A Novel Predictive and Self–Adaptive Dynamic Thread Pool Management

Kang-Lyul Lee, Hong Nhat Pham, Hee-seong Kim, and Hee Yong Youn
School of Information and Communication Engineering
Sungkyunkwan University, 440-746, Suwon, Korea
{ intensity, phnhat, maxker }@skku.edu, and youn@ece.skku.ac.kr

Ohyoung Song
School of Electrical and Electronics Engineering
Chung-Ang University, 156-756, Seoul, Korea
song@cau.ac.kr

Abstract

Multithreading is an efficient technique commonly used to maximize the performance of CPU. One of the most important challenges in multithreading is thread pool management. It needs to retain a proper number of threads in the pool, which minimizes the response time and maximizes the resource utilization. To achieve this goal, this paper proposes a novel trendy exponential moving average (TEMA) scheme for predicting the number of threads. Also, a prediction-based thread pool management scheme is proposed which adjusts the idle timeout period and thread pool size to effectively adapt the system to the changing environment. The experiment results show the effectiveness of the proposed approach in terms of response time and CPU usage, compared with the existing prediction-based scheme and Sun watermark.

Keywords- dynamic thread management; multithreading; prediction; queuing model; resource management; thread pool

1. Introduction

The client-server model has long been adopted to efficiently provide the services in the distributed environment. To maximize the performance of the system of this model, though, several issues affecting the QoS need to be properly handled. Especially, if a large number of simultaneous requests arrive, the QoS can degrade significantly such as long response time, unavailable service, etc. This problem gets more serious in the distributed, ubiquitous system which deals with large, heterogeneous, dynamic data and resources to provide various kinds of services, from simple to complicated intelligent ones.

One of the primary causes of QoS degradation is insufficient resources, *e.g.*, lacking CPU time and memory required to handle several requests at the same time. The requests could be queued, but in the worst case, they might be dropped. While the multithreading technique has been proven to effectively solve this problem by enhancing the resource utilization, thread pool management needs a special attention due to many design issues. First, thread creation time is one of the factors lengthening the response time [1,4]. Second, redundant threads waste memory space and CPU time. Third, lack of threads causes under-utilization of CPU. Finally, frequent creation and destruction of threads degrades the system performance since a lot of resources are wasted instead of processing the requests. If the thread pool is managed efficiently, the system performance can thus be significantly enhanced.

Several models for thread pool management have been developed. The earliest and simplest model is one thread per request, in which a new thread is created when a request arrives and destroyed after the process of the request is finished. The next generation of this model is worker thread pool and its enhancement, water mark model. With the worker thread pool model a certain number of threads are created when the server is started. If there is no idle thread in the pool when a request arrives, it is queued. Meanwhile, the water mark model adjusts the number of threads according to the current number of incoming requests such that it lies within the predefined low and high threshold [3,4]. Recently, the prediction-based model [3,4] has been introduced which considers the request rate to create threads ahead of time. Although these models evidently improve the system performance, they still have some weaknesses since only a factor such as the request rate or number of worker threads is considered in the management.

This paper proposes a new prediction scheme for thread pool management of client-server system. It extends the exponential moving average (EMA) model [3] by considering the trend of time series [7,11]. The proposed scheme called 'trendy exponential moving average (TEMA)' is used in adjusting the thread pool size. The paper also introduces a mathematical model representing the relation of the factors considered in adjusting the thread pool size on the fly. It then proposes a system configuration mechanism based on two configuration parameters adjusted dynamically; idle timeout period and threshold of thread pool size. Computer simulation shows that the proposed

978-0-7695-4428-1/11 $26.00 © 2011 IEEE
DOI 10.1109/ISPA.2011.61

approach significantly outperforms the existing solutions even with greatly varying request rates and workloads.

The rest of the paper is organized as follows. Section II presents some backgrounds including the existing solutions for thread pools management. In Section III the proposed approach is presented regarding TEMA and prediction-based thread pool management. Section IV shows simulation results verifying the effectiveness of the proposed approach compared to the existing approaches. Finally, in Section V, we conclude the paper with a summary of the contributions and future work.

2. Related Works

This section discusses the management of thread pool along with some existing schemes and challenges. The schemes quoted here, including the proposed scheme, are based on non-preemptive thread model.

The thread per request model [3] spawns a new thread for each request and then destroys it after completing the service of the request. It does not have lacking or redundant threads but servicing the request is usually delayed due to thread creation time and substantial resources are wasted for frequent thread creation/destruction. Although the worker thread pool model [4] can fix these problems, many redundant threads exist when the request rate is low while threads are unavailable when the rate is high. This is because a fixed number of threads are created when the server is initiated. The watermark model [3,4] can reduce the redundancy but still has the issue of lacking threads because it does not create threads any more if the thread pool size reaches the limit. This model then has the same problem as the thread per request model for launching a new thread for each request when the system has lacking threads. If high watermark is used, however, the cost of thread maintenance quickly degrades the system performance.

The solution in [1] can alleviate the limitations of the prior models by calculating an optimal thread pool size based on thread creation time, thread context switching time, and distribution of the number of threads. However, the estimated optimal thread pool size might be inaccurate since accurately estimating the thread creation and thread context switching time is difficult in practice. Similar to [1], [2] proposes a scheme determining an optimal configuration, including the size of thread pool. It analyzes various information of the running system to decide the best configuration. Since this scheme is too complex to quickly make a conclusion, it is not suitable for the system having frequent state change.

The current trend in managing the thread pool is predicting the number of threads to be created in advance. The model in [4] calculates the rate of change in thread numbers, and then uses it as a coefficient of a linear equation identifying the expected number of threads. Furthermore, the interval between two consecutive predictions is dynamically adjusted based on Gaussian distribution to enhance the effectiveness of the prediction and save the resources. The weakness of this model is deciding the coefficient of the linear equation purely based on the last and current number of threads. Consequently, imprecise prediction may occur. The other prediction-based model was introduced in [3]. In this paper EMA was extended to forecast the number of threads, which improves the accuracy of EMA. Its drawback is, however, waste of substantial resources because of frequent thread creation/destruction and many redundant threads.

The models mentioned above do not cover the factors affecting the system performance in an integrated way. In the proposed approach these factors, e.g., the request rate, number of worker and idle threads, number of waiting requests in the queue, thread pool size and idle timeout period, are considered together in managing the thread pool. With the proposed approach lacking and redundant threads can be avoided by creating the threads in advance and destroying them when the idle timeout period expires and some other conditions are satisfied. Moreover, it does not require large time and resources.

3. The Proposed Scheme

In this section we solve the problem mentioned above by proposing TEMA, which is followed by a novel prediction-based thread pool management scheme. An M/M/c queuing model [9] is also introduced**.**

3.1 Trendy Exponential Moving Average

In each prediction, a small number of lacking threads allows higher system performance than that with redundant threads. Even with only one redundant thread per each time of thread creation, the number of redundant threads will rapidly grow since a large number of threads are created during an operation. When it happens, the thread creation/destruction overhead together with the maintenance cost for the redundant threads will significantly degrade the system performance. Meanwhile, we found that lacking small number of threads does not considerably degrade the response time through an experiment, especially when the coming request rate is high.

In [3] EEMA scheme was proposed by extending the EMA scheme to predict the number of threads.

However, it is not accurate and results in many redundant threads. To solve this problem, the EMA scheme is extended, called TEMA, with a_t which is the trend of time series at t defined as follows.

$$\boldsymbol{a_t} = \frac{\sum_{i=t-N+1}^{t-1}(x_{i+1}-x_i)}{N-1} = \frac{x_t - x_{t-N+1}}{N-1} \quad (1)$$

where N is the number of observations orderly sequenced and x_t is the observed number at t. Eq. (1) indicates that a_t is the average increase of the observations in the time series. Let ema_{t+1} be the predicted number at time (t+1) based on the EMA scheme. The predicted number at time (t+1) using the proposed TEMA scheme, e_{t+1}, becomes

$$e_{t+1} = ema_{t+1} + a_t \quad (2)$$

An evaluation of TEMA with others is made in terms of

- Chi-Square (CS) [6].
- Thread redundancy which is the sum of the differences between the predicted and observed value (SD), $\sum_{i=1}^{100}(\Delta m_i)$, in which

$$\Delta m_i = \begin{cases} e_i - x_i & \text{, when } e_i - x_i > 0 \\ 0 & \text{, otherwise} \end{cases}$$

  and 100 is the number of predictions.
- Lacking threads which is the sum of absolute values of the differrences between the predicted and observed value (SAD), $\sum_{i=1}^{100}(\Delta m_i)$, in which

$$\Delta m_i = \begin{cases} |e_i - x_i| & \text{, when } e_i - x_i < 0 \\ 0 & \text{, otherwise} \end{cases}$$

Figure 1 shows that the proposed TEMA is much more accurate than EEMA for CS and SD, while it is slightly worse for the SAD criterion. However, as already identified, lacking thread is much less harmful than redundant thread. As a result, the proposed TEMA will be more effective than EEMA for thread pool management.

3.2 Prediction-Based Thread Pool Management

Some theorems are used for calculating the number of new threads, described as follows.

Theorem 1. Let a_t^w be the trend of the number of waiting requests at t. The system needs a_t^w new threads ($a_t^w \geq 0$) to process the waiting requests at t.

Proof. Let n_t^w be the number of waiting requests at t. In M/M/c queuing model, the mean queue length $E(L)$ is defined as follows

$$E(L) = \lambda E(W) \quad (3)$$

in which,

$$E(W) = \frac{\pi_w}{\left(1-\frac{\lambda}{c\mu}\right)(c\mu)} \quad (4)$$

and

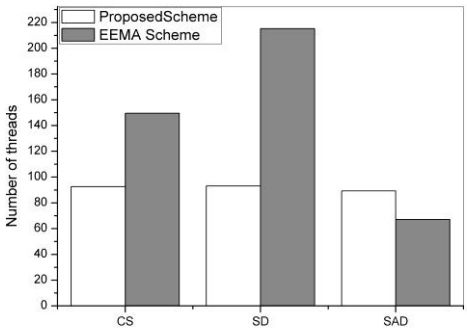


Figure 1. The comparison of the two schemes.

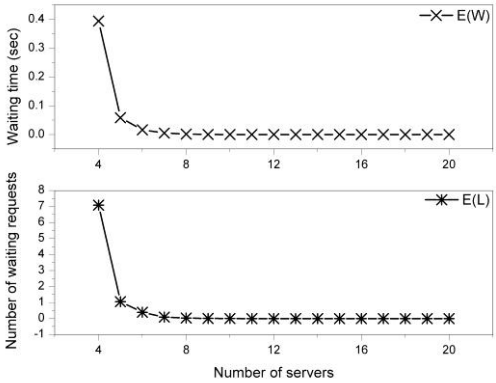


Figure 2. The mean waiting time and queue length.

$$\pi_w = \frac{\frac{(c\lambda)^c}{c\mu}}{c!}\left((1-\frac{\lambda}{c\mu})\sum_{n=0}^{c-1}\frac{(\frac{c\lambda}{c\mu})^n}{n!} + \frac{(\frac{c\lambda}{c\mu})^c}{c!}\right)^{-1} \quad (5)$$

where $E(W)$, λ, and μ are the mean waiting time, arrival rate of the requests, and service rate of the servers, respectively. $E(L)$ with λ = 18 and μ = 5 requests/s depicted in Figure 2 shows that the number of waiting requests is usually small when there exist several threads ($\geq$ 5) in the system. Therefore, when $a_t^w < 0$, n_t^w will be reduced to zero soon. Next, when $a_t^w = 0$ the system is stable. If $n_t^w > 0$, it gradually decreases without any new threads. Meanwhile, creating new threads quickly decreases n_t^w and produces redundant threads. Note that new thread is unnecessary when $a_t^w = 0$.When $a_t^w > 0$, since the trend a_t defined by Eq. (1) is the average increase of threads in one interval, new threads as much as a_t^w are needed to reduce a_{t+1}^w to zero in order to gradually decrease the waiting requests. In general, a_t^w new threads ($a_t^w \geq 0$) are required to process the waiting requests and thus maximize the system performance. □

The TEMA-based method can avoid inaccurate prediction even though the input data are not stable but

change abruptly. In this paper e_{t+1} of EMA is used as the predicted value. However, it is also referred as the smoothed value of the observation x_t and calculated as follows [5].

$$e_{t+1} = \frac{2}{N+1} x_t + \left(1 - \frac{2}{N+1}\right) e_t \quad (6)$$

Eq. (6) shows that the observed value contributes little to the smoothed value since the ratio 2/(N+1) is smaller several times than (1− (2/(N+1)), when N is greater than ten. As a result, the smoothed data has no peek and nicely represents the trend of the time series. Moreover, since the TEMA value is the sum of EMA value and the trend a_t (which does not have any peak), TEMA also has the ability of smoothing the data of the time series and reducing the inaccuracy of the estimation.

Based on the properties of the proposed scheme identified above, a mathematical model is constructed to calculate the number of required threads. First, the required threads from t to (t+1) consist of:

- the ones for processing the waiting requests at time t. Due to Theorem 1, the threads as many as a_t^{w} are necessary,
- the ones for processing incoming requests from t to (t+1), $p_{t,t+1}^{in}$. It is calculated by Eq. (2).

Therefore, the number of required threads from t to (t+1) is

$$n_{t,t+1}^{r} = a_t^{\mathrm{w}} + p_{t,t+1}^{in} \quad (7)$$

Second, the threads estimated above are provided from

- idle threads, n_t^i (=n_t- n_t^o), in which n_t and n_t^o are the number of threads and worker threads at t. There exists some peaks in the time series of n_t^o but not in that of n_t. n_t^o is replaced by the predicted number of worker threads at (t+1), p_{t+1}^{o}, to increase the prediction accuracy,
- worker threads, $p_{t,t+1}^{s}$, which is the predicted number of serviced requests from t to (t+1),
- and new threads, $n_{t,t+1}^{n}$, if necessary.

The number of available threads to process the requests from t to (t+1) is thus:

$$n_{t,t+1}^{a} = n_t^i + p_{t,t+1}^{s} + n_{t,t+1}^{n} \quad (8)$$

Finally, to achieve the maximal performance,

$$n_{t,t+1}^{r} = n_{t,t+1}^{a} \quad (9)$$

Therefore, we have

$$n_{t,t+1}^{n} = p_{t+1}^{w} + p_{t,t+1}^{in} + p_{t+1}^{o} - n_t - p_{t,t+1}^{s} \quad (10)$$

The watcher thread will create more threads when $\boldsymbol{n_{t,t+1}^{n}} > 0$. Otherwise, the possibility of lacking threads is low and the system does not need new threads.

3.3 Dynamic Adjustment of Configuration Parameters

In this subsection a dynamic approach is proposed to adjust some configuration parameters on the fly to promptly reflect the operation condition.

A. Threshold of the Number of Threads

Though the number of threads in the system is dynamically adjusted based on the system status, it might be still inadequate in some cases. To resolve this issue, this paper suggests to adjust the number of threads in the system with a limit, th_u, which is identified using the M/M/c queuing model described in Figure 2. Due to Eq. (4), we see that $lim_{c\to\infty} E(W) = 0$. With λ = 18 and μ = 5 requests/s, Figure 2 shows that $E(W)$ is reduced little (< 0.1ms) when c > 8. In reality, if c > 8, many threads become redundant and degrades the system performance. Based on this observation, th_u is set to 8.

B. Idle Timeout Period

Currently, most existing thread pool models assume constant idle timeout period, t_i. In many cases, constant t_i cannot maximize the performance since it does not allow to keep a proper number of threads in the system. This paper considers three cases to solve this problem. Let t_p be the prediction interval.

The first case occurs when a_t^o grows and there exist several idle threads in the pool. In this case the risk of lacking threads is not high. Therefore, t_i needs to be slightly increased by t_p to compensate the increase of a_t^o while preventing the thread redundancy. Let a_t^o be the trend of the number of worker threads at t, determined by Eq. (1). Mathematically, this case occurs when $a_t^o > 1$ and $\frac{(n_t - n_t^o)}{n_t^o} \geq 0.5$.

The second case has ascending $\boldsymbol{a_t^o}$ and very few idle threads in the pool. Here t_i ought to be increased larger than the first case due to higher risk of lacking threads. The new t_i will allow idle threads not to be destroyed until the system state is checked again in the next interval to reduce the possibility of lacking threads. Mathematically, this case happens when $\boldsymbol{a_t^o > 1}$ and $\frac{(n_t - n_t^o)}{n_t^o} < 0.5$.

In the last case, t_i is decreased in proportion to the ratio of n_t^o and n_t if one of the following three conditions is satisfied:

- a_t^o is slightly increased and there are many idle threads.

- The number of worker threads tends to be reduced since the workload of the system is descending.
- The system has no request.

Mathematically, this case occurs when ($0 \le a_t^o \le 1$ and $\frac{(n_t - n_t^o)}{n_t^o} \ge 0.5$) or $a_t^o \le 0$ or $n_t^o = 0$.

In summary, t_i is dynamically adjusted based on the current status of the system as follows.

$$t_i = \begin{cases} t_i + t_p, & \text{for the first case} \\ t_i + 2t_p, & \text{for the second case} \\ \frac{t_i\, n_t^o}{n_t}, & \text{for the third case} \end{cases} \quad (11)$$

C. Thread Destruction

There exist two cases for which the idle threads ought not to be destroyed.

- If new threads have just been created, thread destruction causes the thread creation to be useless.
- The probability of lacking threads is high since there is no idle thread in the pool. To prevent this condition, the idle threads are permitted to be destroyed at t when one of the two conditions below is satisfied:
  - ✓ The system has enough threads in the case of maximum n_t^o; $n_t \ge n_{max}^o$, where n_{max}^o is the maximum value of the time series of n_t^o.
  - ✓ At least one idle thread exists in the pool after a thread is destroyed; $n_t - n_t^o \ge 2$.

4. Performance Evaluation

This section presents the experiment setup used to evaluate the performance of the proposed approach and the experiment results.

4.1 Experiment Setup

To realistically validate the proposed scheme, we setup a real system of an authentication server managing 5000 user accounts stored in the database server of MySQL Community 5.1. The authentication requests are generated by the clients and they are sent to the authentication server using the traces of traffic actually addressed to Wikipedia [10]. Sampling from the original trace of Wikipedia is used to create the traces of various request rates, which ensures realistic performance evaluation. Here the password of user account is encoded with MD5 algorithm. The client and servers have AMD Dual Core 2.80 GHz processor, 2GB main memory, Window 7 OS, and Sun JDK 6. Due to the results in [3], t_p is set to 100 ms to achieve the highest performance. The configuration parameters are described in Table 1.

Besides, the feasibility and effectiveness of the proposed scheme are evaluated, compared with the Sun watermark scheme [8] commonly used in various applications and the EEMA scheme [3]. The performances are evaluated in term of response time and CPU usage. Memory usage is not investigated since the memory space taken by thread pool is small and the difference between the schemes is ignorable (about 30KB), as identified through an experiment. In the experiment, the average values are obtained from the results of 10000 runs. Here th_i^l and th_i^u are the lower and upper threshold of t_i.

Table 1. The parameters used in the experiment.

Parameter	t_i*	th_i^l**	th_i^u**	t_p***
Value	300 ms	100 ms	30000 ms	100ms

(* : for the EEMA and watermark scheme.
** : for the proposed scheme.
*** : for the proposed and EEMA scheme.)

4.2 Result of Experiment

A. Response Time

Figure 3 compares the response times of the three schemes. Observe that the proposed scheme significantly outperforms the existing schemes, especially when the request rate is high from 220 requests/sec. The proposed scheme reduces the response time by up to 60% and 94% compared with the EEMA and watermark scheme in the case of 380 requests/sec. With 410 requests/sec, the EEMA scheme cannot handle the requests since the number of threads becomes too large causing the system crash. The response time with the watermark scheme is 76 times larger than that of the proposed scheme, and thus it cannot be shown in the figure.

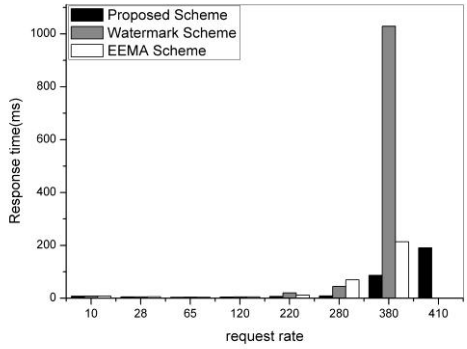
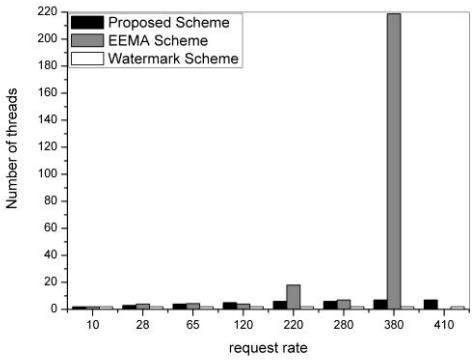


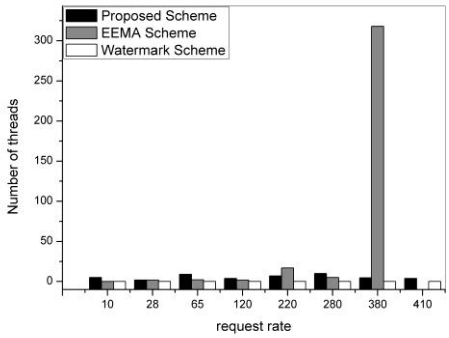
Figure 3. The comparison of response times.

B. CPU Overhead

Figure 4(a) shows that when the request rate is small, the maximal number of threads in the system, n_{max}, of the proposed scheme is similar to that with the EEMA scheme. However, when the request rate is large (greater than 120 requests/s), n_{max} of the proposed scheme is 36 times smaller than that of the EEMA scheme. It means that the proposed scheme reduced the CPU usage required for context switching $36 \times p$ times, where p is decided according to the workload of the system and the quantum time of a thread. Observe from Figure 4(b) that the number of threads created by the proposed scheme is stable with small deviation in both the cases of small and large request rate while that of EEMA greatly varies when the request rate is large. The number of threads created by the EEMA scheme is 67 times higher than that of the proposed scheme. It implies that the proposed scheme reduces the CPU overhead taken for thread creation/destruction 67 times compared with the EEMA scheme.



a) The maximum number of threads.



b) The number of threads created.

Figure 4. The comparison of the number of threads created with different schemes.

5. Conclusions

This paper has presented a novel scheme which dynamically adapts the thread pool to the operation condition to maximize the system performance. To achieve the goal, the TEMA scheme was proposed to accurately predict the number of threads. In addition, a new prediction-based thread pool management was developed to dynamically adjust the number of threads in the pool, considering the idle timeout period and the conditions of thread destruction. It allows effective thread creation/destruction according to the current status. An experiment shows that the proposed scheme significantly outperforms the existing schemes for various request rates in terms of response time and CPU overhead.

For future work, we plan to develop a new prediction scheme enhancing the prediction accuracy. Furthermore, a scheme considering the priority of the watcher and worker thread will be investigated to effectively arbitrate the operations of thread creation/destruction along with the request processing.

Acknowledgment

This research was supported by a grant (CR070019) from Seoul R&BD Program funded by the Seoul Development Institute of Korean government. Corresponding author: Hee Yong Youn.

References

[1] Y. Ling, T. Mukken, and X. Lin, "Analysis of Optimal Thread Pool Size," *ACMSIGOPS Operating Systems Review*, vol. 34, no. 2, pp. 42–55, 2000.

[2] 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 (IM '09)*, pp. 1-8, 2009.

[3] D. Kang, S. Han, S. Yoo, and S. Park, "Prediction-based dynamic thread pool scheme for efficient resource usage," *The 8th IEEE International Conference on Computer and Information Technology*, pp. 159-164, 2008.

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

[5] http://stockcharts.com

[6] M.K. Molloy, *Fundamentals of Performance Modeling*, 1989.

[7] C. Chartfield, *The Analysis Of The Time Series, An Introduction*, ed. 5th, 1995.

[8] *ThreadPoolExecutor* of j2se 1.6.0, website at: http://java.sun.com/

[9] Ivo Adan and Jaques Resing , *Queueing Theory*, 2001.

[10] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen, "Wikipedia Workload Analysis for Decentralized Hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830-1845, July, 2009.

[11] G. E. P. Box and G. M. Jenkins, *Time Series Analysis: Forecasting and Control*, Holden Day, San Francisco, 1976.