# Dynamic Thread Management in Kernel Pipeline Web Server

LI Shan-Shan LIAO Xiang-Ke TAN Yu-Song LIU Jin-Yuan

School of Computer, National University of Defense Technology, ChangSha, China, 410073
{littlegege ,xkliao }@263.net {pine_tan, yuanchuangliu}@yahoo.com.cn

**Abstract.** With the development of high-speed backbone network, more and more traffic load is pushed to the Internet end system. The satisfactory execution of common business applications depends on the efficient performance of web server. In this paper, we propose a pipeline multi-thread kernel web server open KETA which divides the processing of a request into several independent phases. This architecture reduces parallelism granularity and achieves inner-request parallelism to enhance its processing capability. Furthermore, a thread allocation model is used to manage threads effectively in this special architecture. This model can adjust the thread allocation based on the server load and the work character of each phase so that the thread resource of web server can be utilized properly. Experimental result shows the capability of this web server and the effectiveness of the thread allocation model.

## 1 Introduction

Internet is undergoing substantial change from a communication and browsing infrastructure to a medium for conducting business and selling a myriad of emerging services. Because of the complexity of the web infrastructure, performance problems may arise in many aspects during a Web transaction. Although both network and server capacity have improved in recent years, the response time continues to be a challenge to the research on Web system. Some statistic shows that an e-commercial web site should guarantee its response in 7 seconds or it will lose more than 30% customers [1]. Recent measures suggest that web servers contribute for about 40% of the delay in a Web transaction and this percentage is likely to increase in the near future [2]. Some prediction estimated that network bandwidth would triple every year for the next 25 years. So far, this prediction seems to be approximately correct [3], while the Moore law estimates "just" a doubling of system capacity every 18 months. So we can see that the bottleneck is likely to be on the server side.

In order to solve above problem, some improvement should be made on web servers. There are mainly three ways to achieve this [4]:

- Improve the performance of a web server node at the software level, namely software scale-up.
- Upgrade web server's hardware such as CPU, memory and network interfaces to improve processing capability. This strategy, referred to as hardware scale-up,

This work is supported by HI-TECH Research and Development Program of China (863 Program) under Grant No.2002AA1Z2101 and No. 2003AA1Z2060

simply consists in expanding a system by incrementally adding more resources to an existing node.

- Deploy a distributed web system architecture consist of multiple server nodes where some system component such as a dispatcher can route incoming requests among different servers. The approach in which the system capabilities are expanded by adding more nodes, complete with processors, storage, and bandwidths, is typically referred to as scale-out.

Here we concentrate on the first method, software scale-up. Through comparison and analysis among some popular web servers' architecture and processing mechanism, we put forward a kernel pipeline web server open KETA (KErnel neTwork geAr). This web server divides the processing of a request into four phrases. Different phases of different requests can be executed concurrently like a pipeline on condition that there are no data or structure dependency. This architecture can reduce parallelism granularity effectively so that the resources of a web server can be utilized fully. Furthermore, the thread number of each phase is adjusted according to the server load dynamically in order to manage and schedule thread effectively.

The rest of this paper is organized as follows. In Section 2, we briefly describe some related work on the mainstream web server nowadays. The framework of open KETA and the thread management in open KETA are described in Section 3 and Section 4. In Section 5, some experimental results are presented.

## 2 Related Work

In view of the architecture, the mainstream web server can be classified into three categories: Single Process (SP), Symmetrical Multiple Threads (SMT) and Asymmetrical Multiple Threads (AMT).

In SP web server, a single process is responsible for the whole processing of all requests, including listening to the port, setting up connection, analyzing and processing requests, sending responses, etc. Some representative examples are µserver[5], Zeus[6] and kHTTPd[7]. This kind of web server always uses non-blocking systems calls to perform asynchronous I/O operation. SP server is able to overlap all the operations associated with the serving of many HTTP requests in the context of a single process. As a result, the overheads of context switching and process synch- ronization in the MT and MP architectures are avoided. However, relied on operating system's well support for asynchronous disk operations, SP web server may only provide excellent performance for cached workloads, where most requested content can be kept in main memory.

On workloads that exceed that capacity of the server cache, servers with MT or MP architecture usually perform best. SMT web server employs multiple threads to process requests. Some representative examples are KNOT[8] and Apache[9]. SMT web server can overlap the disk activity, CPU processing and network connectivity concurrently so that it improves the server's parallelism capability. However, SMT web server ignores that the processing of a request also can be divided into several phases among which there are some potential parallelism.

AMT web server allocates different tasks to different thread. Flash [10] and Tux [11] are examples for this kind. They use one thread to process all connections and several helper threads to deal with the I/O operation. They decrease blocking time and improve the efficiency of the service. However, it increases IPC cost between threads and helper threads and can not utilize system resource fully like SMT architecture.

From the discussion above, we can see that most web servers have some parallelism capability and their parallelism granularity is request. Once a request is blocked on some operation, the thread will stop. It's well known that thread resource is limited and costly in web system so this paper tries to find a way to reduce parallelism granularity and achieve inner-request parallelism. Open KETA divides the processing of a request into four phrases. Thread in different phases performs different function and doesn't intervene with each other just like different pipeline phase. In this frame, even if a request is blocked in one phase, threads in other phases still can process other requests. So the whole system performance is improved. In the following section, framework of open KETA is presented in Detail.

## 3 Framework of Open KETA

Open KETA is a kernel web server, the original developing intention of which is to improve web server's performance by transferring the processing of static requests from user space to kernel space. When overloaded, performance of web server in user space is not so well due to much copy and syscall cost. Now many web servers are implemented in kernel space, such as kHTTPd and TUX. Considering system stability, kernel space web server only processes static requests instead of complex dynamic requests, and that dynamic requests are redirected to user space web server such as Apache. What's more, measurements [12, 13] have suggested that the request stream at most web servers is dominated by static requests. Serving static requests quickly is the focus of many companies. Figure1 shows the processing flow of open KETA. For Linux already has a kernel web server TUX to accelerate requests processing, FreeBSD doesn't have yet, open KETA is implemented in FreeBSD kernel.

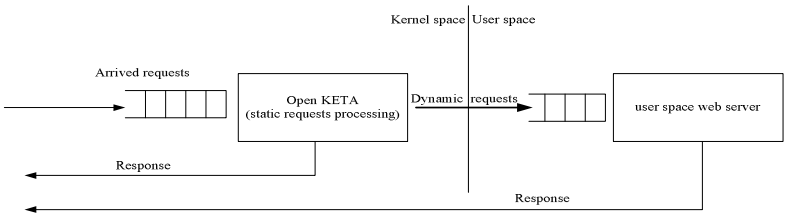


**Fig.1.** processing flow of open KETA

As introduced above, Open KETA divides the processing of request into four phrases: Accept Connection (accept), Socket Recv (receive), Data Process and Send Response (send) each of which has its own thread pool. Threads of different phases run in a pipeline-like manner. Partition of pipeline phases is not at random but with some principle. Firstly, coupling degree of different phase should be relatively low so

that threads in different phases could run concurrently. Secondly, depth of pipeline should be proper because too flat can't bring much parallelism and too deep will cause much scheduling cost.

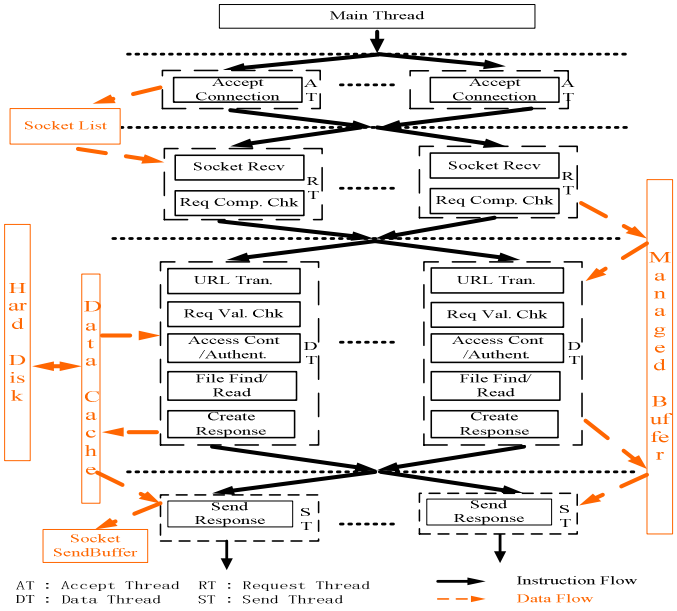


**Fig.2.** Framework of open KETA

Open KETA uses a managed buffer (MB) to transfer some control structures among all the phases. Furthermore, a software cache data cache (DC) is used to cache objects to reduce the times of disk access. DC and MB are initialized by a main thread as open KETA is loading. The framework of open KETA is presented in Figure 2.Main task of each phase is stated as followed:

- Accept phase is responsible for listening to the port. Applied with HTTP 1.1, once it finds a new arrived request which doesn't belong to an existing socket, it will create a new socket and set up connection, else if the socket is still keep alive, the request will stride over the accept phase and go to receive phase directly.
- Receive phase checks the completeness of http request and judges whether it's a static request. If not it will be redirected to web server in user space such as Apache. Here the socket the request belongs to is thrown to the socket list of user space web server directly in order to avoid the cost of recreating and destroying socket. If the arrived request is a static one, it is inserted to the task list of data process phase.
- Data process phase first validates requests and then judges whether the object requested is in DC or not by a hash map, if yes the response message is generated. It is worth saying that the response head is stored in DC as long as the object is in DC so that the response message can reuse the response head. Once

the object is not hashed in DC, get it from disk. If the conflict list of hash table is full or DC doesn't have enough space, some object will be washed out from DC.

- Just as its name implies, send phase sends the object requested back to clients. Open KETA utilizes Zero Copy which FreeBSD supports to reduce copy times and improve sending efficiency.

Owning to the Asymmetrical thread character, thread management is very important in open KETA. When should these threads be created, how to activate threads in each phase and how many threads should be allocated to each phase? The thread management will be presented in the following section.

# 4 Thread Management in Pipeline Architecture

## 4.1 Creation and Activation of Thread

In order to guarantee the real time service, all thread pools are initialized by a main thread when open KETA is loading. The number of thread is set empirically in a configuration file. As to the activation of threads, there are two ways in common: One is that a scheduler is specialized in this work in each thread group. After the execution, thread in previous group passes the result to the scheduler in this group. The scheduler will choose a thread based on some special rules. This method is extendable in implementation but the scheduler may be the bottleneck. Another way is that thread chooses the next-phase thread itself based on some rules. The advantage of this method is that cost of copy and control can be reduced but thread scheduling of each group is not transparent to other groups. Considering that open KETA is implemented in kernel, efficiency may be more important, so the latter is chosen and MB is used to transfer all control structures. When a thread has finished one task, it will check whether there are some unsettled tasks, if yes the thread continues to process another task else it will sleep and not wake up until thread in previous phases activate it.

## 4.2 Dynamic Tread Allocation

In section 3 the main task of each phase has been introduced respectively, from which we can see their burden is different owing to different length of execution code, different resource they mainly use, etc. With the changing of load, optimal thread number allocated to different phases is different. In this section, a feedback control model is proposed to control the thread allocation of each phase. First, we will analyze the runtime burden of each pipeline phase, from which thread allocation policy can be set with pertinence.

**Burden Analysis.** In web requests processing, CPU, memory and network bandwidth may be the consuming character of open KETA, threads in send phase may be first blocked in overloaded condition. Threads burden in data process are not as heavy as that in send phase since objects can be cached on DC. However, open KETA is running in kernel whose space can be used totally is 1G, so not all objects

have chance to be cached in DC. In this case open KETA has to access disk at times to get the object requested and replace some other objects with it in DC. Threads in accept phase may be most light-burdened since their main task is only creating socket. Threads in receive phase examine socket list to see whether there are some new requests, if yes some prearrange checks will be done on these requests. Main resource receive phase uses is CPU. From these analysis, we can see that work process of open KETA is like a four level funnel, work burden is more and more heavy from accept phase to send phase. When system is overloaded, thread allocation should be adjusted based on this special character of open KETA.

**Feedback Control Model of Thread Allocation.** When open KETA is loading, all thread pools are initialized with some empirical value. Although these values can suit many load conditions, they cannot deal with all the cases. Ideally, threads allocated to each phase should be adjusted based on their task list and server utilization. Figure 3 presents a feedback control model to achieve this. From this figure, we can see that a load monitor in open KETA gathers the queue length of the task list of each phase and the server utilization periodically, based on which decision is made to adjust thread allocation.

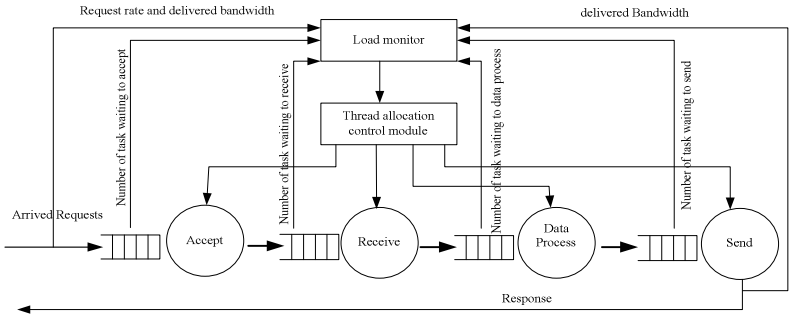


**Fig.3.** Thread allocation feedback control model

1. Load Monitoring

The objective of load monitor is to inspect the task list of each phase and quantify server utilization with a single value that summarizes resource consumption. The queue length of each task list can be easily obtained. It's noticed that the service time of a request can be decomposed into a fixed overhead and an object size dependent overhead [14], that is:

$$T(x) = c_1 x + c_2 \quad (1)$$

where x is the object size, $c_1$and $c_2$ are platform constants

For summing the service time of n requests:

$$\sum_{k=1}^{n} T(x_k) = c_1 \sum_{k=1}^{n} x_k + kc_2$$

And dividing by the length of the period t we obtain the system utilization U:

$$U = \frac{\sum_{k=1}^{n} T(x_k)}{t} = \frac{c_1 \sum_{k=1}^{n} x_k + kc_2}{t} = c_1 \frac{\sum_{k=1}^{n} x_k}{t} + c_2 \frac{k}{t} = c_1 W + c_2 R \quad (2)$$

From the Eq. (2) we get the quantify guideline of server utilization. We can repeat the experiment with different concurrent connections or URL sizes. Each time a different Wmax and Rmax are recorded, every case is corresponding to a fully utilized server. i.e., U =100%.Thus, each experiment yields a different point (Rmax, Wmax), then using linear regression coefficient c1, c2 are found. These two constants are obtained off-line and written into a configuration file.

2. Thread Allocation Control Module

Just as its name implies, the main task of thread allocation control module is to adjust thread allocation based on the information load monitor provides and some special character of open KETA. In section 4.2.1 it has been analyzed that all the pipeline phases of open KETA make up a four level funnel like structure, bottleneck would easily appear in send phase when overloaded, data process phase followed and then does the receive and accept phase. The number of thread allocated to each phase should be in accordance with this character. In order to avoid resource wasting, the initial value should not be too large. Supposed that the maximal thread number of open KETA is M which can be configured based on server's hardware condition and that the initial number of phase k is $P_k$ (k = 0...3, 0 is accept phase, 1 is receive phase, 2 is data process phase and 3 is send phase). When open KETA is loading, $\sum P_k$ is less than M. With the increase of concurrent connection, thread number of each phase is adjusted by the following formulas. Owning to the four level funnel structure the calculation sequence of $P_k^{i+1}$ is from $P_3$ to $P_0$:

$$\text{If} \quad \sum_{n=0}^{k} P_n^{\,i} + \sum_{n=k+1}^{3} P_n^{\,i+1} + T_{i+1}(a_k + b_k \Delta W_{i+1}) \le M$$

$$\text{then} \quad P_k^{i+1} = P_k^{i} + T_{i+1}(a_k + b_k \Delta W_{i+1}) \quad (3)$$

Else

$$P_k^{\,i+1} = M - \sum_{n=0}^{k-1} P_n^{\,i} \quad (4)$$

$$P_k^{\,i+1} = M - \sum_{n=k+1}^{3} P_n^{\,i+1} \quad (5)$$

$$P_k^{\,i+1} = M - \sum_{n=0}^{k-1} P_n^{\,i} - \sum_{n=k+1}^{3} P_n^{\,i+1} \quad (6)$$

Here it means when some tasks are waiting, thread number of the corresponding phase will be increased but the total number should not exceed M. $P_k^i$ is the current thread number of phase k and $P_k^{i+1}$ is the new adjusted one. $T_{i+1}$ is the queue length of the task list of phase k. $\Delta W_{i+1}$ represents $W_{i+1}$-$W_i$. If the $P_k^{i+1}$ is not an interger, $\lceil P_k^{i+1} \rceil$ is taken. $a_k$, $b_k$ can be well approached by some off-line experiment. But if $\sum_{k=0}^{3} P_k = M$, that thread number can not be increased, threads should be transferred from phase n (n<k) to phase k in order to release the burden of bottleneck phase. Thread number transferred is set empirically. It is worth saying that all threads can be implemented in a switch like manner to avoid destroying and creating thread frequently, here for limited length we do not discussed in detail. When Wi is low which means server is not so busy, thread number will be set back to the initial value by reducing the priority of some threads to a lower value of kernel thread just like destroying these threads so that other applications can utilize more system resource (because thread of other application can be schedule preferential). When the load of web server is increased again, Eq. (3) (4) (5) (6) are used to repeat the process.

## 5 Experimental Evaluation

The open KETA is implemented in FreeBSD 5.3 kernel. In order to contrast its performance with other web servers, we have done some experiments under different loads. In view of open KETA nature, all experiments are carried out only with static requests. The testing environment is made up of one server and three or five clients:

Server: SMP with two xeon 2.0G hz cpus,2GB memory, 36G SCSI hard disk and 1000M network card;

Clients: 2.4G hz cpu, 512M memory, 40GB 5400 rpm hard disk and 10-100M adaptive network card;

A testing tool SPECWeb99 is used to test the performance of the web servers. Platform for these web servers are Apache, open KETA in FreeBSD 5.3, Apache, tux, Zeus in Redhat Enterprise Linux v3.0. Note that the results of Table1, 2, 3 for open KETA do not include the thread allocation control model.

**Table 1.** results of 300 concurrent connections (3 clients)

| Tested object | Mean response time（ms） | Weighted bandwidth(bps） | Valid ＋ Invalid | Conforming | Operations per second |
|---|---|---|---|---|---|
| Apache(freebsd) | 410.0 | 303272.69 | 300+0 | 50 | 761 |
| Apache(Redhat) | 382.2 | 313600.49 | 300+0 | 56 | 765 |
| Tux | 320.4 | 373585.24 | 300+0 | 300 | 907 |
| Zeus | 342.5 | 357853.37 | 300+0 | 300 | 855 |
| Open KETA | 307.0 | 389930.76 | 300+0 | 300 | 954 |

**Table 2.** results of 600 concurrent connections (3 clients)

| Tested object | Mean response time（ms） | Weighted bandwidth(bps) | Valid＋Invalid | Conforming | Operations per second |
|---|---|---|---|---|---|
| Apache(freebsd) | 719.3 | 166083.41 | 600+0 | 0 | 771 |
| Apache(Redhat) | 758.2 | 157416.85 | 600+0 | 0 | 769 |
| Tux | 456.1 | 261535.11 | 600+0 | 600 | 1296 |
| Zeus | 536.1 | 228577.33 | 600+0 | 600 | 1100 |
| Open KETA | 352.4 | 356495.45 | 600+0 | 600 | 1702 |

When the concurrent connections are 1000, client may be the bottleneck (due to 10-100M network card), so more clients are used.

**Table 3.** results of 1000 concurrent connections (5 clients)

| Tested object | Mean response time（ms） | Weighted bandwidth(bps) | Valid＋Invalid | Conforming | Operations per second |
|---|---|---|---|---|---|
| Apache(freebsd) | 1077.7 | 110974.79 | 983+17 | 0 | 773 |
| Apache(Redhat) | 1247.2 | 95514.28 | 989+11 | 0 | 750 |
| Tux | 791.1 | 150558.99 | 999+1 | 678 | 1244 |
| Zeus | 992.5 | 126145.36 | 996+4 | 565 | 987 |
| Open KETA | 437.7 | 290117.36 | 1000+0 | 35 | 2285 |

We can see from the results, the performance of open KETA is much better than the web servers listed above. A simultaneous connection is considered conforming to the required bit rate if its aggregate bit rate is more than 320,000 bits/second, or 40,000 bytes/second. Other guidelines can be easily understood by their name. Table 4 presents the mean response time of open KETA with and without thread allocation model. Although thread adjustment brings additional system cost, we can see that the mean response time is reduced through the action of this model from the table.

**Table 4.** mean response time of open KETA with and without thread control model

| Concurrent connection / Policy | 300 | 600 | 800 | 1000 |
|---|---|---|---|---|
| Open KETA with thread allocation control model | 307.0 | 352.4 | 391.2 | 437.7 |
| Open KETA without thread allocation control model | 307.0 | 350.1 | 386.8 | 430.2 |

## 6 Conclusion

In this paper, we proposed the pipeline framework of a kernel web server open KETA. This web server has a four level funnel like work flow architecture, based on which a Feedback control model is in action to control thread allocation. This model can adjust thread number of each pipeline phase with the change of server load. The experiment results showed in section 5 validate the effectiveness of the control model.

Finally, although the number of threads is allotted based on the queue length of task list and the change of server utilization, actually this method do not handle transient behavior very well. As a part of the future work, we will try to find the relation between thread allocation and mean response time in different server load, through which thread number can be adjusted to a proper value promptly.

## 7 References


1. SHAN Zhi-Guang, LIN CHuang, et. al.: Web Quality of Service :A survey. JOURNAL OF COMPUTERS, Feb, 2004
2. C. Huitema.: Network vs. server issues in end-to-end performance. Keynote speech at Performance and Architecture of Web Servers 2000, Santa Clara, CA. http://kkant.ccwebhost .com/PAWS2000/huitema _keynote.ppt.
3. J. Gray and P. Shenoy.: Rules of thumb in data engineering. In Proc. of IEEE 16th Int'l Conf. on Data Engineering, pages 3-10, San Diego, CA, Apr. 2000.
4. Valeria Cardellini, Emiliano Casalicchio.: The State of the Art in Locally Distributed Web-server Systems. IBM research report, Computer Science, RC22209 (W0110-048) October 16, 2001.
5. Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, andJohn Tracey.: High-performance memory-baxde Web servers:Kernel and user-space performance. In Proceedings of the USENIX 2001 Annual Technical Conference, 2001.
6. Tim Brecht, David Pariag, Louay Gammo.: In:Proceedings of the USENIX 2004 Annual Technical Conference:General Track, June,2004.
7. Arjan wan de Ven.: kHTTPd Linux http accelerator. http://www.fenrus.demon.nl.
8. Rob von Behren, Jeremy Condit, et. al.: Why events are a bad idea for high- concurrency servers. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*,2003.
9. The Apache Group.: Apache http server project. http://www.apache.org.
10. Vivek S.Pai, Peter Druschel, Willy Zwaenepoel.: Flash:An efficient and portable Web server. In Proceedings of the USENIX 1999 Annual Technical Conference, Monterey,CA,June 1999.
11. Red Hat, Inc. *TUX 2.2* Reference Manual, 2002.
12. B. Krishnamurthy and J. Rexford. Web Protocols and Practices:HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement. Addison-Wesley, 2001
13. A. Feldmann. Web performance characteristics.: IETF plenary. http://www.research.att.com/anja/feldmann /papers.html.
14. Tarek F., Nina Bhatti.: Web server QOS management by adaptive content delivery.