Accurately Modeling Workload Interactions for Deploying Prefetching in Web Servers [*]

Xin Chen and Xiaodong Zhang

Department of Computer Science
College of William and Mary
Williamsburg, VA 23185, USA
{xinchen,zhang}@cs.wm.edu

Abstract

Although Web prefetching is regarded as an effective method to improve client access performance, the associated overhead prevents it from being widely deployed. Specifically, a major weakness in existing Web servers is that prefetching activities are scheduled independently of dynamically changing server workloads. Without proper control and coordination between the two kinds of activities, prefetching can negatively affect the Web services and degrade Web access performance. In this paper, we first develop an open queuing model to characterize detailed transactions in Web servers. Using this model, we analyze server resource utilization and average response time with different request arrival rates when prefetching is involved under different kinds of Web services. Guided by this model, we then design a responsive and adaptive prefetching scheme that dynamically adjusts the prefetching aggressiveness in Web servers. Our scheme not only prevents the Web servers from being overloaded, but it can also minimize the average server response time. We have effectively implemented this scheme on an Apache Web server. Our measurement-based performance evaluation shows our model can accurately predict the utilization of Web server resources and the correspondent average response time.

1 Introduction

With the popularity of the World Wide Web, latency perceived by the clients becomes an important factor of the quality of Web services. Web prefetching can effectively reduce the server response time, since idle server resources can be utilized for Web prefetching activities. Web prefetching techniques have been proposed for different kinds of Web services. For static Web objects, a prefetching scheme pre-loads those objects to be accessed possibly in the near future [16]. For dynamically generated Web objects, server response time can be reduced by pre-generating Web objects based on client access information [20]. For a search engine Web site, performance can be improved by pre-loading most related searching results [13]. For a CDN provider, pushing related objects to the proper CDN servers can achieve better performance than passively pulling [6]. An ideal prefetching scheme should have no negative effects on existing activities in the Web server while the reduction of client perceived server processing time can be maximized.

The potential effectiveness of Web prefetching has been widely investigated, and associated overheads have also been noticed. Possible network traffic overhead is analyzed in [9, 23]. It has been shown that if prefetched objects could be transferred at low rates, the network condition would be improved over that without prefetching. In order to avoid network overhead, a partial prefetch scheme [15] and prefetching between proxies and dial-up clients [10] are presented. Recently, researchers propose to utilize the unused network bandwidth for prefetching with marginal effects on existing traffic [16, 22], which makes Web prefetching more practical. The space overhead of building predictor trees can also been reduced by considering the specific access patterns [5]. The use of a threshold to adjust the aggressiveness of prefetching is analyzed in [11]. In contrast to the above cited studies, we look into the performance impact of prefetching and associated overhead in Web servers.

With the increase in types and amount of Web services, the server can easily become a bottleneck in Internet. A major concern about a wide deployment of Web prefetching is related to the associated overhead that may negatively affect the performance of the Web servers and the response time. In this study, we focus on evaluating existing techniques and providing new solutions to address a major weakness of these techniques — prefetching activities are scheduled independently of the dynamic server workloads. Therefore, if prefetching activities are not properly controlled and coordinated with Web servers, Web access performance can be significantly hurt.

Our research focus on Web servers in this paper is motivated by the structure of current Internet services that heavily rely on HTTP based on TCP protocols. Before an HTTP

[*]This work is supported in part by the National Science Foundation under grants CCR-0098055 and ACI-0129883.

request is sent to the Web server, a TCP connection must be first established though a three-way handshake mechanism. Once the TCP connection is established successfully, a client can send a series of HTTP requests to a Web server while the server uses the same connection to transfer the requested data to the client. The client-perceived response time comes from three parts: (1) the time to establish the TCP connections; (2) the time for Web servers to process requests; and (3) the time to send the response via the network. The last two parts account for the major delay. We further believe the Web server processing time is crucial to ensure the quality of Web services for the following two reasons:

- TCP connection time does not change much when the load on the server changes.

  As pointed out in [18], when the server is lightly loaded, the connection time can be ignored since the processing time is the major part. In fact, Web prefetching is always applied only when the idle resources are available. In our experiments, the average connection time is never larger than 10% of the average client-perceived response time. In order to reduce the connection overhead, `KeepAlive` directive is widely used in HTTP 1.0 and 1.1. In our experiments, we construct the requests with the directive following the format of HTTP 1.0.

- Prefetching requests will not increase the transmission time of regular requests.

  This is because (1) prefetching used for dynamic content does not consume additional network resources; and (2) a new TCP/IP protocol has been proposed [22] to avoid network resource competition between background traffic and existing traffic.

The effectiveness of designing and implementing an efficient control and coordination mechanism in Web servers mainly relies on insightful understanding and accurately characterizing the dynamic behaviors of Web servers. In this paper, we develop an open queuing model to characterize detailed transactions in Web servers. Using this model, we analyze server resource utilization and average response time with different request arrival rates when prefetching is involved with different kinds of Web services. Guided by this model, we design a responsive and adaptive prefetching scheme that dynamically adjusts the prefetching aggressiveness in Web servers. Our scheme not only prevents Web servers from being overloaded, it can also minimize the average server response time. We have effectively implemented this scheme on an Apache Web server. Our measurement-based performance evaluation shows our model can accurately predict the utilization of Web server resources and correspondent average response time.

2 Prefetching Performance Analysis

2.1 BCMP Queuing Networks

If the customers of a queuing network model have different service demands, it is regarded as a model of multiple class customers. Developed by Baskett *et al.* [4], the BCMP queuing networks allow different classes of customers, each with different service requirements and service time distributions other than exponential. Open, closed, and mixed networks are allowed. The queuing networks we have developed for prefetching in Web servers are based on an open model, which consists of K devices and C different classes of customers. The network state is denoted by a vector $\vec{n} = (\vec{n}_1, \vec{n}_1, ..., \vec{n}_k)$, where component $\vec{n}_i$ is a vector that represents the number of customers of each class at device i, which is $\vec{n}_i = (n_{i,1}, n_{i,2}, ..., n_{i,C})$. An open network allows customers to enter or leave the network while a closed network always has a constant number of customers in the network.

2.2 Prefetching Background

Prefetching Procedure
In order to evaluate prefetching effects on Web servers, we use a typical Web prefetching procedure: when the Web server receives a request from a client, it will make predictions based on the access history for the client and piggyback the results with the response. When the client receives the response, it sends requests for predicted objects if they are not cached in its browser. In order to fully exploit the potential effects of prefetching, some researchers suggest the client send messages to the Web server to notify of its status even if a hit happens. In our implementation, we cache the prediction results and use them when hits on the associated prefetched data happen. Our experiments indicate this method can significantly reduce the number of messages received by the server with marginal loss of hit ratios.

Prediction Structure
We use the Prediction by Partial Matching (PPM) method [7] to build the prediction tree, which is widely used in Web prefetching. The PPM model structure is represented by a set of trees, each of which is rooted by the first accessed URL of a sequence of Web URL accesses. Two parameters determine the tree structure. The parameter m is the number of previous accesses from the same client used to predict future accesses and the parameter l is the number of next accesses the PPM tree trying to predict. When the trees are used to make predictions, the last m accesses are matched from the roots of the trees. Every node in the tree structure has its access probability, which is defined as the ratio between its access frequency and the frequency of its parent node. A $threshold$ is set in the prefetching algorithm to select those nodes that have higher access probabilities than the predefined value. In our experiments, we always use the previous 2 accesses (m = 2) to predict the next immediate access (l = 1), which is commonly used in practical systems.

2.3 Queuing Networks for Web Services

In our analysis, we consider the situation where only a single Web server exists. The results can be easily extended to multiple servers. A typical Web server is connected to a LAN, which is connected to a router that connects the site to the ISP and then to the Internet. The queuing networking model is shown in Figure 1. It is an open queuing network model with a queue for each of the three components: the network interface card (NIC), the CPU and the disk.

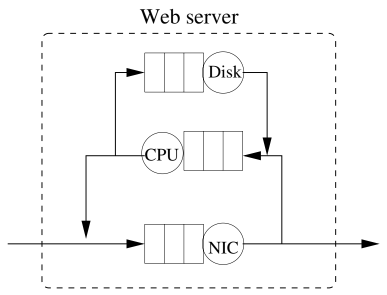

Figure 1. The queuing network model for Web services.

A typical Web service completes with several operation steps. For example, the Apache server [1] has the following procedure to process incoming requests:

1. Translating URI to the local filename,
2. Checking ID authorization,
3. Checking access authorization,
4. Access checking other than authorization,
5. Determining MIME type of the requested object,
6. Sending a response back to the client, and
7. Logging the request.

Different steps rely on different devices. For example, the first five steps mainly use the CPU while the sixth step normally needs the NIC, the CPU and the disk. With the improvement of Web techniques, a Web server provides various kinds of services to clients. Different kinds of requests have different resource requirements.

Parameter	Meaning
K	number of devices
C	number of request classes
λ_r	class r request arrival rate
$D_{i,r}$	service demand of class r requests at device i
$U_{i,r}$	utilization of device i by class r requests
U_i	utilization of device i by all requests
$R_{i,r}$	response time of class r requests at device i
R_r	class r average response time
R	average response time for all requests

Table 1. Input parameters for Web service models

When prefetching is applied for a specific class of Web requests, some requests can be prefetched while the rest are still explicitly requested by clients. Prefetching may change the resource requirements. Static object prefetching has a limit on the size of prefetched pages to avoid the overhead of wrong predictions. Dynamic content prefetching utilizes idle CPU cycles to pre-compute the results that may be requested by clients, but the results are not required to be transferred to clients until they are requested explicitly. Due to the variance of surfing behaviors in the Web, it is natural to model a Web site as an open network with multiple classes of requests.

In our analysis, we use a BCMP queuing network model to estimate the capacity of the Web server and the average server response time. In this section, we give the analysis in a general situation, where the number of devices and number of request classes are not limited. The parameters used in our analysis are shown in Table 1.

2.4 Capacity of A Web Server

The capacity of a Web server is measured by its system throughput that is a function of resource utilization. Here, we model the resource utilization of each device to set up system service thresholds.

Resource Utilization Without Prefetching
We can calculate the utilization of each device by summing the utilizations of each class of requests as follows:

$$U_i = \sum_{r=1}^{C} U_{i,r} = \sum_{r=1}^{C} \lambda_r D_{i,r}.$$

If a steady state solution exists, the maximum utilization of each device must be less than 100%, *i.e.*:

$$max_i \left\{ \sum_{r=1}^{C} \lambda_r D_{i,r} \right\} < 1.$$

It guarantees that no device will receive more service requests than it can handle.

Resource Utilization With Prefetching
When prefetching is applied in the Web server, for a given class of requests, two kinds of requests will be received by the server: $regular\ requests$ are explicitly sent by clients and $prefetch\ requests$ are automatically delivered by the browser with the prefetching function after it receives the prediction results from the server.

In order to accurately calculate resource utilization, we divide the class of requests into two parts when prefetching is applied to a specific class of requests.

- λ_r^r: regular request arrival rate of class r after prefetching is applied,
- λ_r^p: prefetch request arrival rate of class r,
- $D_{i,r}^r$: average service demand of regular requests of class r requests at device i, and
- $D_{i,r}^p$: average service demand of prefetch requests of class r requests at device i.

In consequence, additional C new classes of requests will be received by the Web server while the correspondent C original classes of requests may have different resource requirements from those without prefetching. In order to achieve a steady state, the maximum resource utilization in each device demanded by both regular and prefetch requests must be less than 100%, *i.e.*:

$$max_i \left\{ \sum_{r=1}^{C} (\lambda_r^r D_{i,r}^r + \lambda_r^p D_{i,r}^p) \right\} < 1.$$

2.5 Average Response Time

Average Response Time Without Prefetching
In order to compute the average server response time of all classes of requests, we need to calculate the average server response time for each class of requests. For class r requests, we have:

$$R_r = \sum_{i=1}^{K} R_{i,r} = \sum_{i=1}^{k} \frac{D_{i,r}}{1 - U_i}.$$

For all classes of requests, the average server processing time is:

$$R = \frac{\sum_{r=1}^{C} R_r \times \lambda_r}{\sum_{r=1}^{C} \lambda_r}.$$

Average Response Time with Prefetching
Since the prediction is based on history information, not all prefetched files are useful. The effectiveness depends on the accuracy of prediction and the prefetch hit ratios, which are determined by the prefetching threshold. Here are additional parameters we have defined:

- P_r: the prefetch hit ratio of class r customer, *i.e.* the percentage of all requests prefetched before they are requested explicitly by clients, which is determined by the prefetching threshold;
- A_r: the accuracy of prefetching of class r customer, *i.e.* the ratio between the accessed prefetched files and all prefetched files, which is also determined by the prefetching threshold;
- R_r^r: regular requests of class r response time; and
- R_r^p: prefetch requests of class r response time.

The regular request rate and the prefetch request rate for class r customer can be calculated by:

$$\lambda_r^r = \lambda_r * (1 - P_r),\ \lambda_r^p = \frac{\lambda_r * P_r}{A_r}.$$

The average response time for the two kinds of requests are:

$$R_r^r = \sum_{i=1}^{K} R_{i,r}^r = \sum_{i=1}^{k} \frac{D_{i,r}^r}{1 - U_i},\ R_r^p = \sum_{i=1}^{K} R_{i,r}^p = \sum_{i=1}^{k} \frac{D_{i,r}^p}{1 - U_i}.$$

The server response time of prefetch requests may not be perceived by clients, since plenty of time is available for prefetching requests to finish [10]. Thus, we assume all prefetch requests are completed before the clients require them explicitly. We define client-perceived average server response time R_{client} as the ratio between the total server response time of regular requests and the number of requests when no prefetching is applied. In order to minimize the client-perceived average server response time, we want to minimize the average response time for all requests expressed as follows:

$$R_{client} = \frac{\sum_{r=1}^{C} \lambda_r^r \times R_r^r}{\sum_{r=1}^{C} \lambda_r} = \frac{\sum_{r=1}^{C} (1 - P_r) \times \lambda_r \times R_r^r}{\sum_{r=1}^{C} \lambda_r}. \quad (2.1)$$

2.6 Summary of the Model

A Web server with multiple kinds of services makes the analysis complicated. BCMP queuing model provides an approximation tool to estimate the device utilizations and response time when multiple classes of requests exist. It also facilitates to account the effects of prefetching on Web servers. By estimating the server resource utilizations, we can control the prefetching aggressiveness and deduce the average server response time.

3 Adaptive Prefetching Algorithm

The analysis in the above section shows that the average response time for all requests, R, is determined by the arrival rates of different requests $\lambda_r, r = 1, ..., C$, and the prefetching threshold that determines the prefetching hit ratio $P_r, r = 1, ..., C$ and accuracy $A_r, r = 1, ..., C$.

This model guides us to develop an adaptive prefetching algorithm for the following objective: for given $\lambda_r, r = 1, ..., C$, we minimize R by adjusting the prefetching thresholds denoted as T.

3.1 Basic Idea of the Algorithm

In order to minimize the response time by selecting optimal prefetching thresholds, we need to compute the response time of each class of requests with a set of prefetching thresholds, which is composed of the response time of the class of requests in each device. Although different kinds of requests (e.g. dynamic, static) are divided into classes, the service demands of the requests in a class span in a large range. For example, the sizes of static requests have a heavy tail distribution [3, 8]. The average service demand in one class may not be accurate to represent that of the whole requests in this class. We further divide the requests into several groups based on the size of the service demands to improve the accuracy when we estimate the whole service demands of the class.

- $\vec{\lambda_r}$: the class r requests in different groups, and
- $\vec{l_r}$: the percentage of class r requests in different group requests.

In order to account for the service demands of the requests in each group for each class, we build a table named *Group Demand* to collect the information of the request distribution in different groups for all classes of requests and correspondent service demands. Furthermore, we build another table named *T, A and P* to record the relationships among prefetching thresholds, prefetching accuracies and prefetching hit ratios for different groups of requests for all classes, respectively.

The procedure of computing average response time of one device (device i) consists of five steps. The input parameters are all classes of request arrival rates of both regular and prefetch requests and the output is the average response time of the device for each group of requests of all classes.

Step 1: Estimating $\lambda_r, r = 1, ..., C$.

In order to estimate server response time, we need to know the request arrival rates and the service demands of each class of requests. When prefetching is used, we are not able to observe the request arrival rates directly since a part of requests have been prefetched. However, from the previous analysis, for a specific class of requests using prefetching with a given prefetching threshold, we can compute the request arrival rates without prefetching by the following equation:

$$\lambda_r = (\lambda_r^r + \lambda_r^p)/(1 - P_r + \frac{P_r}{A_r}).$$

Step 2: Estimating $\vec{\lambda_r}, r = 1, ..., C$.

The service demands of each class of requests can be computed by analyzing the server logs or monitoring the server utilization in real time. The estimation accuracy can be improved by dividing all requests in a class into different groups. From the table *Group Demand*, we know the distribution of class r requests in predefined groups $\vec{l_r}$. Now we have $\vec{\lambda_r} = \lambda_r * \vec{l_r}$.

Step 3: Estimating $\vec{\lambda_r^p}$ and $\vec{\lambda_r^r}, r = 1, ..., C$.

As we have pointed out, prefetch requests may have different service demands and we need to characterize the request streams including regular and prefetch requests. In our scheme, we also compute the prediction accuracy and hit ratio for each group of requests in a given class. If we know the request arrival rate at each group without prefetching, we can calculate each group request rate of prefetch and regular requests, which are represented by $\vec{\lambda_r^p}$ and $\vec{\lambda_r^r}$. In this step, multiple prefetching thresholds are used.

Step 4: Estimating U_i.

The total service demands (service utilization) of one class of requests using prefetching can be approximated by multiplying the service demand of each group of requests (defined as $\vec{D_r}$) with the request rate of them as follows:

$$U_{i,r} = (\vec{\lambda_r^p} + \vec{\lambda_r^r}) * \vec{D_r}.$$

The device utilization U_i is equal to summing the utilization of all kinds of requests on device i.

Step 5: Estimating $\vec{R_{i,r}^r}$ and $\vec{R_{i,r}^p}, r = 1, ..., C$.

We can compute the device average response time for each class of requests as follows:

$$\vec{R_{i,r}^r} = \frac{\vec{D_{i,r}^r}}{1 - U_i}, \ \vec{R_{i,r}^p} = \frac{\vec{D_{i,r}^p}}{1 - U_i},$$

where the $\vec{R_{i,r}^r}$ and $\vec{R_{i,r}^p}$ represent the device i response time for class r regular and prefetch requests at each group.

Furthermore, the average response time of a Web server for every class of request can be computed by summing all response time of the individual devices. The server average response time for all classes of requests can be calculated by equation 2.1. By repeating the above procedure for all selected thresholds for every class of requests, the optimal thresholds are the set that achieve the minimal server response time for all requests.

3.2 Workload

The workload used in our experiments is from the WorldCup 98 Web site, which is available from the Internet Traffic Archives [12]. It was one of the busiest Web sites in 1998 and represents a popular Web site trace available in the public domain [2]. During the collection period, there were 33 different HTTP servers at four geographic locations, although not all of them were in use for the entire collection period. During this 92 day period (April 26th - July 26th, 1998), 1,352,804,107 requests were received by the Web site. We have conducted our experiments on more than 10 days' traces and all results are consistent. We select the 46th day, one of the busiest days during this period, in our presentation. During that day, a total of 252,753 clients sent 50,395,084 requests for 8,265 data objects on the servers. A total of 187 GBytes were transferred from the servers to all clients. In order to simplify the presentation in the rest of the paper, we only use a single class of requests in our experiments and evaluations.

3.3 Accounting for the Heavy Tail Distribution in Service Demands

A heavy-tailed distribution has been observed in Web traffic [3, 8]. A random variable that follows a heavy-tailed distribution varies in a large range of sizes, with many occurrences as small mixed with a small amount of occurrences as large. In the Web environment, a large percentage of HTTP requests are for small objects and a small percentage of requests are for objects that are several magnitudes larger than the small objects.

As pointed out in [8, 14], average results for the whole population of requests would have little statistical meaning due to the large size variability of objects. The accuracy of service demand estimation can be improved by dividing the requests into a number of groups by the object sizes.

In our experiments, we also define the maximal size of objects to be prefetched, which should also be considered when categorizing the requests. For the WorldCup 98 traces, we divide all requests into 4 groups by their sizes.

Group	Size *KB*	Percent	Avg. *KB*	CPU	NIC
1	[0, 5)	84.6%	1.1	0.4 ms	0.09 ms
2	[5, 20)	11.9%	10.8	0.8 ms	0.89 ms
3	[20, 100)	3.4%	33.6	1.7 ms	2.78 ms
4	[100, ∞)	0.83%	1149.7	44.2 ms	95.3 ms

Table 2. Characterizations of Different Group (Table Group Demand)

In order to measure the service demands for every group, we measure the CPU, NIC and disk utilizations by changing request arrival rate with different parallel connections. The CPU and disk utilizations are taken from the Linux `/proc` file system and the the NIC utilization is taken by using `tcpdump`. In our experiments, we find the disk utilization is marginal and we do not count it in our following analysis. The service demands are measured by using a PIII 500 MHz computer with 128 MByte memory and a 100 Mbps

Ethernet card as the Web server. The size ranges for different groups and the service demands are shown in Table 2. In our experiments, when a single connection is used to send and receive requests, the requests in every group have higher service demands. With a large number of parallel connections (larger than 10), the service demands are decreased. Considering a busy server connected by a lot of clients, we use service demands at multiple parallel connections (10 connections in our measurement) as service demands for the requests in each group.

3.4 Relationships among Thresholds, Accuracies and Hit Ratios

	Group 1		Group 2		Overall	
T	$A_1(\%)$	$P_1(\%)$	$A_2(\%)$	$P_2(\%)$	$A(\%)$	$P(\%)$
0.01	32	93	21	85	30	89
0.05	44	82	35	71	43	78
0.15	54	49	49	44	54	47
0.25	57	24	51	25	56	24
0.35	62	14	57	13	61	13
0.45	53	5.9	49	6.7	52	5.8
0.55	49	2.1	47	3.0	49	2.2

Table 3. Relationships Among Threshold, Accuracy, and Hit Ratio (Table T, A, and P).

As we discussed in the previous section, in order to estimate the prefetch effects, we first need to build a table to collect the accuracies and hit ratios for all possibly used thresholds. In most prefetching algorithms, in order to reduce the overhead of wrong prefetching requests, the maximal size of the prefetched objects is defined. In this example, the upper bound size of the prefetched objects is 20 KBytes, so only the Web objects in group 1 and 2 can be prefetched. Table 3 shows the results from traces of day 46. For those thresholds larger than 0.6, the hit ratios are less than 1% and have very limited influence on the response time. We only focus on prefetching thresholds from 0.01 - 0.55.

3.5 Prefetching Performance Evaluation

Request Arrival Rate Estimation
In order to evaluate the CPU utilization when prefetching is applied, we need to know both the regular and prefetch request arrival rates, which can be calculated by using Table 3. The estimated values for a specific request arrival rate (λ = 100) are shown in Figure 2. It clearly shows that the regular request arrival rates can be effectively reduced by setting low prefetching thresholds, while the prefetch request arrival rates are increased very fast. For example, if the threshold is set to 0.01, the regular request rate is reduced to 10 requests/seconds and the prefetch request rate is close to 300 requests/second. Compared with the request arrival rate without prefetching (100 requests/second), the load on the server is increased significantly.

Server Capacity
As we pointed in the previous section, the server capacity

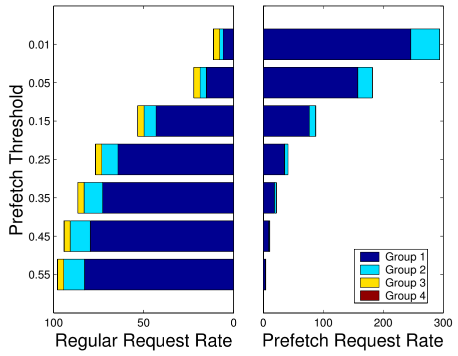

Figure 2. The request distributions for each group when different thresholds are used.

is determined by the bottleneck device, which is the CPU in our experiments. In order to estimate the CPU utilization, we need to know the request arrival rates in all groups and the service demands of each group of requests. The estimated server CPU utilization for a specific request arrival rate (100 requests/second) is shown in Table 4. As expected, when a low threshold is set, the CPU utilization is increased with the increment of request arrival rates. However, the CPU utilization is increased at a slower pace than the request rate due to a large percentage of small-sized requests. For example, if the threshold is set to 0.01, the request arrival rate is increased from 100 to 305, while the CPU utilization is increased from 5.4% to 15.5%.

Group	0.01	0.05	0.15	0.25	0.35	NP
1	251.77	172.86	119.92	99.92	91.86	84.6
2	49.96	27.59	17.35	14.76	13.06	11.9
3	3.4	3.4	3.4	3.4	3.4	3.4
4	0.083	0.083	0.083	0.083	0.083	0.083
Sum	305.2	203.9	140.8	118.2	108.4	100
D_{CPU}	155.1	103.4	73.0	62.7	57.9	54.0
U_{CPU}	15.5%	10.3%	7.3%	6.3%	5.8%	5.4%

Table 4. CPU Utilization Comparison among Different Thresholds

Response Time
Once we have the device utilizations, we can use the service demands to estimate the average response time of each device.

If we assume all prefetched files can be fully downloaded before the clients explicitly request them, the server processing times of prefetch requests are not perceived by clients. Since only a part of requests (regular requests) are explicitly sent out by clients, the client-perceived server response time can be reduced after prefetching is deployed. For example, when the request arrival rate is 100 requests/second and the threshold is 0.01, the request rate explicitly sent by the clients is decreased to 11.2 requests/second.

The response time with variable request arrival rates are

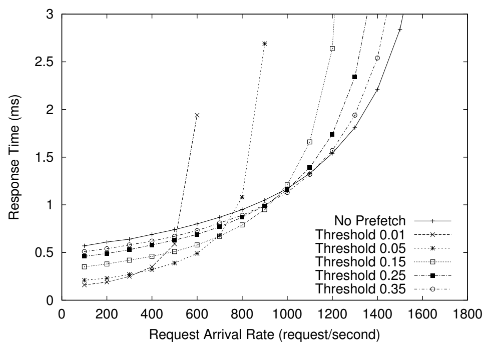

Figure 3. CPU response time comparisons among different thresholds.

shown in Figure 3. It is clear that low thresholds should be used when the server's load is light while high thresholds should be set for heavy server's loads. It is also interesting to observe that prefetching can bring marginal benefits if the request rate is larger than 1000 requests/second. However, most Web servers are utilized far below the maximal capacities needed to accommodate the bursty request streams. Thus, prefetching can be an effective way in most cases in practice. In our experiments, the response time is normally below 3 ms, which limits the performance improvement of prefetching. If we consider dynamic content with response time of hundreds of milliseconds, prefetching can significantly reduce the response time perceived by clients.

4 Prototype and Results

4.1 Implementation

We have implemented the proposed prefetching methods on Apache 2.0.40 [1]. The Web server will make predictions for all requests. When it prepares to serve the responses, prediction results will be added in the header and sent back to the clients. When persistent connections are used, a connection can receive both types of requests from the same client. Two kinds of headers have been added in the request: `Regular` and `Prefetch`, which are included in regular requests and prefetch requests, respectively. When more than one previous URLs are used to make predictions, the clients also include previous access information with the header. In order to make it compatible with the currently deployed protocols, every request without the additional headers is considered as a regular request. A new header `Prediction` in the server's response header is added to convey the prediction results.

Periodically, the Web server checks if the threshold is suitable for the current average request arrival rate. A counter is used to record the number of requests received in the last period. When the predefined time slice is reached, a maintenance procedure is called. First, it estimates the average request rate in the last period. Then it checks if the current threshold is suitable and selects an optimal one for the current load level. When the request rate is lower than a predefined value, the minimal prefetch threshold value is set safely. For the WorldCup 98 traces, we repeat the procedure every 10 seconds.

4.2 Experiment Settings

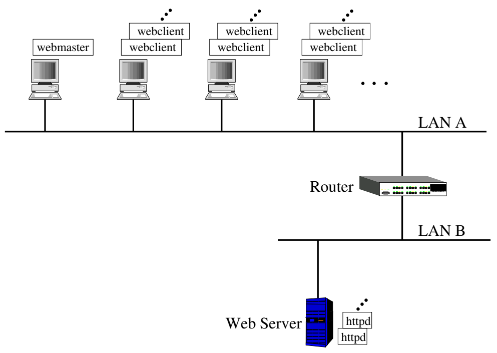

Figure 4. The experimental environment.

The clients are simulated by an enhanced WebStone 2.5 [21]. In our enhanced WebStone 2.5, every client process has a URL list recording the requested URLs and the time to send the request, which is extracted from the real Web server traces. In the current implementation of WebStone, the maximum number of `webclient` processes is set to 1024, due to the limitation of the number of sockets for a process to open simultaneously. In order to make it scalable, we assign every webclient process several real clients, represented by several URL lists. In this way, we can simulate more than 1024 clients by using a relatively small number of processes on a limited number of machines.

The experimental environment is shown by Figure 4, where simulated clients and the Web server are located in two different 100 Mbps Ethernet LANs connected by a router. On the client side, a number of clients, which are represented by processes (webclient) distributed on a number of computers, send requests to the server. The `webmaster` is running on another computer to manage the webclient processes and collect the results from all webclients. A number of `httpd` processes are created in the server to process incoming requests.

In our experiments, 100 to 1000 webclients, each in charge of 15 real clients, are equally distributed on 5 computers with Intel 2.26 GHz P4 CPU and 1 GByte memory. The Web server uses a computer with Intel 500 MHz PIII CPU with 128 MBytes memory and a 100 Mbps Ethernet card. The Apache Web server uses the `worker` module to support threads for high performance and uses default parameters in Apache `httpd.conf` to set the initial number of server processes and maximum number of simultaneous client connections. All machines run the Linux operating system with kernel 2.4.18.

All webclient processes read client traces extracted from traces of day 46 from the WorldCup 98 Web server site. We use a 10-minute section in the trace of day 46. During the 10 minutes, 15,304 clients visited WorldCup 98 Web site. Since the cache status has influential effects on hit ratios, in order to make the results more accurate, before we start our experiments, we use a previous hour period trace to warm

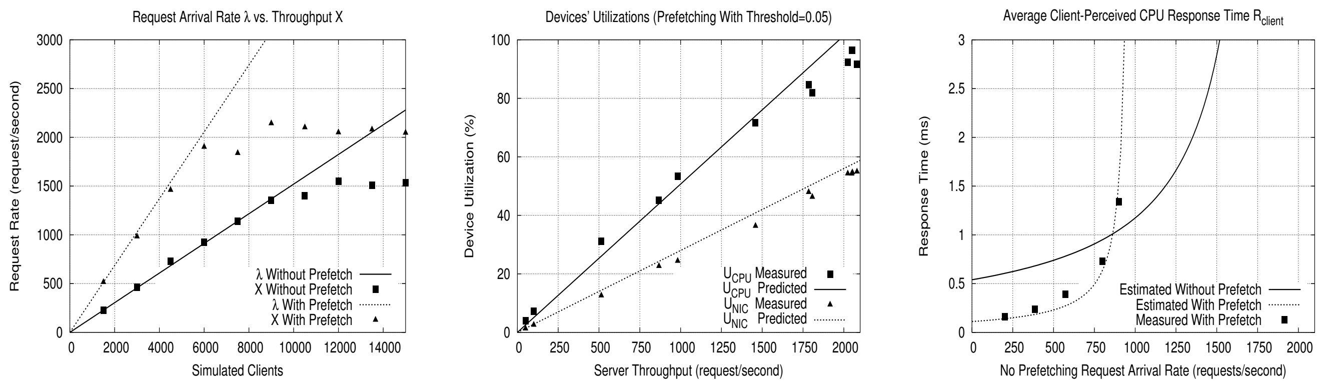

Figure 5. The left figure shows server throughputs (represented by X) and request arrival rates λ of no prefetching scheme and a prefetching scheme with threshold 0.05. The middle figure gives the comparisons of resource utilizations U_{CPU} and U_{NIC} between the estimated values from the model analysis and measured ones in our experiments when prefetching is used with threshold 0.05. The right figure gives the comparisons of client-perceived server response times R_{clint}.

up the browser caches. The server uses 300,000 requests in the day 46 trace as the training data to build the predictor.

4.3 Performance Comparisons

The effectiveness of our adaptive prefetching model is evaluated by two metrics.

- The accuracy of estimating server's capacity. An accurate estimation is important to prevent the Web server from being overloaded.
- The accuracy of estimating server's response time. This value is essential to select the optimal threshold to adjust the aggressiveness of Web prefetching.

As an example, we select a commonly used threshold 0.05 to present the related results in this paper.

Server Throughput
By adjusting the number of clients, a request stream with a variable request rate is used to test the performance of the Web server. Starting from 1,500 clients, an additional 1,500 clients will be added every minute, which results in a total of 15,000 clients at the end of in the 10 minute test. The left figure of Figure 5 presents the request arrival rates λ and correspondent server throughputs (represented by X) for no prefetching and prefetching with threshold 0.05 schemes.

In both schemes, the server's throughput is always equal to the request arrival rate until the server's capacity is reached. For the prefetching scheme with threshold 0.05, the server can process up to 2000 requests per second while it can only process up to 1500 requests per second in no prefetching scheme. There are two reasons: a) the average service demand per request in the prefetching is lower than that without prefetching. b) the ratio of small sized requests in total requests is increased when prefetching is used.

Server Capacity
The server resource utilizations for different server throughputs are shown in the middle figure in Figure 5, which presents the results for a system with prefetching with threshold 0.05. The two lines are estimated CPU and NIC utilizations while the points are the measured values in our experiments. For both devices, our measured results are within 5% from the predicted values. With the increase of the server throughput, the CPU utilization is not increased strictly proportionally. When the throughput is approaching the server's capacity, the CPU utilization is increased at a lower pace. As pointed in [8, 14], the service demands can be higher due to the burstiness in Web request rates. When the request arrival rate is close to the server's capacity, the effects of burstiness on the service demands is reduced.

Average Response Time
The server's response time is the sum of all device response times. In our experiments, the CPU is the bottleneck and the NIC response time is proportional to the server throughput. In the right figure in Figure 5, we present the average CPU response time comparisons between the experimental results and the values predicted by our model. It shows the comparison of the average client-perceived CPU response time between the no prefetching scheme and the prefetching scheme with threshold 0.05. The x-axis is the request arrival rate when prefetching is not used. The two lines are estimated average client-perceived CPU response time for no prefetching scheme and prefetching scheme with threshold 0.05. The points are the calculated values from our experimental results. Prefetching with fixed thresholds 0.05 can reduce the response time for light loads (e.g., less than 800 requests/second), while prefetching increases the response time for heavy loads. Our predicted results are accurate, which can be used to optimize the prefetching aggressiveness.

5 Conclusion

In this paper, we analyze the effects of Web prefetching on Web server's average response time. Although prefetching is well known for its potential to improve Web latency, our study shows it can also increase the Web server response time without a proper control. We have made the following contributions in this study:

- We have developed an open queuing network model to characterize the interactions between prefetching and Web server workloads. The model is validated and proved to be accurate by trace-driven simulations and Web server measurements.
- Based on our analysis, we propose an adaptive prefetching scheme to prevent Web servers from being negatively influenced by prefetching. By monitoring the request arrival rate, the Web servers can adjust the threshold adaptively and periodically to maximize performance.
- We have also effectively implemented our prefetching scheme on an Apache server. The measurement results show that our methods are accurate and responsive, and demonstrates that if prefetching is used properly, the response time perceived by clients can be significantly improved.

We are currently testing our adaptive prefetching scheme embedded in the Apache server in a real-world Internet environment, where diverse types of Web accesses are conducted, including dynamic and multimedia contents.

Acknowledgment: This work is a part of an independent research project sponsored by the US National Science Foundation for program directors and visiting scientists. We thank Phil Kearns for providing the experiment environment, and Bill Bynum for reading the paper and his suggestions. Comments from anonymous referees are constructive and helpful.

References

[1] Apache HTTP Server Project. http://httpd.apache.org/.

[2] M. Arlitt, and T. Jin, "Workload characterization of the 1998 World Cup Web site", IEEE Network, Vol. 14, No. 3, May/June 2000, pp. 30-37.

[3] P. Barford, and M. Crovella, "Generating representative Web workloads for network and server performance evaluation", *Proceedings of Performance'98/SIGMETRICS'98*, Madison, Wisconsin, July 1998, pp. 151-160.

[4] F. Baskett, K. Chandy, R. Muntz, and F. Palacios, "Open, closed, and mixed networks of queues with different classes of customers", *Journal of the ACM*, Vol. 22, No. 2, April 1975, pp. 248-260.

[5] X. Chen, and X. Zhang, "A popularity-based prediction model for Web prefetching", *IEEE Computer*, Vol. 36, No. 3, March 2003, pp. 59-65.

[6] Y. Chen, L. Qiu, W. Chen, L. Nguyen and R. H. Katz, "Clustering Web content for efficient replication", *Proceeding of the 10th IEEE International Conference on Network Protocols*, Paris, France, November 2002, pp. 165-174.

[7] J. G. Cleary, and I. H. Witten, "Data compression using adaptive coding and partial string matching", *IEEE Transactions on Communications*, Vol. 32, No. 4, April 1984, pp. 396-402.

[8] M. Crovella, and P. Barford, "Self-similarity in World Wide Web traffic: evidence and possible causes", *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer systems*, Philadelphia, Pennsylvania, May 1996, pp. 160-169.

[9] M. Crovella, and P. Barford, "The network effects of prefetching", *Proceedings of the IEEE INFOCOM'98 Conference*, San Francisco, California, March/April 1998, pp. 1232-1240.

[10] L. Fan, P. Cao, W. Lin, and Q. Jacobson, "Web prefetching between low-bandwidth clients and proxies: potential and performance", *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Atlanta, Georgia, May 1999, pp. 178-187.

[11] Z. Jiang, and L. Kleinrock, "An adaptive network prefetch scheme", *IEEE Journal on Selected Areas of Communication*, Vol. 17, No. 4, April 1998, pp. 358-368.

[12] Lawrence Berkeley National Laboratory, URL: http://ita.ee.lbl.gov/

[13] R. Lempel, and S. Moran, "Optimizing result prefetching in Web search engines with segmented indices", *Proceedings of VLDB 2002*, Hong Kong, China, August 2002, pp. 370-381.

[14] D. A. Menasc, and V. A. F. Almeida, "Capacity planning for Web services: metrics, models, and methods", Prentice Hall, New Jersey, 2002.

[15] J. I. Khan, and Q. Tao, "Partial prefetch for faster surfing in composite hypermedia", *USENIX Symposium on Internet Technology and Systems*, San Francisco, California, March 2001, pp.13-24.

[16] R. Kokku, P. Yalagandula, A. Venkataramani, M. Dahlin, "A non-interfering deployable Web prefetching system", *USENIX Symposium on Internet Technology and Systems*, Seattle, Washington, March 2003.

[17] T. M. Kroeger, D. D. E. Long, and J. C. Mogul, "Exploiting the bounds of Web latency reduction from caching and prefetching", *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, April 1997, pp. 13-22.

[18] D. P. Olshefski, J. Nieh, and D. Agrawal, "Inferring client response time at the Web server", *Proceedings of SIGMETRICS 2002*, Marina Del Rey, California, June 2002, pp. 160-171.

[19] V. N. Padmanabhan, and J. C. Mogul, "Using predictive prefetching to improve World Wide Web latency", *Computer Communication Review*, Vol. 26, No. 3, July 1996, pp. 22-36.

[20] S. Schechter, M. Krishnan, and M. D. Smith, "Using path profiles to predict HTTP requests", *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, April 1998, pp. 457-467.

[21] G. Trent, and M. Sake, "Webstone: the first generation in http server benchmarking", February 1995. Silicon Graphics White Paper.

[22] A. Venkataramani, R. Kokku and M. Dahlin, "System support for background replication", *Proceedings of Fifth Operating Systems Design and Implementation conference*, Boston, Massachusetts, December 2002.

[23] Z. Wang, and J. Crowcroft, "Prefetching in World-Wide Web", *Proceedings of IEEE Globecom*, London, England, November 1996, pp. 28-32.