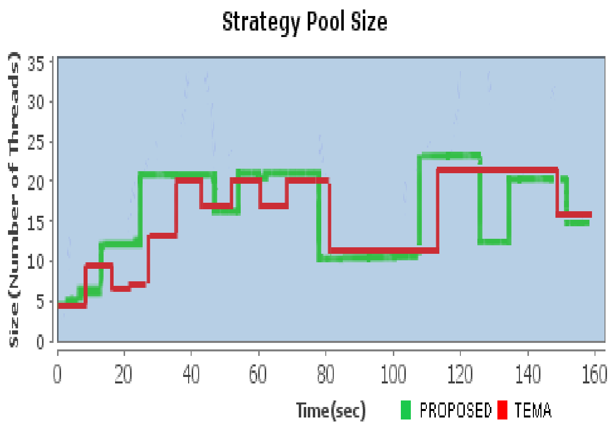

Figure 6.2: Pool Size Comparison

Figure 6.2 shows the pool size i.e. the number of threads in pool, in which the green line shows the predicted pool size of hybrid model while the red line shows the pool size of TEMA. Table 2 is a tabular representation of Figure 6.2 that contains the pool sizes of both Hybrid model and TEMA.

Time (sec)	Pool Size of Hybrid pool	Pool Size of TEMA
1-20	8	8
20-40	18	8
40-60	18	19
60-80	18	17
80-100	9	20
100-120	18	14
120-140	18	10
140-160	18	15

Table 2: Average Pool size

On the basis of the pool size noted in the above-mentioned table, resulting from both strategies, we have calculated the Average Pool size.

Average Pool Size =Sum of Pool size/Total number

i.e. Average Pool Size of Hybrid Model=125/8=16

So, the average pool size of Hybrid Model is 16

Whereas, Average Pool size of TEMA=111/8=13

The average pool size of TEMA comes out to be 13.

In Figure 6.3, the prediction accuracy rate of both the strategies is shown where X-axis shows the pool size of both strategies and Y-axis shows the time in milliseconds, and whereas the green line shows pool size of the hybrid model, the red line shows the pool size of TEMA and the yellow line shows the load.

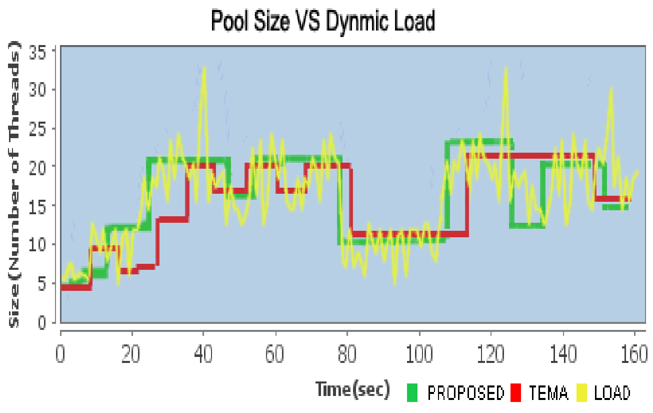

Figure 6.3: Pool Size VS Dynamic Load

We can see that the pool size prediction of our strategy is much more accurate than that of TEMA. The prediction accuracy of TEMA is 76%, whereas the prediction accuracy of Hybrid model is 94%.

So Hybrid Model is free from the shortcoming of wrong prediction, and therefore leads to performance improvement.

Figure 6.4 shows the performance improvement in terms of response time of our strategy against TEMA.

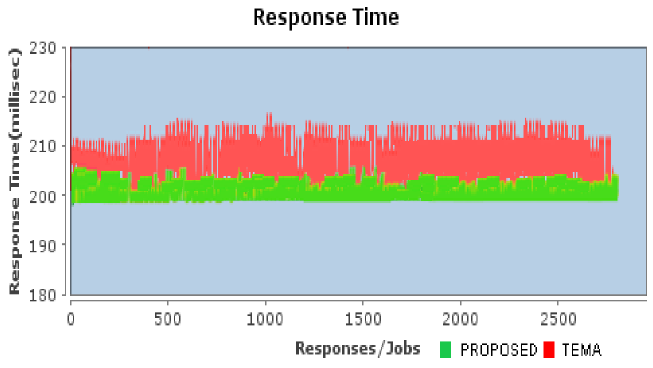

***Figure 6.4*: Response Time Improvement**

The horizontal axis shows the responses/jobs and the vertical axis shows the response time in millisecond i.e. the time taken by jobs to be completed. The graph shows that the response time of TEMA strategy is up to 213ms whereas the response time of Hybrid strategy is up to 203ms. As a result, the overall performance improvement achieved by adopting our strategy comes out to be 23%.

CONCLUSION

In this research paper a complete strategy is presented that dynamically changes the size of a thread pool depending upon request arrival rate, with the aim to enhance the outcome and performance of system. In order to obtain its desired objective, the Hybrid model i.e. prediction and frequency based dynamic thread pool model is presented that precisely predicts the required number of thread, depending upon the incoming request frequencies.

This model also dynamically tunes the amount of threads in the pool by giving consideration to the inactive time period, as with the increase in the inactive time the thread is automatically destroyed.

Consequently, it allows efficient and quick thread creation and destruction.

The Hybrid strategy avoids synchronization overhead by considering the incoming request frequencies instead of thread pool size. By avoiding synchronization overhead, its predictions are much more accurate than those of the TEMA strategy. The resulting accurate prediction eschews the lacking of thread and leads to performance improvement.

The aim of this research was to improve the response time of client's requests, which has been quite successfully attained in the form of simulation results.

VIII. FUTURE WORK

In future, we plan to implement a more efficient prediction scheme on thread pool, for minimizing the response time and increasing pool size accuracy up to 100 %. In addition to it, we also intend to focus over the worker pool with the help of more improved scheme that can efficiently handle the thread creation and destruction.

REFERENCES

[1] D. C. Schmidt and S. Vinoski, "Object Interconnections: Comparing Alternative Programming Techniques for Multithreaded Servers - the Thread-Pool Concurrency Model", C++ Report, SIGS, Vol 8, No 4, April 1996.

[2] Yue-Shan. Chang, W. Lo, Chii-Jet. Wang, Shyan-Ming. Yuan and D. Liang, "Design and Implementation of Multi-Threaded Object Request Broker". International conference on Parallel and Distributed Systems, (Washington, DC, USA) ISSN : 1521-9097, 1998, pp. 740-747.

[3] S. Hafizah, Ab. Hamid, M. Hairul, N. M. Nasir, W. Y. Ming and H. Hassan, "Improving Response Time of Authorization Process of Credit Card System Using Multi-Threading and Shared-Memory Pool Techniques", Journal of Computer Science 4 (2), ISSN 1549-3636, 2008, pp. 151-160.

[4] D. C. Schmidt and F. Kuhns, "An overview of the real-time CORBA specification," IEEE Computer, vol. 33, no. 6, June 2000, pp. 56-63.

[5] D. Xu and B. Bode, "Performance Study and Dynamic Optimization Design for Thread Pool Systems", ;in proc. of the Int. Conf. on Computing Communications and Control Technologies. (Austin, Texas, USA), 2004, pp. 167-174.

[6] Y. Ling, T. Mullen, and X. Lin, "Analysis of optimal thread pool size", ACMSIGOPS Operating Systems Review, 34(2), 2000, pp. 42–55.

[7] N. Chen and P. Lin, "A Dynamic Adjustment Mechanism with Heuristic for Thread Pool in Middleware", 3rd Int. Joint Conf. on Computational Science and Optimization. IEEE Computer Society, (Washington DC, USA), 2010, pp. 324-336.

[8] T. Ogasawara, "Dynamic Thread Count Adaptation for Multiple Services in SMP Environments," IEEE International Conference on Web Services (ICWS '08), Sep 23-26, 2008, pp. 585-592

[9] J.L. Hellerstein, "Configuring resource managers using model fuzzing: A case study of the .NET thread pool" IFIP/IEEE International Symposium on Integrated Network Management (Washington, USA), 2009, pp. 1-8.

[10] S. Ramisetti and R. Wanker, "Design of hierarchical Thread Pool Executor", Second International conference on modeling and Simulation, (kualalampur, Malaysia), 2011, pp. 284-288.

[11] A. Bhattacharya, "Mobile Agent Based Elastic Executor Service", Ninth International Joint Conference on Computer Science and Software Engineering (JCSSE), 2012, pp. 351-356.

[12] J. H. Kim, S. Han, H. Ko and H. Y. Youn, "Prediction- based Dynamic Thread Pool Management of Agent Platform for Ubiquitous Computing", ;in Proc. of UIC 2007, pp. 1098-1107.

[13] D. Kang, S. Han, S. Yoo and S. Park, "Prediction based Dynamic Thread Pool Scheme for Efficient Resource Usage", ;in Proc. of the IEEE 8th Int. Conf. on Computer and Information Technology Workshop, IEEE Computer Society, (Washington, DC, USA), 2008, pp. 159-164.

[14] Kang-Lyul. Lee, H. N. Pham, Hee-seong. Kim, and H. Y. Youn, "A novel predictive and self-Adaptive Dynamic Thread Pool management", Ninth IEEE International Symposium on Parallel and Distributed Processing with applications, (Busan, Korea), 26-28 May, 2011, pp. 26-28.

[15] F. Bahadur, M. Naeem, M. Javed, A. Wahab, “FBOS: Frequency Based Optimization Strategy For Thread Pool System”, The Nucleus 51, No. 1, 2014, pp. 93-107.

[16] F. Bahadur, “ThreadPoolTester Simulation Tool” Availble: https://github.com/faisalsher/ThreadPoolTester, Aug. 12, 2015 [Accessed: Aug. 12, 2015].

Authors’ profile

Ms Summat Nazeer, is pursuing her MS in Computer Science from Hazara University, Mansehra. Her research interests include Simulation tools, Programming in Java, Operating System functions and developing system tools.

Mr. Faisal Bahadur has completed his Master in Computer Science from the Department of Computer Science, University of Peshawar. He completed his Master of Science in Operating Systems from Virtual University Pakistan. Presently, he is serving as Assistant Professor at the Department of Information Technology, Hazara University Mansehra. His research interests include Java Programming, Game design, Efficient Web Server design, Operating Systems development, Console and Windows Application development using C++, Web Server Thread Pool Optimization, concurrency control, multithreading architecture, performance testing, web server workload modeling and Simulation Tools.

Dr. Arif Iqbal Umar is leading the Department of Information Technology, Hazara University Mansehra. He completed his Doctorate in 2010 from Bei-Hang University Beijing, People Republic of China. His research interests include Data Mining, Machine Learning, Databases, Computer Networks and Digital Image Processing.

Mr. Mohammad Abrar Khan is serving as Lecturer at the Institute of Information Technology (IIT), Kohat University of Science and Technology Kohat (KUST). He holds Master of Science in Computer Science from University of Peshawar. In 2005 he joined IIT, KUST, as a lecturer. In 2008 by availing scholarship under Human Resource Development Program, he got registered in PhD program at Brunel University, West London under the supervision of Dr. Thomas Owens. His research interests include Network Simulation, Wi-Fi access control, Network and Information Security, Network programming in C++, Multithreading architectures and Efficient Operating System Design.

.