한국항행학회 논문지 제17권 제2호 2013년 4월 (JKONI 17(2): 189-195, Apr. 2013)

스레드 생성 및 삭제 오버헤드를 줄이기 위한 히스토리 기반의 동적 스레드풀 방법

A History-based Dynamic Thread Pool Method for Reducing Thread Creation and Removal Overheads

오삼권*, 김진섭*

Sam-Kweon Oh*, Jin-Sub Kim*

요 약

작업의 실행요청이 빈번하고 실행시간이 짧은 환경에서, 스레드 생성 및 삭제로 인한 오버헤드를 줄여 작업 처리효율을 향상시키기 위해 스레드풀 방법이 자주 사용된다. 그중에서 워터마크 스레드풀 방법은 최대로 필요한 스레드 수 이하의 스레드들을 유지함으로써 자원의 불필요한 사용을 줄인다. 그러나 사용가능한 스레드가 없을 경우에는 지정된 한도 내에서 스레드를 추가로 생성하여 작업들을 처리해야 하므로 스레드 생성으로 인한 시스템 오버헤드가 증가하고 결국 성능 저하 현상이 발생할 수 있다. 본 논문은 이런 성능 저하를 줄이기 위한 방법으로써 히스토리 기반의 동적 스레드풀 방법을 제안한다. 이 방법은 작업처리에 필요한 스레드 수를 측정하고 유지함으로써 스레드 생성 및 삭제로 인한 오버헤드를 줄인다. 실험결과에 따르면, 제안방법은 워터마크 스레드 풀 방식과 비교하여 보유 스레드 수는 평균 33% 증가하나 스레드 생성 수는 평균 62% 감소함으로써 평균 6%의 시스템 처리량 증가를 보인다.

Abstract

In an environment with frequent job requests and short job processing times, thread pool methods are frequently used to increase throughput by reducing overheads due to thread creation and removal. A watermark method normally reduces unnecessary uses of resources by keeping the number of threads less than those needed in the maximum. In the absence of available threads, however, it processes jobs by creating additional threads up to a specified limit so that the system overhead increases due to creation of threads, which results in throughput degradation. This paper presents a history-based dynamic method that alleviates throughput degradation. By estimating and maintaining the number of threads needed for jobs, it reduces overheads due to thread creation and removal. According to experiments, compared to the watermark thread pool method, it shows average 33% increase in the number of threads kept and average 62% reduction in the number of threads created, which results in 6% increase in terms of system throughput.

Key words : History-based, Dynamic Thread Pool, Overhead, Throughput
키워드 : 히스토리-기반, 동적 스레드 풀, 오버헤드, 처리량

I. Introduction

A thread is the smallest sequence of program instructions that can be executed by the OS scheduler

* 호서대학교 뉴아이티공과대학 컴퓨터공학전공(College of NewIT Engineering(Department of Computer Engineering), Hoseo University)
· 제1저자 (First Author) : 오삼권(Sam Kweon Oh, Tel : +82-41-540-5711, email : ohsk@hoseo.edu)
· 접수일자 : 2013년 2월 18일 · 심사(수정)일자 : 2013년 2월 18일 (수정일자 : 2013년 4월 16일) · 게재일자 : 2013년 4월 30일
http://dx.doi.org/10.12673/jkoni.2013.17.2.189

and multiple threads can exist in the same process[1]. Among the methods for processing jobs using threads, the simplest one is to create a thread for processing a job and remove it after it completes the job. This method is called a thread per request method(TPR) since it uses only one thread for handling a job. In case of using TPR, as the number of jobs to be processed increases, the time used for thread creation and removal increases. Consequently, the time for processing jobs is wasted and system throughput decreases. To alleviate this problem, thread pool methods are frequently used [2].

A thread is the smallest sequence of program instructions that can be executed by the OS scheduler and multiple threads can exist in the same process[1]. Among the methods for processing jobs using threads, the simplest one is to create a thread for processing a job and remove it after it completes the job. This method is called a thread per request method(TPR) since it uses only one thread for handling a job. In case of using TPR, as the number of jobs to be processed increases, the time used for thread creation and removal increases. Consequently, the time for processing jobs is wasted and system throughput decreases. To alleviate this problem, thread pool methods are frequently used [2].

Thread pool methods in general reduce the time overheads due to thread creation and removal, not by removing a thread that completes its job but by using it for a next job to be processed. According to the way the number of threads is managed, thread pool methods can be divided into two categories: static and dynamic. The methods in the former only use a fixed number of threads, regardless of the amount of jobs. On the other hand, those in the latter dynamically manage the number of threads, depending on the amount of jobs. A watermark pool method(WM) is a well-known one in the latter category. It maintains a fixed number of threads until it needs more threads due to more jobs. It creates additional threads up to a specified number when needed; they are of course removed after they complete their jobs. In summary, WM shows better performance compared to those in the static category, in an environment where the amount of jobs vary dynamically. However, it still has a problem of throughput decrease due to creation and removal of additional threads, as in the case of TPP.

To alleviate this problem, this paper suggests a history-based dynamic pool method(HisDyn). HisDyn, by measuring mean job arrival times and mean job processing times, estimates the range of needed threads. According to experiments, HisDyn yields better performance in terms of system throughput, with reduced overheads due to thread creation and removal. The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 presents HisDyn. Section 4 presents experimental results. Section 5 concludes the paper.

Ⅱ. Related Work

Thread pool methods are usually used for systems, such as web servers, with frequent job occurrences and short job processing times[2]. By maintaining a fixed number of threads for processing jobs, they reduce overhead due to thread creation and removal so that they can yield better system throughput compared to TPR.

Figure 1 shows a typical structure of thread pool methods. In general, it consists of a task queue where jobs to be processed reside, an I/O thread that inserts jobs to the task queue, and worker threads that process jobs in the task queue. A thread pool method, when started, creates a fixed number of threads in advance and let them wait for job arrivals. Once a job arrives for execution, the I/O thread inserts it to the task queue, and one of the waiting worker threads processes that job. Once completed, the worker thread goes back to the waiting state and waits for next job arrival.

Thread pool methods can be either static[3] or

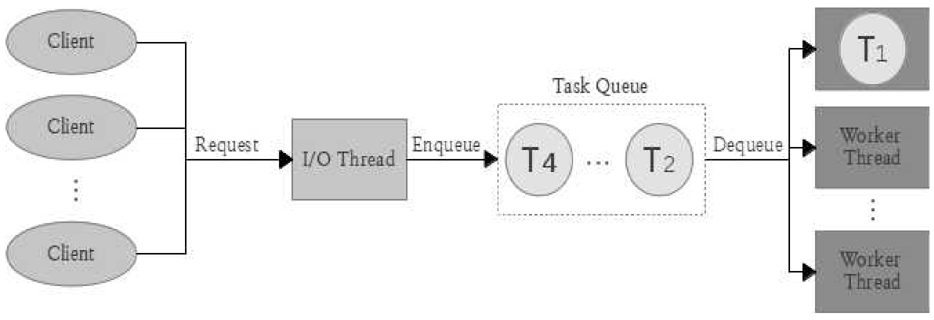

그림 1. 스레드 풀 방식의 일반적 구조
Fig. 1. A typical structure of thread pool methods

dynamic[4]. In case of a static method, a fixed number of threads are created when started and remain thereafter. Therefore, these methods have no overhead due to thread creation and removal. However, they can suffer from throughput degradation when job arrivals exceed that of threads in a thread pool. If reverse is the case, system resources are wasted due to unused threads.

Dynamic pool methods control the number of threads according to the changes in the amount of jobs. They, however, have overhead problems due to thread creation and removal, as in the case of TPR. Depending on the way how the number of threads is managed, dynamic pool methods can be in either watermark-based or prediction-based methods.

A watermark-based method(WM)[5] manages the number of threads within the lower limit and the upper limit. It creates threads up to the lower limit when started. It can create additional threads up to the upper limit when needed and the additionally- created threads are removed when they become unnecessary.

A prediction-based method[6] is one that uses prediction for thread creations. It uses an exponential moving average(EMA)[7] to identify the direction of the trend in the number of needed threads. In case of a growing trend, based on the rate of increase, it calculates the number of threads needed in the future. Other variations using different prediction methods for prediction exist in this category[8,9,10].

Prediction-based methods are those proposed for fast response times. Similar to WM, they create new threads when no threads are available for new jobs. In case of prediction failures (i.e., in case that newly-created threads become unnecessary), the number of threads may be unnecessarily increased. We do not discuss prediction-based methods in this paper since the proposed method (HisDyn) is one not for fast response times but for preventing unnecessary removal of threads (and hence for reducing the overhead due to thread creation and removal). We do not discuss prediction-based methods in more detail here since they are beyond the scope of this paper. Interested readers may refer to[11].

Ⅲ. History-based Dynamic Method

WM performs better than those of static methods since it manages the number of threads within a specified range. However, it suffers from performance degradation due to thread creation and removal overheads. Another burden is that user must specify the range of needed threads and continuous monitoring should be done for range adjustment.

To alleviate these problems, this paper presents a history-based dynamic thread pool method called HisDyn. HisDyn estimates the number of needed threads by calculating a job waiting probability using a mean job arrival time and a mean job processing time it controls the range of needed threads by finding the minimum number of needed threads where the job waiting probability is less than a user-specified job waiting probability. In other words, it reduces thread creation and removal by estimating the number of threads needed for processing jobs and using the number as the lower limit for thread creation.

Figure 2 shows the thread pool structure of HisDyn. It is similar to that of WM, but with two additional components: a data manager and a range manager

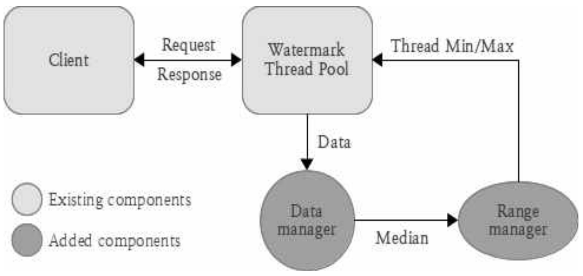

그림 2. HisDyn의 스레드 풀 구조
Fig. 2. Thread pool structure of HisDyn

The data manager accumulates job arrival intervals and processing times. It calculates their mean values for estimating the number of needed threads. The range manager determines the (min, max) range of needed threads by estimating the number of threads needed for processing jobs. The max value is initially set by the system user and the min value is obtained by using a mean job arrival time between two consecutive jobs and a mean job processing time. To reduce the overhead due to finding the (min, max) range, the range manager is executed only when it notices a symptom of changes in the amount of jobs that is, the (min, max) range is adjusted only when a thread is newly created due to thread shortage and when existing threads are more than needed. The job waiting time is set to (i * n) * t, where i is the mean time between two consecutive job arrivals, n is the number of existing threads, and t is a user-specified parameter for controlling the job waiting time.

The procedure to find the number of threads needed for jobs is as follows:

1. Accumulate job arrival intervals and job processing times
2. Obtain their mean values
3. Calculate job waiting probability
4. Estimate the min value for the (min, max) range of needed threads

Job waiting probability is one that a new job arrives when all the threads are being used for processing jobs (i.e., there are no threads available for a new job). A job waiting probability higher than the user-specified one implies more threads need to be created. A reverse-case probability implies some threads need to be removed. Based on these implications, the number of needed threads is determined to be the minimum number of threads having the job waiting probability less than the user-specified one.

The probability that a single thread processes a job at any point in time is (c/i), where i is the mean job arrival time and c is the mean job processing time. Since (c/i) is the same as the one that a thread is processing an immediate previous job, it is also the job waiting probability for one thread. Expanding this to an n-threads case, the job waiting probability becomes {c/(i*n)}**n, where * mean 'times' and ** means 'to the power of'.

Figure 3 is a pseudo code for adjusting the (min, max) thread range. This algorithm adjusts the range of threads by using a mean job arrival time, a mean job processing time, a user-specified job waiting probability, and the number of existing threads. It performs a linear search while modifying the lower limit(i.e. min value) of the thread range. Since the maximum search length is in between the minimum and the maximum numbers of threads, it is not large. In addition, since the job waiting probability decreases exponentially as the number of threads increases, the number of search is usually shown to be within a few search times (according to our experiment, it is usually within 5 times). Therefore, a simple linear search is more suitable than any other algorithms in this case. In the code, CalcWaitProb calculates job waiting probability and SetThreadRange

```
AdjustThreadRange(current_no_threads, interval_t, proc_t, user_spefied_prob)
{
  no_threads = current_no_threads;
  wait_prob = CalcWaitProb(interval_t, proc_t, no_threads);
  if (wait_prob> = user_spefied_prob) {
    while (wait_prob > user_specified_prob) {
      if (no_threads >= max_no_threads) break;
      no_threads = no_threads+1;
      wait_prob = CalcWaitProb(interval_t, proc_t, no_threads);
    }
  } else {
    while (wait_prob < user_specified_prob) {
      if (no_threads <= 2) break;
      no_threads = no_threads-1;
      wait_prob = CalcWaitProb(interval_t, proc_t, no_threads);
    }
    no_threads++;
  }
  SetThreadRange(no_threads, max_no_threads);
}
```

그림 3. 스레드 범위 조절 알고리즘
Fig. 3. Thread-range adjustment algorithm

sets the (min, max) thread range.

Ⅳ. Performance Evaluation

HisDyn aims to improve throughput by reducing the overheads due to thread creation and removal, which is realized by estimating and maintaining the number of threads suitable for the amount of jobs. For performance evaluation, we compare the number of created threads, system throughput, and the average number of threads in a thread pool with WM and those with HisDyn, where system throughput is the number of jobs per unit time (i.e. the total number of jobs divided by total execution time) and the average number of threads in a thread pool is calculated by averaging the numbers of threads measured at every 1ms.

Experiment is made in a Linux computer system equipped with Intel Core 2 (dual core) processor and a 2 GB memory board with 1000 job arrivals. To maximize the processor utilization, the minimum number of threads is set to 3 for WM[2] and the maximum number of threads to 100 for both methods. The job waiting probability is set to less than 1% and the job waiting time to double the mean job arrival time. Using the exponential distribution and the normal distribution functions in C++ TR1 library, a job request is made with mean arrival rate 50 (i.e., a job arrives at every 0.02 sec on average) and the system load is increased at intervals of 10% respectively.

As shown in Figure 4,while the number of created threads increases with WM as load increases, it is relatively constant with HisDyn. This is because the number of needed threads is dynamically estimated and maintained according to an amount of jobs with HisDyn. The number of created threads is increased with WM as load increases, while it is almost constant with HisDyn sinceHisDyn controls the number of needed threads according to the amount of jobs. Compared to WM, HisDyn creates average 62% less threads.

As you can see in Figure 5, HisDyn maintains more threads in a thread pool than WM except with very low loads. Since it creates threads only when necessary, threads are not kept unnecessarily. Compared to WM, HisDyn maintains average 33% more threads. Figure 6 shows system throughput. With increasing loads, HisDyn shows relatively-constant throughput while WM shows reduced throughput. It's because the number of thread creation islower with HisDyn than that with WM, except for low loads. Compared to WM, HisDyn shows average 6% increase in terms of system throughput. Figure 7 shows changes in the number of created threads with increasing job arrival rates. HisDyn shows small changes in the number of created threads with increasing job arrivals, while WM does a hundred of changes with increasing job arrivals.

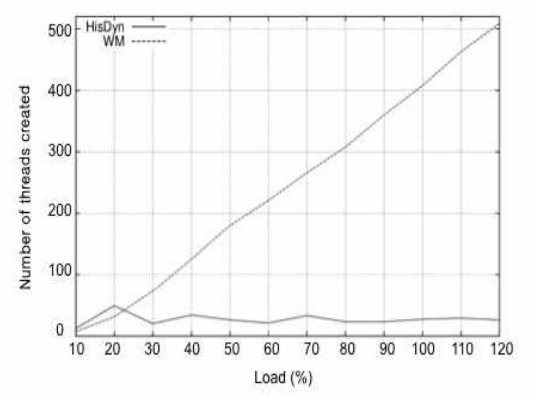

그림 4. 스레드 생성 수
Fig. 4. Number of created threads

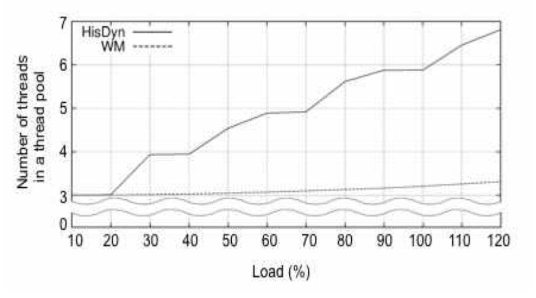

그림 5. 스레드 풀의 스레드 수
Fig. 5. Number of threads in a thread pool

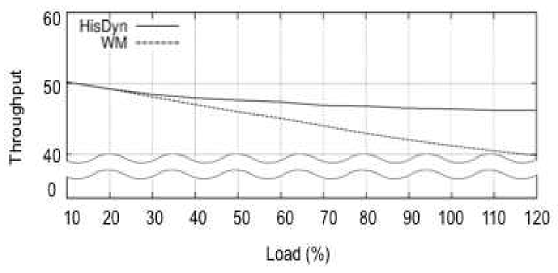

그림 6. 처리량
Fig. 6. System throughput

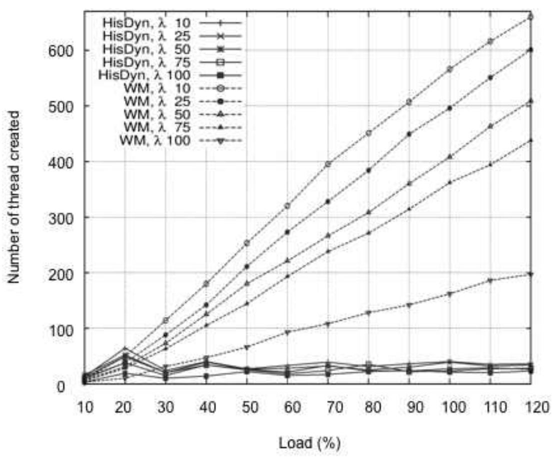

그림 7. 작업 요청 비율 증가에 따른 스레드 생성 수
Fig. 7. Number of created threads with increasing job arrival rates

Ⅴ. Conclusion

Both WM and HisDyn are dynamic methods in the sense that they dynamically change the number of threads in a thread pool, depending on the amount of jobs. One major difference is that while the lower limit in the range of needed threads is user-specified at system-start time and fixed thereafter with WM, it is estimated and adjusted dynamically at run-time with HisDyn. Creation of a thread frequently occurs repeatedly as load changes with WM, which results in poor system throughput. On the other hand, HisDyn yields better system throughput than WM, mainly by virtue of the ability to maintain dynamically the lower limit in the range of needed threads.

According to experiments with WM, HisDyn shows average 33% increase in the number of threads kept and average 62% reduction in the number of threads created, which results in 6% increase in terms of system throughput. Lastly, we plan to apply this method to a web server such as Apache or Nginx in which thread creation requests are expected to be very high so that we can evaluate it in a more practical environment.

감사의 글

이 논문은 2011년도 호서대학교의 재원으로 학술 연구비 지원을 받아 수행된 연구임(2011-0286)

Reference

[1] A. Silberschatz, P.B. Galvin, and G. Gagne. Operating System Concepts Essentials, *First Edition Binder Ready Version. John Wiley & Sons,* 2011.

[2] Brian Goetz. Java theory and practice: Thread pools and work queues. IBM Develop works, http://www.ibm.com/developerworks/library/j-jtp0730, 2002.

[3] D. Xu and B. Bode. “Performance study and dynamic optimization design for thread pool systems,” *PhD thesis, United States Department of Energy. Office of Science*, 2004.

[4] D.C. Schmidt, “Evaluating architectures for multi-threaded object request broker.” *Communications of the ACM,* vol. 41, no. 10, pp. 54–60, 1998.

[5] Dynamic TAO Documentation, http://choices.cs.uiuc.edu/2k/dynamicTAO/doc/

[6] D.H. Kang, S. Han, S.H. Yoo, and S. Park, “Prediction-based dynamic thread pool scheme for efficient resource usage,“ *In Computer and Information Technology Workshops, CIT Workshops 2008, IEEE 8th International Conference*, pp. 159–164, 2008.

[7] DongHyun Kang, SeoHee Yoo, Sungyong Park. "A Dynamic Thread Pool Scheme based on the Learning for a Web Server," *ITC-CSCC(International Technical Conference on Circuits Systems , Computers and Communications)*, pp. 268-271, 2009.

[8] AutoRegressive, http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/ar/

[9] Chang-Hwan Lee, “Calculating Attribute Weights in K-Nearest Neighbor Algorithms using Information Theory,” *KIISE JCSE, vol. 32, no. 9,* pp. 920-926, September 2005.

[10] Lee, K.L. and Pham, H.N. and Kim, H. and Youn, H.Y. and Song, O. “A Novel Predictive and Self-Adaptive Dynamic Thread Pool Management,” *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium,* pp. 93-98, 2011.

[11] Jinsub Kim, “A History-based Thread Pool Method for Reducing Overhead of Thread Creation and Deletion,” *Master's thesis for a degree in Hoseo University*, 2012.

오 삼 권 (Sam-Kweon Oh)

1980년 2월 ~ 1984년 7월: 삼성전자 통신연구소(연구원)
1986년 12월: University of South Florida(컴퓨터과학석사)
1994년 5월: Queen's University(컴퓨터 과학박사)
1995년 3월 ~현재 : 호서대학교 컴퓨터 공학전공(교수)
관심분야 : Embedded Systems, Communication Protocol

김 진 섭 (Jin-Sub Kim)

2011년 3월 ~ 현재: 호서대학교 일반대학원 컴퓨터공학과(컴퓨터 공학 석사과정)
2013년 1월 ~ 현재 : (주)SDCmicr(연구원)
관심분야 : Parallel Processing, Communication Protocol