Sensors & Transducers, Vol. 173, Issue 6, June 2014, pp. 189-196

www.sensorsportal.com

Sensors & Transducers

http://www.sensorsportal.com

Analytical Modeling and Performance Optimization of a Proactor-based Server for Wireless Sensor Network

YUE Guo-dong, * XU Zheng, WANG Li-ding, LIU Chong, CHEN Yi
Key Laboratory for Micro/Nano Technology and System of Liaoning Province,
Dalian University of Technology, Dalian 116085, China
* Tel.: + 86-411-84707713-2193, fax: + 86-411-84707940
* E-mail: xuzheng@dlut.edu.cn

Received: 12 May 2014 /Accepted: 30 May 2014 /Published: 30 June 2014

Abstract: The Proactor-based server, which effectively encapsulates the asynchronous mechanisms provided by an operating system, can provide a high performance of concurrency. A queuing network model of the server is presented. And based on the model, an algorithm is proposed to recursively determine the maximal throughput on a specific server and find the best configuration of server for a given workload. Then the algorithm is experimentally verified by means of comparison on the number of threads per node, queue length, response time and so on. In the end, the use of the model is demonstrated to guide key provisioning and configuration decisions using several examples. *Copyright © 2014 IFSA Publishing, S. L.*

Keywords: Proactor pattern, Performance optimization, Queuing network model, Wireless sensor network.

1. Introduction

Wireless sensor network (WSN) has gained worldwide attention in recent years because of their potential to facilitate data acquisition and scientific studies. A WSN includes a large number of sensor nodes, which are densely deployed either inside the phenomenon or very close to it. Data collected by nodes are transferred into the data center, which is responsible for data collection, storage and analysis, through the gateways [1]. However, as the prevalence of WSN to monitor physical or environmental conditions, it is becoming evident that the data center must be offered with superior performance in order to make connections with a growing number of gateways and receive more data.

A critical component of data center is a network server. In the industry, hundreds of gateways are connected with the server and transfer hundreds of megabytes of data to the server at the same time. In order to fulfill such high workload demands, it is inevitable that servers be equipped with the capability to process multiple requests concurrently. Concurrency may be implemented in a server using the synchronous or asynchronous capabilities provided by the underlying operation system (OS). Compared to multi-thread and multi-process software architecture, the asynchronous mechanisms may be attractive because of the benefit of concurrency while alleviating much of the overhead and complexity of multi-threading [2]. The Proactor pattern in middleware, which effectively encapsulates the asynchronous mechanisms supported by the OS, can be used to implement a high performance server. Thus, a Proactor-based server is very applicable to meet the performance demand in WSN.

Due to the high performance expectation and the high utilization of server, it is imperative that service performance should be analyzed prior to deployment and during the running. If performance is measured

once the service is implemented, it is often too late and expensive to take corrective action at this stage when the target performance cannot be met. Besides, it is beneficial for calculations to automatically look for the proper number of threads to cope with a given load while the system is running. Model-based analysis is an attractive approach to conduct such design-time performance analysis and runtime configuration adjustment.

Queuing network models have been often adopted as models for the evaluation of software performance [3]. Nossenson R. et al [4] introduced an N-Busrst/G/1 model with heavy-tailed service-time distribution, which captured many of the issues that affected web servers as observed by empirical studies. Cao J. et al [5] presented an M/G/1/K*PS queuing model of a web server to obtain closed form expressions for web server performance metrics such as average response time, throughput and blocking probability. Praphamontripong U et al [6] described a Proactor-based queuing model for the analysis of an asynchronous web server by means of the model decomposition strategy.

The above approaches strive to estimate the software performance during design time by checking the model for inconsistencies and by optimizing the model's design. However, optimizing the design and minimizing resource conflicts are not enough to yield optimal performance [7]. There is a supplementary need to balance resources utilization in order to maximize the benefit per resource ratio and result in better system performance. A different approach is to address the problem during run time instead of design time. Osogami T. et al [8] proposed that an algorithm referred as Quick Optimization *via* Guessing (QOG) quickly selected one of nearly best configurations with high probability for a web system. Xi B et al [9] proposed a Smart Hill-Climbing algorithm using ideas of importance sampling and Latin Hypercube Sampling (LHS) to find an optimal configuration for a given application as a black-box optimization problem. Hlavacs H et al [10] described an optimization tool to find the optimal number of threads for multi-thread data-flow software. However, the research on the runtime optimization of a Proactor-based server in WSN is still not reported.

In this paper, we describe the Proactor-based server modeled as queuing network, and develop an algorithm to find an optimal configuration of threads on a given host and estimate the server performance. Then the number of threads for each node and other performance metric which result from the algorithm are verified through measurement when giving an external workload. At last, we illustrate how the model can be used to guide configuration and provisioning decisions with several examples.

2. Theoretical Principle

2.1. Proactor Pattern

The Proactor pattern is a software architectural pattern for event handling, which is used to describe how to initiate, receive, demultiplex, dispatch and process events in network systems [2]. By means of the asynchronous call feature underlying the OS platform, it has been developed to support many simultaneous user requests.

Fig. 1 describes the general process of the Proactor pattern, which is to wait for an event to occur and then initiate the appropriate operation. Handler could initiate an appropriate asynchronous operation without blocking its caller's thread of control and then register itself at the Completion Dispatcher to wait for an event completion. Once the asynchronous operation completes, the OS notifies the Completion Dispatcher and then the Completion Dispatcher notifies the handler. When the operation finishes execution, the Proactor demultiplexes the completion event and dispatches it to an appropriate event handler for subsequent processing of the results of the operation.

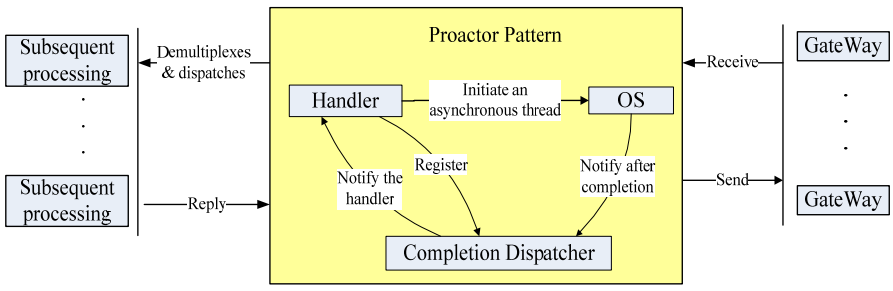


Fig. 1. Frame of Proactor pattern.

2.2. Optimization Idea

The software is modeled as multi-thread data flow model for process event-based data in the form of packets. The functionality of each operation in the software is internally represented by interconnected nodes. Queues buffer the output for succeeding nodes and incoming tokens pass the graph and visit each

node once. Since these nodes technically are lightweight processes (threads), they can be replicated for splitting the load.

All nodes in software require certain hardware resource. Accordingly, replication of nodes is restricted. The optimal configuration of threads for a certain host is needed in order to maximize the benefit. And the main idea is to sequentially adjust the number of threads for nodes which do not meet the constraints until an optimization goal is reached. It is reflected in two performances.

Throughput. With optimization towards throughput, the goal is to gain the maximal throughput and the arrival rate is varied from the initial rate in steps of about *exInc* until one node breaks through the given constraints. Then the number of the corresponding threads is increased as long as constraints are satisfied. Again, the arrival rate will be increased and optimization goes on as long as hardware limit (CPU cores in this paper) is reached. For the maximum possible number of threads, the highest possible throughput is determined.

Consolidation. With optimization towards consolidation, a desired arrival rate is given and the goal is to determine the minimum number of threads required to meet the given constraints. With the constant external arrival rate, the minimum initial number of threads is determined to ensure that all nodes are below a predefined utilization rate threshold. Only when a constraint node doesn't meet the constraints, the number of its threads will be incremented for splitting load among available cores as evenly as possible.

3. Performance Analysis Methodology

3.1. Description of the Model

Fig. 2 shows a Proactor-based server modeled as queuing network. It is assumed that the requests of each type arrive according to a Poisson distribution, with λ_i denoting the arrival rate of type i requests (M/././.) [5]. The size of the event handler pool of type i requests is denoted n_i (./././n_i). The service time of an asynchronous operation for each request type follows an exponential distribution (./M/./.), with the parameter of type m requests denoted μ_m. The capacity of the common completion queue is denoted n and the capacity of the separate completion queue of type i requests is denoted m_i. The demuptiplexing time of the Proactor is exponentially distributed with parameter s. The service times of the completion event handlers are exponentially distributed, with the service time of event handler i denoted γ_i.

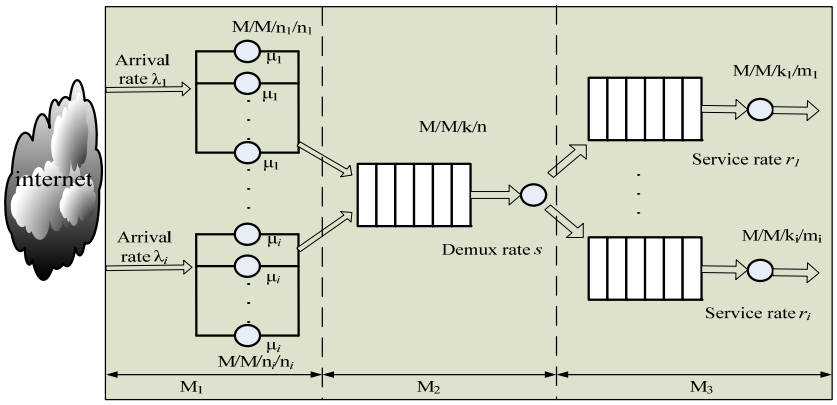


Fig. 2. A Practor-based server modeled as queuing network.

Each event handler pool which handles asynchronous operations for a single request type is modeled as a multi-server processing station with no queuing. The capacity of the queue of type i requests is denoted n_i (./././n_i) equal to the size of event handler pool. These servers feed completion events to the common completion event queue and they block if there is no space available in the common completion queue. The operation of demultiplexing the completion events is conducted by k servers, which accept the completion events from the common completion queue and dispatch them to the appropriate separate completion queue depending on the request type that generates the completion event. Because the scheduling used for demultiplexing is First-Come, First-Served (FCFS), the demultiplexing operation blocks if the completion queue to which the event is to be dispatched is full. For example, if the current completion event to be demultiplexed and dispatched is of type i and the type i completion queue has k_i completion events, the demultiplexing operation blocks. The completion event queue of each event type feeds the corresponding completion handler which completes the processing. Thus, the three steps involved in fulfilling a single client request are: (i) asynchronous operation, (ii) completion event demultiplexing and (iii) dispatching and completion event handling.

3.2. Mathematical Analysis

In the above three steps, we assume the external arrival rate of each node is λ and the service rate of each server is μ, the size of servers in each step is n ($n \geq 1$) and the queue capacity is defined by parameter m ($m \geq n$). So, we can gain the performance metrics for each request type.

There is a race between λ and μ in the sense. Under the assumption $\lambda \leq \mu$ and given number of servers n, the following for the utilization rate ρ must hold:

$$\rho = \frac{\lambda}{n\mu} \leq 1, \tag{1}$$

which leads to a stable system.

The utilization rate given in Equation (1) is the base for most other measures. The probability of k tokens in the queuing system is given by

$$P_k = \begin{cases} \frac{n^k \rho^k}{k!} P_0 & 0 \leq k < n \\ \frac{n^n \rho^k}{n!} P_0 & n \leq k \leq m \end{cases}, \tag{2}$$

where P_0 is the probability of no token in the system. And P_0 is given by

$$P_0 = \begin{cases} \left[\sum_{k=0}^{n-1} \frac{(n\rho)^k}{k!} + \frac{(n\rho)^n}{n!} \frac{1-\rho^{m-n+1}}{1-\rho} \right]^{-1} & \rho \neq 1 \\ \left[\sum_{k=0}^{n-1} \frac{n^k}{k!} + \frac{n^n}{n!}(m-n+1) \right]^{-1} & \rho = 1 \end{cases}, \tag{3}$$

If an event handler is unavailable or the queue is full, an incoming token will be rejected. And the expected loss probability, which is the average probability, is denoted by

$$L_m = P_m = \frac{n^n \rho^m}{n!} P_0, \tag{4}$$

The effective arrival rate is

$$\lambda' = \lambda(1 - L_m), \tag{5}$$

For the common completion event queue, the total input rate is given by:

$$\lambda_2 = \lambda'_{1,1} + \lambda'_{1,2} + \cdots + \lambda'_{1,m}, \tag{6}$$

where $\lambda'_{1,i}$ is the effective average rate at which type i requests are processed by the event handler pool.

For the separate completion queue of type i requests, the input rate is given by:

$$\lambda_{3,i} = \frac{\lambda'_{1,i}}{\lambda'_{1,1} + \cdots + \lambda'_{1,m}} \lambda'_2, \tag{7}$$

where λ'_2 is the effective demultiplexing rate.

The expected throughput, which is the average processing rate of the asynchronous server, is

$$T = \lambda'_{3,1} + \lambda'_{3,2} + \cdots + \lambda'_{3,i}, \tag{8}$$

where $\lambda'_{3,i}$ is the effective completion rate of completion event handler i.

The loss rate ε is

$$\varepsilon = \lambda P_m, \tag{9}$$

The expected number of busy servers, which is the average number of busy servers, can be used to guide provisioning decisions regarding the size of the servers for a given load. And it can be given by

$$B_s = \sum_{k=0}^{n-1} k P_k + n \sum_{k=n}^{m} P_k, \tag{10}$$

where there is obviously a relation with $B_s \leq n$.

The expected waiting length, which is the average number in the queue, is

$$L_q = \sum_{k=n}^{m} (k-n) P_k, \tag{11}$$

The expected queue length, which is the average number for a kind of node with queue, is

$$L_s = \sum_{k=0}^{m} k P_k, \tag{12}$$

The expected response time of the queuing system is

$$R = \frac{L_s}{\lambda'}, \tag{13}$$

The expected waiting time of a token in the queue:

$$W = \frac{E_q}{\lambda'}, \tag{14}$$

The expected time spend in a sub-model is

$$\tau = R + W, \tag{15}$$

The end-to-end response time of a type i request obtained by adding the time spent by the request in the three processing steps is given by:

$$\Gamma = \tau_{1,i} + \tau_2 + \tau_{3,i}\,, \tag{16}$$

In Equation (16), $\tau_{1,i}$ is the contribution of $M_{2,I}$ to the response time, τ_2 is the contribution of M_2 (demultiplexing and queuing in the common completion queue) to the response time. Since the demultiplexing is conducted on a FCFS and one at a time, the contribution of M_2 to the response time is the same for all the types of requests. $T_{3,i}$ is the contribution of sub-mode $M_{3,i}$ (queuing in the separate completion and completion event handling) to the response time.

3.3. Optimization Goal and Constrains

Except for the goals described in section 2.2, the approximate performance estimates, which can provide sufficient information for design-time analysis, are also obtained. For each request type, it can includes utilization rate ρ, loss rate ε, expected response time R, expected queue lengths L_q, end-to-end response time Γ and so on.

The constraints include loss rate limit (limL), utilization rate limit (limU) and thread per core in order to adjust the number of threads or stop running while optimizing towards the goals.

4. Results and Discussion

4.1. Model Validation

This experiment serves to validate algorithm by comparing the performance estimates obtained from simulation with from actual experiment. Benchmarks performed for this paper were run on dell R620 with 8 CPU cores, 2.0 GHz and 32 GB of main memory, running Window Server 2003. All computers were connected through a 100 *Mbps* Ethernet switch.

Nominal parameter values were summarized in table 1. Generators repeatedly sent data to the server the sum of which was equal to the given external rate $\lambda_1 = \lambda_2$=500 *jobs/s*. The service rates μ_i, s, γ_1 and γ_2 were derived out of some experiments.

After six iterations, the optimal thread configuration was got with n_1=n_2=6, k=4, k_1=4 and k_2=5. The performance estimates obtained from simulation and actual experiment are shown in Table 2. The confidence intervals for the estimates obtained using simulation are within 5 % of the mean and are not shown here. The results indicate that the performance estimates on loss probability, utilization rate and throughput obtained using simulation match well with estimates obtained from the actual experiment. The performance estimates on queue length and response time, especial for M_2, from simulation are larger than from the actual experiment. The major reason is that the actual service rates are not equal to the ones used during simulation.

Table 1. Nominal parameter values.

Parameters	Value
Arrival rate (λ_i)	500
Pool size (n_i)	1
Service rate (μ_i)	787
Capacity, common queue (m)	10000
Demultiplexing rate (s)	400
Pool size (k)	1
Capacity, separate queue (m_i)	10000
Completion event handling rate (γ_1 , γ_2)	190 ,148
Pool size (k_1 , k_2)	1,1

Table 2. Comparison of performance measurements.

Perf. Measure	Simulation			Experiment		
	M_1	M_2	M_3	M_1	M_2	M_3
Loss probability (L_i)	1.31e-4,1.27e-4	0	0	0.024,0.023	0	0
Utilization		0.76	0.76, 0.75		0.74	0.74,0.73
Waiting length	0	1.73	1.63,1.40	0	5.49	1.7,3.4
Response time		0.0017	0.0033, 0.0028		0.0046	0.0031, 0.0056
Throughput	499.93,499.94	999.87	499.93, 499.94	488.00, 488.67	976.67	488.00, 488.67

4.2. Optimization Towards Throughput

In this section we shows the outcome of optimization towards throughput with lim$U$=0.8, lim$L$=0.0001, λ=100 *jobs/s, m=10000*, exInc=20 *jobs/s*, μ1=μ2=787 *jobs/s*, $s = 400\,jobs/s$, γ_1=190 *jobs/s* and γ_2=148 *jobs/s*. The initial configuration has 5 nodes, one thread for each node. λ is increased by *exInc* as long as the constraints is satisfied and updated λ should be adjusted to each send in integer millisecond intervals.

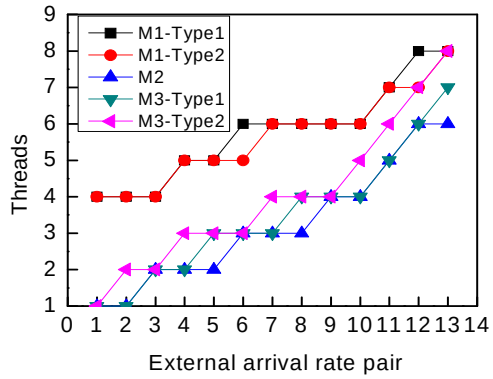
The external arrival rate pairs for type 1 and 2 are listed in Table 3 and meet the constraints during the optimization process. In Fig. 3, the left plot indicates

that the number of threads in M_1 is larger than the other in M_2 or M_3 for request type 1 or type 2 since there is no queue for each request in M_1. And it shows the necessity that Proactor pattern is only responsible for receiving data without processing data. The right plot shows that as the pool size increases, the throughput of each request type increases in M_1, M_2 and M_3. Since the parameter settings ensure that the probabilities of request loss separately in M_1, $M_2$and M_3 are negligible, the throughput of the server for each request type is identical to the request arrival rate. And because service rates for request type 1 is different from the other for request type 2 in M_3, it causes that the maximal throughput of type 1 is 1000 *jobs* / *s* larger than 889 *jobs/s* of type 2 using Equation (8).

Fig. 4 indicates that although the effective input rates increase stepwise and the service rates for each type remain constant during execution, the average queue lengths in M_1, M_2, and M_3 do not significantly increase and are between 0 and 3. This suggests that a buffer space of 10000, which was used in this experiment, should be unnecessary.

Table 3. External arrival rate pair for the valid node configuration.

	1	2	3	4	5	6	7	8	9	10	11	12	13
Type1	100	143	182	294	333	375	429	462	500	600	700	889	1000
Type2	100	121	143	238	294	333	375	429	462	500	636	778	889



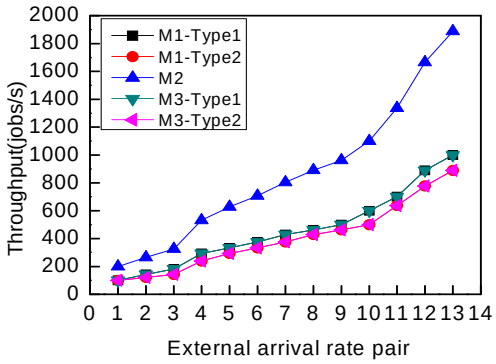


Fig. 3. Valid threads configuration and corresponding throughput in M_1, M_2 and M_3.

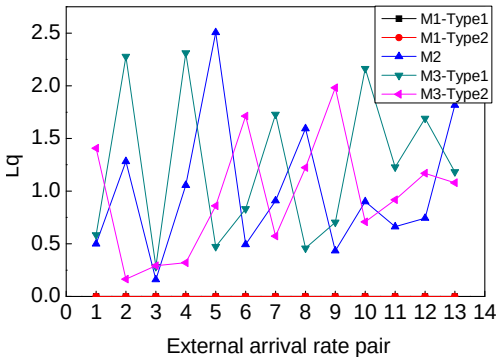
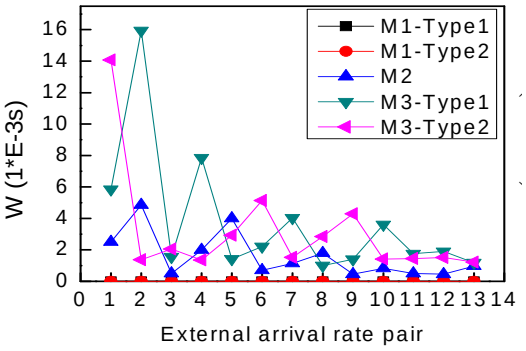


Fig. 4. Waiting length for each type in M_1, M_2 and M_3.

In Fig. 5, the left plot represents the separate time for each type in M_1, M_2 and M_3 while the right plot shows total spent time using Equation (16). Due to no queue in M_1, the expected spent time $\tau_{3,i}$ is equal to the expected response time $R_{1,i}$ and is about 0. The fact that τ_2 is less than $\tau_{3,i}$ in 9 out of 12 optimizations indicate that improving the completion handler rate yields better performance benefits for a given demultiplexing rate. And as intuitively expected, since the threads are added, the total spent time is almost the same while increasing the arrival rate.



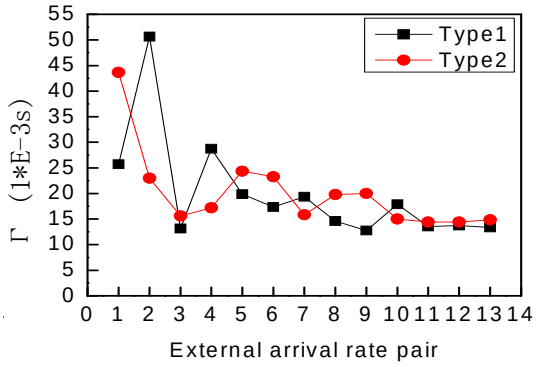


Fig. 5. Separate and total spent time for each type in M_1, M_2 and M_3.

4.3. Optimization Towards Consolidation

This section describes the outcome of optimization towards consolidation with $\lim U = 0.8$, $\lim L = 0.0001$, $\lambda = 500\,jobs/s$, $m = 10000$, $\mu_1 = \mu_2 = 787\,jobs/s$, $s = 400\,jobs/s$, $\gamma_1 = 190\,jobs/s$ and $\gamma_2 = 148\,jobs/s$. The initial configuration has 5 nodes, one thread for each node. Each λ should be adjusted to each send in the integer millisecond intervals.

In Fig. 6, Plots a, c and d illustrate that due not to meet the constraints such as utilization rate larger than limU or loss rate larger than lim$L$, threads configuration would have to adjust itself and at last form the right configuration of n_1=n_2=6, k=4, k_1=4 and k_2=5. As we know, for a given arrival rate the loss probabilities decrease as the threads increase. Further, the throughput is identical to the arrival rate because the loss probability becomes negligible.

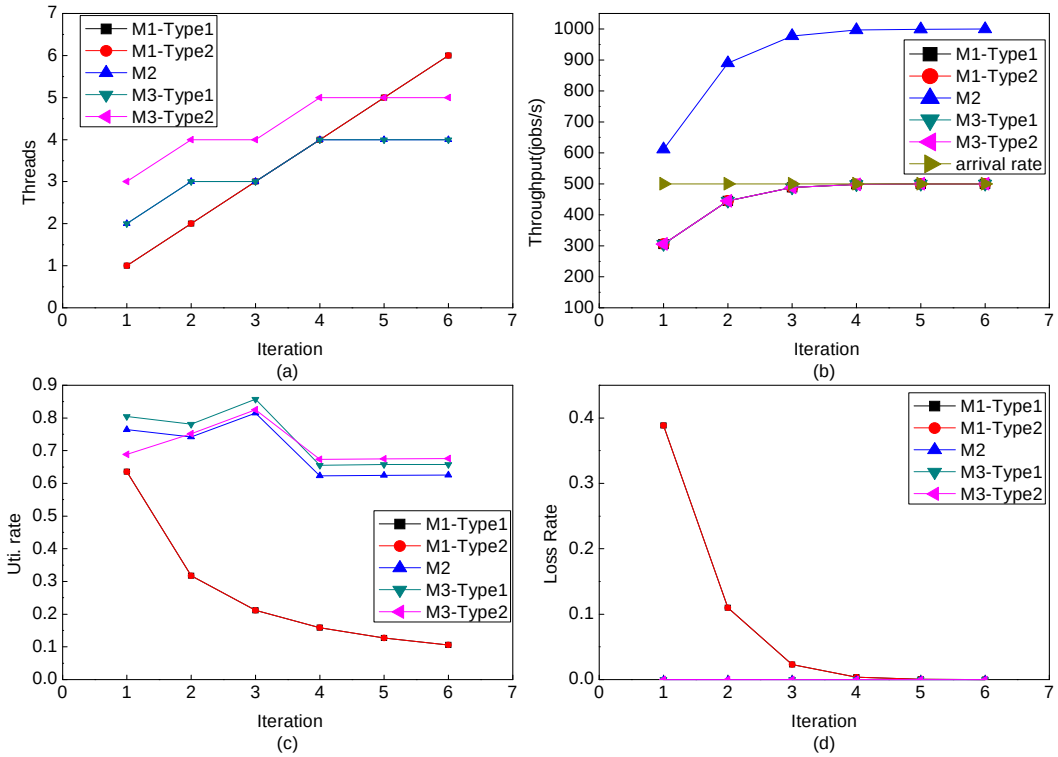


Fig. 6. Optimization towards consolidation. (a) Threads configuration; (b) Throughput for each type; (c) Utilization rate cure; (d) Loss rate cure.

Similar to above experiment, Fig. 7 shows the highest average queue length in this case is approximately 5, indicating that provisioning a buffer of 10000 should be unnecessary.

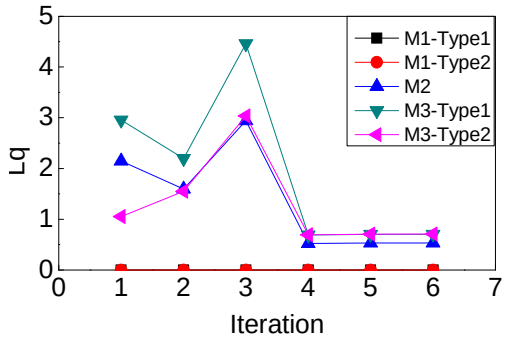


Fig. 7. Waiting length for each type in M_1, M_2 and M_3.

Similar to get Fig. 5 in above experiment, the expected spent time Γ in initial three optimizations is larger than the last three due to the increased threads from Fig. 8. Starting from the fourth iteration, the value Γ for request type 1 and 2 do not change itself and the value of request type 1 is less than the one of request type 2.

5. Conclusion and Future Work

In this paper we presented a model-based approach for the design-time performance analysis and runtime configuration adjustment of a Proacor-based server. By modeling such a server as queuing network consisting of nodes with certain functions, an optimal algorithm was built based on queuing theory. The server performance metrics such as average response time, loss probability, queue length

and throughput were obtained. According to these parameters, the model was experimentally validated. And then the optimization process towards hardware throughput or consolidation was executed in order to find the best configuration of data flow multi-thread software. As a result, the optimal number of threads *per* node and approximate performance metrics can be determined to efficiently utilize available hardware resources.

Our future research consists of developing an analytical/numerical approach for the performance analysis of the Proactor pattern.

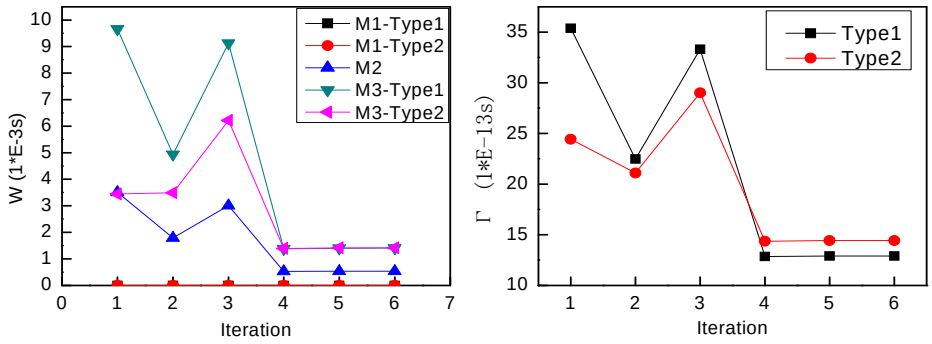


Fig. 8. Separate and total spent time for each type in M_1, M_2 and M_3.

Acknowledgements

This project is supported by the national science and technology support program (2011BAG05B02) and Fundamental Research Funds for the Central Universities (No. DUT14LAB07).

Reference

[1]. Vuppala S. K., Ghosh A., Patil K. A., et al., A scalable WSN based data center monitoring solution with probabilistic event prediction, in *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA)*, 2012, pp. 446-453.

[2]. Schmidt D. C., Stal M, Rohnert H., et al., Pattern-oriented Software Architecture: Patterns for Concurrent and Networked Objects, Vol. 2, *Wiley*, 2000.

[3]. Balsamo S., De Nitto Personè V., Inverardi P., A review on queueing network models with finite capacity queues for software architectures performance prediction, *Performance Evaluation*, Vol. 51, Issue 2, 2003, pp. 269-288.

[4]. Nossenson R., Attiya H., The N-Burst/G/1 model with heavy-tailed service-times distribution, Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, in *Proceedings of the 12th Annual International IEEE Symposium on Computer Society's,* 2004, pp. 131-138.

[5]. Cao J., Andersson M., Nyberg C., et al., Web server performance modeling using an M/G/1/K*PS queue, in *Proceedings of the 10th International Conference on Telecommunications (ICT'03),* Vol. 2, 2003, pp. 1501-1506.

[6]. Praphamontripong U., Gokhale S., Gokhale A., et al., An analytical approach to performance analysis of an asynchronous web server, *Simulation,* Vol. 83, Issue 8, 2007, pp. 571-586.

[7]. Delias P., Doulamis A., Doulamis N., et al., Optimizing resource conflicts in workflow management systems, *Knowledge and Data Engineering, IEEE Transaction*, Vol. 23, Issue 3, 2011, pp. 417-432.

[8]. Osogami T., Kato S., Optimizing system configurations quickly by guessing at the performance, *Performance Evaluation Review, ACM Sigmetrics*, Vol. 35, Issue 1, 2007, pp. 145-156.

[9]. Xi B, Liu Z., Raghavachari M., et al., A smart hill-climbing algorithm for application server configuration, in *Proceedings of the 13th International Conference on World Wide Web (ACM'13),* 2004, pp. 287-296.

[10]. Hlavacs H., Nussbaumer M., Optimization for multi-thread data-flow software, *Computer Performance Engineering. Springer Berlin Heidelberg*, 2011, pp. 102-116.