Computers, Materials & Continua
DOI: 10.32604/cmc.2022.024936
Article

A Perfect Knob to Scale Thread Pool on Runtime

Faisal Bahadur[1,*], Arif Iqbal Umar[1], Insaf Ullah[2], Fahad Algarni[3], Muhammad Asghar Khan[2] and Samih M. Mostafa[4]

[1]Department of Information Technology, Hazara University, Mansehra, 21300, Pakistan
[2]Hamdard Institute of Engineering and Technology, Islamabad Campus, Pakistan
[3]College of Computing and Information Technology, The University of Bisha, Bisha, Saudi Arabia
[4]Faculty of Computers and Information, South Valley University, Qena, 83523, Egypt
*Corresponding Author: Faisal Bahadur. Email: faisal@hu.edu.pk
Received: 05 November 2021; Accepted: 22 December 2021

Abstract: Scalability is one of the utmost nonfunctional requirement of server applications, because it maintains an effective performance parallel to the large fluctuating and sometimes unpredictable workload. In order to achieve scalability, thread pool system (TPS) has been used extensively as a middleware service in server applications. The size of thread pool is the most significant factor, that affects the overall performance of servers. Determining the optimal size of thread pool dynamically on runtime is a challenging problem. The most widely used and simple method to tackle this problem is to keep the size of thread pool equal to the request rate, i.e., the frequency-oriented thread pool (FOTP). The FOTPs are the most widely used TPSs in the industry, because of the implementation simplicity, the negligible overhead and the capability to use in any system. However, the frequency-based schemes only focused on one aspect of changes in the load, and that is the fluctuations in request rate. The request rate alone is an imperfect knob to scale thread pool. Thus, this paper presents a workload profiling based FOTP, that focuses on request size (service time of request) besides the request rate as a knob to scale thread pool on runtime, because we argue that the combination of both truly represents the load fluctuation in server-side applications. We evaluated the results of the proposed system against state of the art TPS of Oracle Corporation (by a client-server-based simulator) and concluded that our system outperformed in terms of both; the response times and throughput.

Keywords: Scalability; performance; middleware; workload profiling; multi-threading; thread pool

1 Introduction

Scalability is one of the utmost non-functional requirements of server-side applications [1], because it increases performance proportional to the resources added in the system on high loads [2]. The scalability is entirely dependent upon the middleware architecture, which means that the

This work is licensed under a Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

middleware must be designed in the view of scaling [3]. A high scalable system can be achieved only, if the designers and software engineers consider scalability at the initial stages of software development life cycle, when architectural choices are made. Thus, middleware architecture makes server-side applications scalable, highly available and highly performant [4,5].

The most significant approach used by most of the middleware in order to improve performance and scalability is TPS [6]. The TPS is a multithreading architecture, that is used as a middleware, that executes incoming tasks (requests) simultaneously. The TPS has been used frequently as a middleware concurrency control service in web servers, application servers, distributed object computing infrastructures, ultra-large-scale (ULS) systems, for better performance.

The key concern in TPS is the dynamic optimization of thread pool size, that determines the overall performance of TPS and improves quality of service [7]. The experimental studies have proved that the most significant factor that effects the overall performance of application servers is the size of thread pool [8]. The TPS-based systems are overloaded either due to the smaller or larger thread pool sizes, instead of adopting an optimal one. Smaller than the optimal pool size wastes processing resources and requests wait most of the time for the availability of thread, that suffers response times and throughput [9]. On the other hand, large pool size than system's capacity increases thread related overheads that suffer response times [9]. These overheads ultimately make the system busy most of the time to manage these overheads instead of letting threads execute requests, thus degrade system performance. These overheads include context switching overhead, scheduling overhead and synchronization overhead. Furthermore, if there are more shared resources between threads, then more synchronization causes additional scheduling overhead that further reduces system performance.

Thus, the key concern in TPS-based systems is the dynamic optimization of thread pool size that avoids too small or too large pool size in order to elude execution time overhead [7]. However, determining the optimal size of thread pool dynamically on runtime is a challenging problem [9]. Therefore, majority of the commercial server applications use a simple frequency oriented TPS [6], that uses fluctuations in the request rate to optimize pool size. Using the request rate as a knob to scale thread pool is widely used because of its implementation simplicity, the negligible overhead and the capability to use it in any system. These systems are either bounded, where thread pool size is specified empirically by minimum and maximum limits at server initialization time [10], or unbounded [6], that avoids specifying upper limit and always keeps thread pool size equal to the request rate.

However, the request rate alone is an imperfect knob to scale thread pool, because request rate is not the only reason that changes the load on the server [11]. There is another way in which the load on the server can be changed besides request rate, and that is the request size (service time of request) [11]. This type of load fluctuation is too common in server-side applications. However, the service times of newly submitted requests are unknown at schedule time [12].

Thus, in this paper, the proposed scheme uses a dynamic workload profiling procedure, that computes the service times of newly entered requests in the system and records these times in a hash table. In every second, the average of service times of requests (submitted to the system) is calculated. The average service time is multiplied by the current request rate to yield a number that represents an optimal thread pool size.

The rest of the paper is organized as follows. Related work is given in Section 2. The design of the proposed system is presented in Section 3. Section 4 is a validation of the proposed system. And finally, the conclusion and future work are presented in Section 5.

2 Related Work

This section explores the tuning strategies for thread pool size optimization presented in the past.

The system resource based TPS has been presented in [13], where CPU and memory utilization metrics are used to optimize thread pool size. However, the system resource metric is a low-level metric that cannot be used as an actual pointer to overload condition. Moreover, successively measuring different types of resource usages after short intervals consumes computational resources and decrease machine capacity [14].

The response time is considered as a performance metric in [15,16] for TPS optimization. The optimization policies are based on response time observations. After specified time intervals, an algorithm compares successive response times on specific pool size and either increases pool size if response times are higher or decreases pool size if response times are better. However, the response time metric becomes vague and useless for server applications that have requests of different processing times, e.g., multi-tier web applications. Hence, policies [17] have been suggested to smooth response time metric to make it effective in multi-tier applications.

The average idle time (AIT) of queued requests has been used in [18,19] to optimize the pool size, however, periodically calculating AIT involves stalling the request queue that increases synchronization overhead that effects system performance.

The throughput metric is used in [20] to optimize the pool size. On throughput fall, the algorithm gradually increases pool size until throughput stability. However, assessing only throughput fall to tune the pool size may affect the response times of the clients.

The prediction based TPSs have been presented in [21–23], that predict an expected thread pool size that may require in the future, so that, the pool size can be configured in priori. An obvious limitation of all the prediction-based schemes presented is that, the predictions might be inaccurate due the unpredictable workload on the servers.

The fuzzy [24] and heuristic approaches [25] have been attempted in TPS optimization. However, systems having variation in the nature and size of requests can affect these algorithms.

The frequency based schemes [26–32] are the most widely used TPSs in the industry [6], that optimizes thread pool size based on request rate on the server. The Oracle Corporation's Java5.0 provides ThreadPoolExecutor [26], that optimizes pool size on the basis of client's request rate (i.e., frequency). The pool size increases gradually on high request rate up to a maximum limit, that defines a boundary that cannot be exceeded. The maximum limit must be defined initially and can't be changed on run time. If the maximum limit is initialized with an unbounded option (Integer.MAX_VALUE), then ThreadPoolExecutor behaves as an unbounded TPS, where an unlimited upscaling of thread pool size is considered. The Apache Software Foundation provides a TPS called Watermark Executor [27] to boost up the performance of web services. This TPS is an extension of ThreadPoolExecutor [26] in the form of a balancing act between request frequency and queue draining speed. If the request queue gets to a certain size (watermark), more threads will be added until the maximum number of threads. This is different from ThreadPoolExecutor [26] that starts adding threads as soon as all the other threads are used. A real time middleware is presented in [28], that optimized the size of thread pool by linear approach. The middleware is presented for Cyber Physical Systems (CPS). The thread pool size is optimized based on the incoming load condition (frequency). Similarly, another real time middleware is presented in [29] that utilized a dynamic thread pool that receives, processes and retransmits the requests arrived in a distributed IoT environment. This scheme optimized thread pool size by the rate of received packets (frequency) and the retransmission time of packets. A distributed FOTP is

presented in [30] for clusters, that uses non-blocking queue that drives thread safety not from locks but from atomic operation. This design was targeted to only distributed shared memory systems. The TPSs at each node of the cluster are optimized on the basis of request rate on corresponding node. Another distributed FOTP is presented in [31], where FOTPs running at the backend servers are optimized on request rate, and memory utilization is assessed to detect an overload condition on backend servers. The queuing theory based TPS is presented in [32], however, an offline system profiling strategy is adopted and run the offline profiling strategy on n-tier system for 100 min in order to generate near optimal thread pool size of each tier. The offline system profiling procedure is tedious, cumbersome, and impractical. Moreover, this approach is ineffective due to the dynamics and unpredictability of the workload, especially in the cloud, hence it needs to regenerate thread pool size when workload characteristics change.

None of the frequency based TPS considered to calculate the combination of request size (service time) and request rate on run time to represent a true load on the system, because the real load fluctuation is truly represented by these two factors which must be calculated on runtime. Hence, this paper presents a dynamic workload profiling based FOTP, that computes the service times of newly entered requests in the system (on runtime) and records these times in a hash table. In every second, the average of service times of requests (submitted to the system) is calculated and multiplied by the current request rate to determine an optimal thread pool size.

3 System Architecture of Proposed FOTP

This section discusses the system architecture of the dynamic workload profiling based FOTP. The FOTP is composed of collection of components shown in Fig. 1.

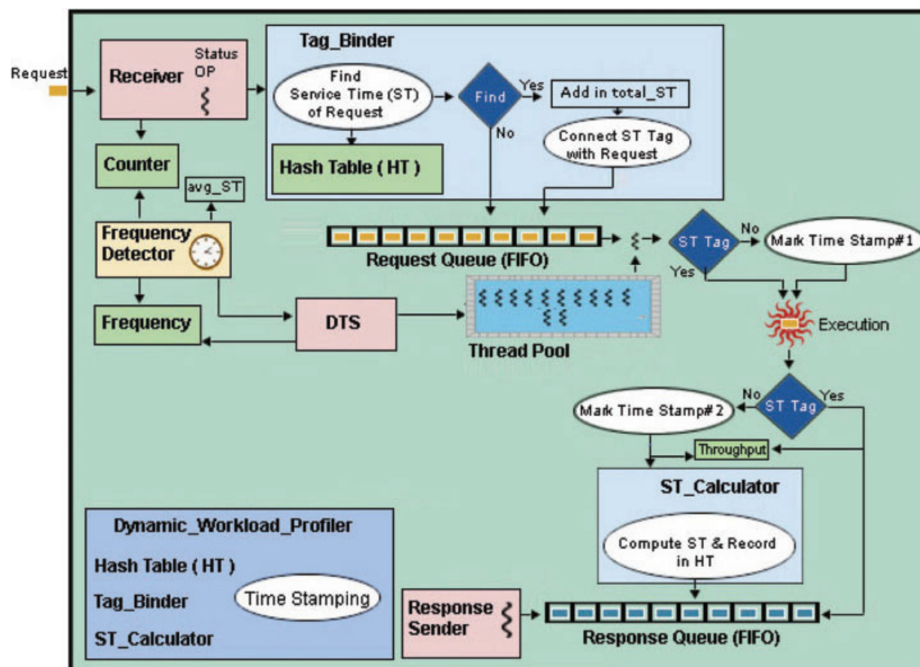

Figure 1: System architecture of workload profiling based FOTP running in a multi-threaded server

The Receiver is a thread, that receives client's requests for processing. On every request arrival, it increments the Counter object that is used to keep track of the request rate per second on FOTP. After every second, the value of the Counter is repeatedly saved and reset to zero by the Frequency Detector (timer). In this way, both Counter and Frequency Detector keep track of the request rate per second on FOTP. The Receiver forwards request to the Tag_Binder for further processing, that uses the request's identifier (ID) as an index into the hash table to fetch the service time of requests. If it is found, then it adds the service time to the variable named total_ST (i.e., total service time), binds a service time tag with the request and finally push the request in the request queue. If request's ID does not exist in the hash table, then the request is simply pushed into the request queue. Eq. (1) is used to calculate the sum of service times of all requests entered in the system per second, which is represented by total_ST variable.

$$\text{total_ST} = \sum_{k=1}^{Counter} Service\ Time(k) \tag{1}$$

The total_ST is accessed by the Frequency Detector to calculate the average of service time per second. This is a timer thread, that activates after every second, and keeps track of the frequency of change in the real load every second on the system. The real load comprises of the combination of request rate and the service times of all requests entered per second in the system. The request rate per second is tracked by the Counter and saved in the Frequency object. The service times of all requests entered per second in the system is tracked by the total_ST variable. Eq. (2) is used by the Frequency Detector to calculate the average of service times of all requests entered per second in the system.

$$\text{avg}_{\text{ST}} = total_ST/Counter \tag{2}$$

For all requests entered per second in the system from 1 to Counter, the avg_ST (i.e., average service time) is calculated by dividing the sum of their service time by Counter. However, the service time of requests are unknown when they first time enter in the system. Thus, the dynamic workload profiling procedure computes the service times of newly entered requests by time stamping and records the service times in a hash table. The requests are time stamped at two stages, right before the request execution (StartTime), and finally, after request completion (EndTime). The stamps are only marked on those requests that are entered first time in the system and their service time is not present in the hash table. This table maintains the service times of each request. The request_ID is used as an indexed into the hash table by a hash function, that keeps the service time of request if index exists. By using the request_ID as an index, the average search time complexity of any element is O (1). The untagged requests are marked with the StartTime stamp by thread just before their execution. After request completion, these untagged requests are again marked with the EndTime stamp and handover to the ST_Calculator (i.e., service time calculator). Eq. (3) is used by the ST_Calculator to calculates the service time of request. The ST_Calculator records the service time in the hash table. Finally, it enqueues the request in the Response Queue.

$$Service\ Time(Request) = EndTime(Request) - StartTime(Request) \tag{3}$$

When Frequency Detector runs, it saves the Counter in the Frequency object, computes the avg_ST, resets the total_ST and Counter to zero and finally, it runs the dynamic tuning strategy (DTS) thread, that is responsible to optimize the size of thread pool. The thread pool is a dynamic linked list, that holds threads to execute client's request. The thread pool expands dynamically, when threads are added in it by the DTS, when needed. It also shrinks, when request rate falls down from high to low request rate, and some threads becomes idle in the pool till certain specified threshold time. Every

thread of thread pool is equipped with a timer, that starts only when thread is idle in the pool. The timer destroys corresponding thread, if it is idle for threshold time (500 milliseconds), in order to reduce the pool size.

The DTS thread is run by the Frequency Detector every second before finishing its own execution. It dynamically optimizes thread pool size by assessing the request rate and service times of queued requests. When it starts running for the first time, it keeps its pool size parallel to the request rate, however, in the subsequent runs, it also assesses the workload's service times.

The avg_ST (Eq. (2)) is passed to the DTS by the Frequency Detector and the current request rate is read by the DTS from Frequency object. The DTS maintains an appropriate number of threads in the thread pool by assessing avg_ST (Eq. (2)) and request rate. Eq. (4) is used to calculate an appropriate number of threads in thread pool by the DTS.

$$Pool_{Size} = \lambda * \text{avg}_{\text{ST}} \tag{4}$$

where, the λ in Eq. (4) is the request rate per second.

Suppose that, the request rate is 10 requests per second, where all requests have service times of 2 s, then according to Eq. (4), the $Pool_{Size} = 10*2 = 20$. In case, if all requests have service time of 0.5 s, then $Pool_{Size} = 10*0.5 = 5$. Since 5 threads can handle 10 requests (of 0.5 s) in one second, hence this will not affect the throughput. However, the response times of 5 out of 10 requests will be compromised, i.e., 5 requests will have to wait for 0.5 s for thread availability and consecutive request arrival will further increase the wait times of requests. Hence, for small service timed workload (having avg_ST less than or equal to 1 s), we only consider λ (i.e., request arrival rate) to set a pool size, so that 10 threads can be assigned to 10 requests to gain response time improvement. Hence, Eq. (4) is used to tune the pool size only when $\sum_{k=1}^{n} Service\ Time(k)/n$ is greater than 1, that represents high service time intensity workload of greater than 1 s. Otherwise, we only consider λ as the pool size. In short, this paper uses the following if-else construct to tune the pool size.

If $(\sum_{k=1}^{n} Service\ Time(k)/n\ ) <= 1$ then $Pool_{Size} = \lambda$

Else $Pool_{Size} = \lambda * \sum_{k=1}^{n} Service\ Time(k)/n$

The algorithm of DTS to tune the pool size is given in Fig. 2.

```
DTS (input: request_rate, avg_ST, pool_size, evaluated_ST)

BEGIN
   If (pool_size < request_rate AND avg_ST <=1)
     make pool_size=request_rate
   Else if (pool_size >= request_rate AND avg_ST>1 AND avg_ST! =evaluted_ST)
    Begin
        make pool_size= request_rate *avg_ST
        evaluted_ST=avg_ST
    End
END
```

Figure 2: Algorithm of dynamic tuning strategy

At the start of the algorithm, the pool size is set to the request rate, if workload (avg_ST) per second is smaller than or equal to 1 s. Otherwise, if the workload is greater than 1 s, then pool size is set to the product of the request rate and average service times of requests.

4 Validating the Proposed FOTP

In this section, we compared the performance of workload profiling based FOTP with ThreadPoolExecutor [26] on different workloads. The ThreadPoolExecutor uses request rate to optimize thread pool size. Both of the TPSs are initialized with 10 threads. This test has been performed on a single system, through jPoolRunner [33] simulation toolkit. We embedded both of the TPSs in the server-tier of simulation system through its extension framework. The client and the server tiers of the simulation system are running on the localhost i.e., single machine.

The workload used for this test is depicted Tab. 1. The workload is mostly I/O bound, because we are performing this simulation test on a single system. The dynamic workload is kept small to protect seizing processing resources. The 50% of workload is high I/O intensive (2 s) in order to show the effectiveness of proposed FOTP system over pool size that improves system performance. Fig. 3 depicts the load generated for 1 min, that follows poisson distribution, with $\lambda = 100$. In Fig. 3 the x axis represents time in seconds and y axis represents total number of requests sent on specific time.

Table 1: Workload detail

Requests	ID	Access frequency	Service time
100 kb file	1	20%	≈300 ms
1000 kb	2	20%	≈400 ms
2000 kb	3	50%	≈2000 ms
Low CPU-intensive	4	10%	≈40 ms

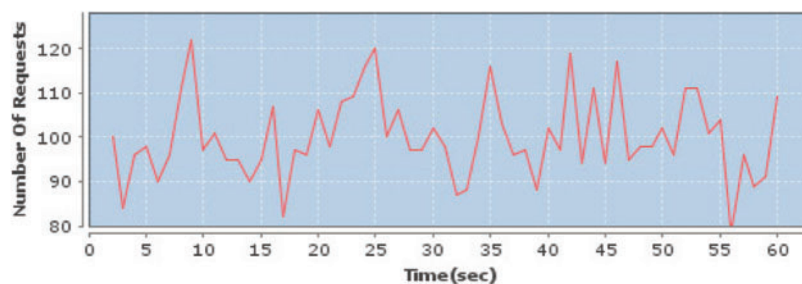

Figure 3: The generated load on the server

Fig. 4 presents a comparative analysis of pool sizes maintained by both of the TPSs in the whole test run. The ThreadPoolExecutor kept its pool size equals to the highest request rate generated during the test run. Initially, the pool size is 10, afterward, the pool size is parallel to the next highest request rate. The proposed FOTP kept its pool size equal to the request rate at the start, as the service times are unknown to the FOTP. The service times of four different types of requests are measured and recorded by the dynamic workload profiler on the very first arrival in the first second, hence the pool size raised (≈114) further in the next second due to assessing both request rate and service time of requests. We can see in the Fig. 4, that on 10th second, the FOTP increased the pool size more than ThreadPoolExecutor. The FOTP created threads more than the request rate due to the large service times of requests, hence all of the requests got a proper response.

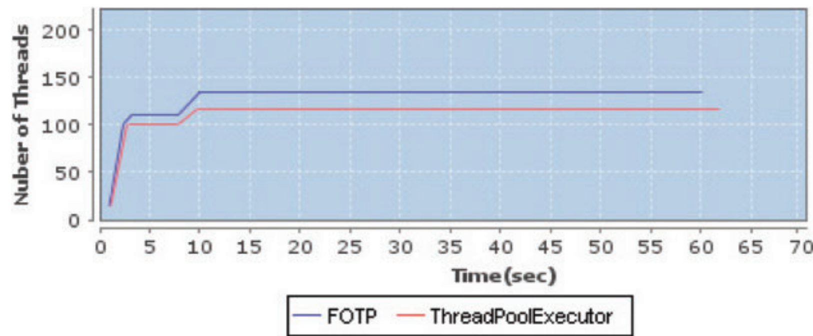

Figure 4: Comparative analysis of pool sizes

Fig. 5 is a comparative analysis of throughputs of both schemes, which shows that the FOTP produces higher throughput than ThreadPoolExecutor due to its appropriate pool size in the whole test run. It can be seen in the figure, that the ThreadPoolExecutor took a little bit long time to complete than FOTP. The FOTP produces all responses in less amount of time (60 s) than ThreadPoolExecutor, because of its appropriate pool size. In case of FOTP, all of the requests almost got a proper response by available threads in the pool, hence the average throughput is 98 responses per second in Fig. 6, whereas, the ThreadPoolExecutor produces 80 responses per second.

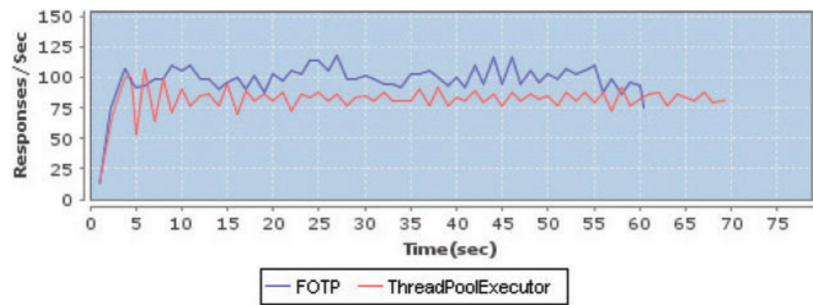

Figure 5: Comparative analysis of throughputs

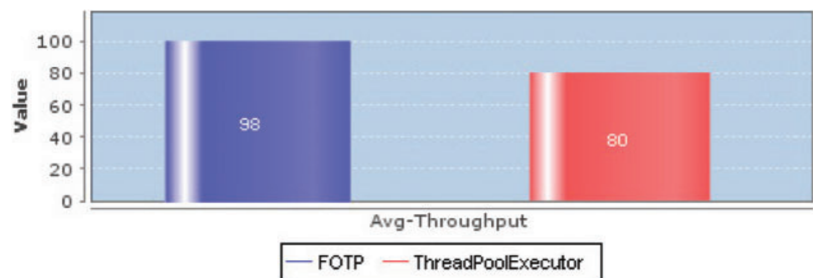

Figure 6: Comparative analysis of average responses produced per second by both schemes

Fig. 7 presents a comparative analysis of response times of both systems. The FOTP reduces response times as compared to ThreadPoolExecutor, because it provides an optimal number of threads in the pool that picks and executes requests in performance efficient manner. ThreadPoolExecutor created small number of threads in the pool, as a result, the requests have to wait more in the queue

for thread availability, hence, response times increased ultimately. The 50th, 90th and 95th percentiles of FOTP are around 2 s. This is due to the workload characteristic of service time, because 50% workload has service time of 2 s. In case of ThreadPoolExecutor, the percentile response times have increased due to the waiting of requests in the request queue due to inappropriate quantity of threads in the pool.

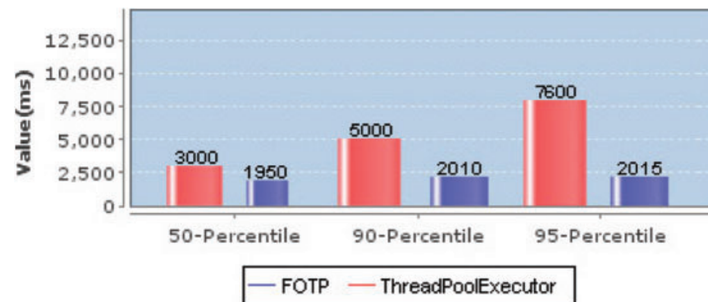

Figure 7: Comparative analysis of response times percentiles produced by both schemes

5 Conclusion and Future Work

This paper presented a workload profiling based frequency-oriented thread pool system, that focused on the combination of request size (service time) and the request rate to represent the real load on the server. The real load is used to set an optimal thread pool size of frequency-oriented thread pool system on runtime. Thus, the paper presented a combination of both; the request size (service time) and the request rate, as a knob to scale thread pool on runtime. We evaluated the proposed scheme against a request rate based TPS on high service time workload, and we found that, the proposed scheme is creating an optimal pool size. Hence, as compared to the request rate based TPS, the average throughput of the proposed scheme is increased by 22.5% and the response time reduction is 35% for 50th percentile, 59.8% for 90th percentile and 73.5% for 95th percentile.

In the future, we will implement an overload management scheme in the proposed system. Moreover, we will provide a distributed form of the proposed scheme for distributed server applications.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] M. Haque, M. Asikuzzaman, I. U. Khan, I. H. Ra, M. Hossain *et al.,* "Comparative study of IoT-based topology maintenance protocol in a wireless sensor network for structural health monitoring," *Remote Sensing*, vol. 12, no. 15, pp. 2358, 2020.

[2] T. Maqsood, O. Khalid, R. Irfan, S. A. Madani and S. U. Khan, "Scalability issues in online social networks," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–42, 2016.

[3] G. Denaro, A. Polini and W. Emmerich, "Early performance testing of distributed software applications," in *Proc. of the 4th Int. Workshop on Software and Performance*, Redwood Shores, California, USA, pp. 94–103, 2004.

[4] M. M. Molla and S. I. Ahamed, "A survey of middleware for sensor network and challenges," in *Proc. ICPPW'06*, Columbus, OH, USA, pp. 6, 2006.

[5] G. Denaro, A. Polini and W. Emmerich, "Performance testing of distributed component architectures," in *Proc. Testing Commercial-off-the-Shelf Components and Systems*, Berlin, Heidelberg, Springer, pp. 293–314, 2005.
[6] M. G. Valls and C. C. Urrego, "Improving service time with a multicore aware middleware," in *Proc. Symp. on Applied Computing*, Marrakech, Morocco, pp. 1548–1553, 2017.
[7] W. Liu, B. Tieman, R. Kettimuthu and I. Foster, "A data transfer framework for large-scale science experiments," in *Proc. of the 19th ACM Int. Symp. on High Performance Distributed Computing*, Chicago, Illinois, pp. 717–724, 2010.
[8] W. Ye, Y. Tong and C. Cao, "A kalman filter based hill-climbing strategy for application server configuration," in *Proc. of the 2018 IEEE SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*, Guangzhou, China, pp. 1524–1531, 2018.
[9] Z. Wang, P. Subedi, M. Dorier, P. E. Davis and M. Parashar, "Staging based task execution for data-driven, in-situ scientific workflows," in *Proc. of the 2020 IEEE Int. Conf. on Cluster Computing (CLUSTER)*, Kobe, Japan, pp. 209–220, 2020.
[10] N. Jia, R. Wang, M. Li, Y. Guan and F. Zhou, "Towards the concurrent optimization of the server: A case study on sport health simulation," *Complexity*, vol. 2021, pp. 1–13, 2021.
[11] A. Gandhi, M. Harchol-Balter, R. Raghunathan and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, pp. 1–26, 2012.
[12] J. Mace, P. Bodik, M. Musuvathi, R. Fonseca and K. Varadarajan, "2dfq: Two-dimensional fair queuing for multi-tenant cloud services," in *Proc. of the 2016 ACM SIGCOMM Conf.*, Florianopolis, Brazil, pp. 144–159, 2016.
[13] F. Muscarella, "Method and apparatus for dynamically adjusting thread pool," US Patent No. 8185906B2, U.S. Patent and Trademark Office, 2012.
[14] S. Ahn, J. Kim, E. Lim and S. Kang, "Soft memory box: A virtual shared memory framework for fast deep neural network training in distributed high-performance computing," *IEEE Access*, vol. 6, pp. 26493–26504, 2018.
[15] N. J. Chen and P. Lin, "A dynamic adjustment mechanism with heuristic for thread pool in middleware," in *Proc. of the 2010 3rd Int. Joint Conf. on Computational Science and Optimization*, Huangshan, China, pp. 324–336, 2010.
[16] N. Costa, M. Jayasinghey, A. Atukoralez, S. S. Abeysinghex, S. Perera *et al.,* "ADAPT-T: An adaptive algorithm for auto-tuning worker thread pool size in application servers," in *Proc. of the 2019 IEEE Symp. on Computers and Communications (ISCC)*, Barcelona, Spain, pp. 1–6, 2019.
[17] S. Guo, L. Cui, S. Liu and C. Pu, "Combining fine-grained analysis and scheduling to smooth response time fluctuations in multi-tier services," in *Proc. of the 2015 IEEE Int. Conf. on Services Computing*, New York, NY, USA, pp. 750–753, 2015.
[18] D. Xu and B. Bode, "Performance study and dynamic optimization design for thread pool systems," in *Proc. CCCT2004*, Austin, USA, pp. 167–174, 2004.
[19] T. Ogasawara, "Dynamic thread count adaptation for multiple services in SMP environments," in *Proc. ICWS*, Beijing, China, pp. 585–592, 2008.
[20] J. L. Hellerstein, V. Morrison and E. Eilebrecht, "Applying control theory in the real world: Experience with building a controller for the net thread pool," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 3, pp. 38–42, 2010.
[21] 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 the Int. Conf. on Ubiquitous Intelligence and Computing*, Berlin, Heidelberg, pp. 1098–1107, 2007.
[22] D. Kang, S. Han, S. Yoo and S. Park, "Prediction-based dynamic thread pool scheme for efficient resource usage," in *Proc. of the 2008 IEEE 8th Int. Conf. on Computer and Information Technology Workshops*, Sydney, NSW, Australia, pp. 159–164, 2008.

[23] K. L. Lee, H. N. Pham, H. S. Kim, H. Y. Youn and O. Song, “A novel predictive and self-adaptive dynamic thread pool management,” in *Proc. of the 9th IEEE Int. Symp. on Parallel and Distributed Processing with Applications*, Busan, Korea (South), pp. 93–98, 2011.
[24] J. L. Hellerstein, “Configuring resource managers using model fuzzing: A case study of the .NET thread pool,” in *Proc. of the 2009 IFIP/IEEE Int. Symp. on Integrated Network Management*, NY, USA, pp. 1–8, 2009.
[25] D. L. Freire, R. Z. Frantz and F. R. Frantz, “Towards optimal thread pool configuration for run-time systems of integration platforms,” *International Journal of Computer Applications in Technology*, vol. 62, no. 2, pp. 129–147, 2020.
[26] Class ThreadPoolExecutor. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html (19th October 2021).
[27] Class WaterMarkExecutor. [Online]. Available: https://axis.apache.org/axis2/java/core/apidocs/org/apache/axis2/transport/base/threads/watermark/WaterMarkExecutor.html (19th October 2021).
[28] M. García-Valls, C. Calva-Urrego, A. Juan and A. Alonso, “Adjusting middleware knobs to assess scalability limits of distributed cyber-physical systems,” *Computer Standards & Interfaces*, vol. 51, pp. 95–103, 2017.
[29] S. Jeon and I. Jung, “Experimental evaluation of improved IoT middleware for flexible performance and efficient connectivity,” *Ad Hoc Networks*, vol. 70, pp. 61–72, 2018.
[30] S. Ramisetti and R. Wankar, “Design of hierarchical thread pool executor for dsm,” in *Proc. of the 2nd Int. Conf. on Intelligent Systems, Modelling and Simulation*, Phnom Penh, Cambodia, pp. 284–288, 2011.
[31] M. García-Valls, “A proposal for cost-effective server usage in CPS in the presence of dynamic client requests,” in *Proc. ISORC*, York, UK, pp. 19–26, 2016.
[32] Q. Wang, S. Zhang, Y. Kanemasa, C. Pu, B. Palanisamy *et al.,* “Optimizing n-tier application scalability in the cloud: A study of soft resource allocation,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 4, no. 2, pp. 1–27, 2019.
[33] jPoolRunnerTM. “Performance testing simulation tool,” [Online]. Available: http://jpoolrunner.net/ (19th October 2021).