NPS: A Non-interfering Deployable Web Prefetching System*

Ravi Kokku Praveen Yalagandula Arun Venkataramani Mike Dahlin
Department of Computer Sciences, University of Texas at Austin
{rkoku, ypraveen, arun, dahlin}@cs.utexas.edu

Abstract

We present NPS, *a novel non-intrusive web prefetching system that (1) utilizes only spare resources to avoid interference between prefetch and demand requests at the server as well as in the network , and (2) is deployable without any modifications to servers, browsers, network or the HTTP protocol. NPS's self-tuning architecture eliminates the need for traditional "thresholds" or magic numbers typically used to limit interference caused by prefetching, thereby allowing applications to improve benefits and reduce the risk of aggressive prefetching.*

NPS avoids interference with demand requests by monitoring the responsiveness of the server and accordingly throttling the prefetch aggressiveness, and by using TCP-Nice, a congestion control protocol suitable for low priority transfers. NPS avoids the need to modify existing infrastructure by modifying HTML pages to include Javascript^{TM} code that issues prefetch requests and by wrapping the server infrastructure with several simple external modules that require no knowledge of or no modifications to the internals of existing servers. Our measurements of the prototype under a web trace indicate that NPS is both non-interfering and efficient under different network load and server load conditions. For example, in our experiments with a loaded server with little spare capacity, we observe that a threshold-based prefetching scheme causes response times to increase by a factor of 2 due to interference, whereas prefetching using NPS decreases response times by 25%.

1 Introduction

A number of studies have demonstrated the benefits of web prefetching [12, 17, 24, 25, 32, 33, 42, 52]. And the attractiveness of prefetching appears likely to rise in the future as the falling prices of disk storage [14] and network bandwidth [41] make it increasingly attractive to trade increased consumption of these resources to improve response time and availability and thus reduce human wait time [7].

Despite these benefits, prefetching systems have not been widely deployed because of two concerns: interference and deployability. First, if a prefetching system is too aggressive, it may interfere with demand requests to the same service (self-interference) or to other services (cross-interference) and hurt overall system performance. Such interference may occur at the server, in the communication network or at the client. Second, if a system requires modifications to the existing HTTP protocol [19] , it may be impractical to deploy. The large number of deployed clients and networks in the Internet makes it difficult to change clients, and the increasing complexity of servers [23, 26, 28, 46, 55] makes it difficult to change servers. What we therefore need is a prefetching system that (a) avoids interference at clients, networks, and servers and (b) does not require changes to the HTTP protocol and the existing infrastructure (client browsers, networks and servers).

In this paper, we make three contributions. First, we present NPS, a novel **n**on-interfering **p**refetching **s**ystem for the web that – (1) avoids interference by effectively utilizing only spare resources on the servers and the network and (2) is deployable with no modifications to the HTTP protocol and existing infrastructure. To avoid interference at the server, NPS monitors the server load externally and restricts the prefetch load imposed on it accordingly. To avoid interference in the underlying network, NPS uses TCP-Nice for low-priority network transfers [51]. Finally, it uses a set of heuristics to control resource usage at the client. To work with existing infrastructure, NPS modifies HTML pages to include JavaScript^{TM} code to issue prefetch requests, and wraps the server infrastructure with simple external modules that require no knowledge of, or no modifications to the internals of existing servers. Our measurements of the prototype under real web load trace indicate that NPS is both non-interfering and efficient under different network

*This work was supported in part by the Texas Advanced Research Program, the IBM Center for Advanced Studies, and an IBM University Partnership Award. Dahlin was also supported by a Sloan Research Fellowship.

and server load conditions. For example, in our experiments on a heavily loaded network with little spare capacity, we observe that a threshold-based prefetching scheme causes response times to increase by a factor of 7 due to interference, whereas prefetching using NPS contains this increase to less than 30%.

Second, and on a broader note, we propose a self-tuning architecture for prefetching that eliminates the need for traditional "threshold" magic numbers that are typically used to limit the interference that prefetching inflicts on demand requests. This architecture divides prefetching into two separates tasks – (i) prediction and (ii) resource management. The predictor proposes prioritized lists of high-valued documents to prefetch. The resource manager limits the number of documents to prefetch and schedules the prefetch requests to avoid interference with demand requests and other applications. This separation of concerns has three advantages – (i) it simplifies the design and deployment of prefetching systems by eliminating the need to choose appropriate thresholds for an environment and update them with changing conditions, (ii) it reduces the risk of interference caused by prefetching that relies on manually set thresholds, especially during periods of unanticipated high load, (iii) it increases the benefits of prefetching by prefetching more aggressively than would otherwise be safe during periods of low or moderate load. We believe that these advantages would also apply to prefetching systems in many environments beyond the web.

Third, we explore the design space for building a web prefetching system, given the requirement of avoiding or minimizing changes to existing infrastructure. We find that it is straightforward to deploy prefetching that ignores the problem of interference, and it is not much more difficult to augment such a system to avoid server interference. Extending the system to also avoid network interference is more involved, but doing so appears feasible even under the constraint of not modifying current infrastructure. Unfortunately, we were unable to devise a method to completely eliminate prefetching's interference at existing clients: in our system prefetched data may displace more valuable data in a client cache. It appears that a complete solution may eventually require modifications at the client [6, 8, 44]. For now, we develop simple heuristics that reduce this interference.

The rest of the paper is organized as follows. Section 2 discusses the requirements and architecture of a prefetching system. Sections 3, 4 and 5 present the building blocks for reducing interference at servers, networks and clients. Section 6 presents the prefetch mechanisms that we develop to realize the prefetching architecture. Section 7 discusses the details of our prototype and evaluation. Section 8 presents some related work and section 9 concludes.

2 Requirements and Alternatives

There appears to be a consensus among researchers on a high level architecture for prefetching in which a server sends a list of objects to a client and the client issues prefetch requests for the objects on the list [9, 36, 42] . This division of labor allows servers to use global object access patterns and service-specific knowledge to determine what should be prefetched, and it allows clients to filter requests through their caches to avoid repeatedly fetching objects. In this paper, we develop a framework for prefetching that follows this organization and that seeks to meet two other important requirements: self tuning resource management and deployability without modifying existing protocols, clients, proxies, or servers.

2.1 Resource Management

Services that prefetch should balance the benefits against the risk of interference. Interference can take the form of *self-interference*, where a prefetching service hurts its own performance by interfering with its demand requests, and *cross-interference*, where the service hurts the performance of other applications on the prefetching client, other clients, or both.

Limiting interference is essential because many prefetching services have potentially unlimited bandwidth demand, where incrementally more bandwidth consumption provides incrementally better service. For example, a prefetching system can improve hit rate and hence response times by fetching objects from a virtually unlimited collection of objects that have non-zero probabilities of access [5, 8], or by updating cached copies more frequently [10, 50, 52].

Interference can occur at any of the critical resources in the system.

- **Server:** Prefetching consumes extra resources on the server such as processing time, memory space and disk.
- **Network:** Prefetching causes extra data packets to be transmitted over the network, potentially increasing queuing delays and packet drops.
- **Client:** Prefetching results in extra processing at clients. Furthermore, aggressive prefetching can pollute a browser's memory and disk caches.

A common way of achieving balance between the benefits and costs of prefetching is to select a threshold and prefetch objects whose estimated probability of use before modification or eviction from the cache exceeds that threshold [17, 29, 42, 52]. There are at least two problems with such "magic number"-based approaches. First, it is difficult for even an expert to set thresholds to optimum values to balance costs and benefits—although thresholds relate closely to the benefits of prefetching, they have little obvious relationship to the costs of prefetching [7, 21]. Second, appropriate thresholds to balance costs and benefits may vary over time as client, network, and server load conditions change over seconds (e.g., changing workloads or network congestion [56]), hours (e.g., diurnal patterns), and months (e.g., technology trends [7, 41]).

Our goal is to construct a self-tuning resource module that prevents prefetch requests from interfering with demand requests. Such an architecture will simplify the design of prefetching systems by separating the tasks of prediction and resource management. Prediction algorithms may specify arbitrarily long lists of the most beneficial objects to prefetch sorted by benefit, and the resource management module issues requests for these objects and ensures that these requests do not interfere with demand requests or other system activities. In addition to simplifying system design, such an architecture could have two performance advantages over statically set prefetch thresholds. First, such a system can reduce interference – when resources are scarce, it would reduce prefetching aggressiveness. Second, such a system may increase the benefits of prefetching when resources are plentiful by allowing more aggressive prefetching than would otherwise be considered safe.

2.2 Deployability

Many proposed prefetching mechanisms suggest modifying the HTTP/1.1 protocol [4, 15, 17, 42], to create a new request type for prefetching. An advantage of extending the protocol is that clients, proxies, and servers could then distinguish prefetch requests from demand requests and potentially schedule them separately to prevent prefetch requests from interfering with demand requests [15]. However, such mechanisms are not easily deployable because modifying the protocol implies modifying the widely-deployed infrastructure that supports the current protocol including existing clients, proxies, and servers. As web servers evolve and increase in their complexity, requests may traverse not only a highly optimized web server [43, 49, 54, 55] but also a number of other complex modules such as commercial databases, application servers or virtual machines for assembling dynamic content (e.g., Apache tomcat for executing Java Servlets and JavaServer pages), distributed cluster services [2, 23], and content delivery networks. Modifying servers to separate prefetch requests from demand requests maybe complex or infeasible under such circumstances.

If interference were not a concern, a simple prefetching system could easily be built with the present infrastructure, where clients can be made to prefetch without any modifications to the protocol. For example, servers can embed JavaScript code or a Java applet [20], to fetch specified objects over the network and load them into the browser cache. An alternative way is to add invisible frames to the demand content that include and thereby preload the prefetch content.

In this paper, we adapt such techniques to avoid interference while maintaining deployability.

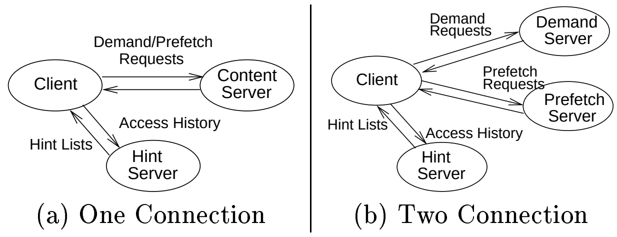
2.3 Architectural Alternatives

In this subsection, we present an overview of two alternative architectures to build a prefetching system. The high-level description in this section is intended only to provide a framework for discussing resource management strategies at the server, network, and client in sections 3 through 5. These architectures and resource management strategies are pertinent regardless of whether prefetching is implemented using a new protocol or by exploiting existing infrastructure. In Section 6, we describe how our implementation realizes one of these architectures in an easily deployable way.

We begin by making the following assumptions about client browsers:

- For easy deployability of the prefetching system, browsers should be unmodified.
- Browsers match requests to documents in their caches based on (among other parameters) the server name and the file name of the object on the server. Thus files of the same name served from different servers are considered to be different.
- Browsers may multiplex multiple client requests to a given server on one or more persistent connections [19].

Figure 1 illustrates what we call the one-connection and two-connection architectures respectively. In both architectures, clients send their access histories to the *hint server* and get a list of documents to prefetch. The hint server uses either online or offline prediction algorithms to compute the hint lists consisting of the most probable documents that the users might request in the future.



(a) One Connection (b) Two Connection

Figure 1. Design Alternatives for a Prefetching System

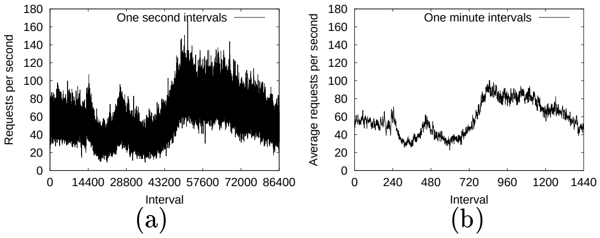


Figure 2. Server loads averaged over (a) 1-second and (b) 1-minute intervals for the IBM sporting event workload.

2.3.1 One Connection

In the one connection architecture (Figure 1(a)), a client fetches both demand and prefetch requests from the same content server. Since browsers multiplex requests over established connections to servers, and since browsers do not differentiate between demand and prefetch requests, each TCP connection may interleave prefetch and demand requests and responses.

Sharing connections can cause prefetch requests to interfere with demand requests for network and server resources. If interference can be avoided, this system is easily deployable. In particular, objects fetched from the same server share the domain name of the server. So, unmodified client browsers can use cached prefetched objects to service demand requests.

2.3.2 Two Connection

In the two connection architecture(Figure 1(b)), a client fetches demand and prefetch requests from different servers or from different ports on the same server. This architecture thus segregates demand and prefetch requests on separate network connections.

Although the two connection architecture simplifies the mechanisms for reducing interference at the server by segregation, this solution appears to complicate the deployability of the system. Objects with the same names fetched from different servers are considered different by the browsers. So, browsers can not directly use the prefetched objects to service demand requests.

2.3.3 Comparison

In the following sections, we show how to address the limitations of both architectures.

- Some of the techniques we develop for avoiding interference are useful for the one connection architecture, but some are less so. In particular, our strategy for reducing interference at servers is based on end-to-end performance and is equally applicable to the one and two connection architectures. Conversely, the techniques we use to avoid network interference appear much easier to apply to the two-connection than the one-connection architecture.
- Despite the apparent deployability challenges to the two connection architecture discussed above, we find that the same basic technique we use to make unmodified browsers prefetch data for the one connection architecture can be adapted to support the two connection architecture as well.

We conclude that both architectures are tenable in some circumstances. If server load is the primary concern and if network load is known not to be a major issue, then the one connection prototype may be simpler than the two connection prototype. At the same time, the two connection prototype is feasible and deployable and manages both network and server interference. Given that networks are a globally shared resource, we recommend the use of two connection architecture in most circumstances.

3 Server Interference

An ideal system for avoiding server interference would cause no delay to demand requests in the system and utilize significant amounts of any spare resources on servers for prefetching. Such a system needs to cope with, and take advantage of, changing workload patterns over various time scales. HTTP request traffic arriving at a server often is bursty with the burstiness being observable at several scales of observation [13] and with peak rates exceeding the average rate by factors of 8 to 10 [37]. For example, Figure 2 shows the request load on an IBM server hosting a major sporting event during 1998 averaged over 1-second and 1-minute intervals. It is crucial for the prefetching system to be responsive to such bursts to balance utilization and risk of interference.

3.1 Alternatives

There are a variety of ways to prevent prefetch requests from interfering with demand requests at servers.

Local scheduling Server scheduling can help use the spare capacity of existing infrastructure for prefetching in a non-interfering manner. In principle, existing schedulers for processor, memory [29, 31, 44], and disk [35] could prevent low-priority prefetch requests from interfering with high-priority demand requests. Furthermore, as these schedulers are intimately tied to the operating system, they should be highly efficient in delivering whatever spare capacity exists to prefetch requests even over fine time scales.

Note that local scheduling is equally applicable to both one- and two-connection architectures.

For many services, however, server scheduling may not be easily deployable for two reasons. First, although several modern operating systems support process schedulers that can provide strict priority scheduling, few provide memory, cache or disk schedulers that isolate prefetch requests from demand requests. Second, even if an operating system provides the needed support, existing servers would have to be modified to differentiate between prefetch and demand requests with scheduling priorities as they are serviced [3]. This second requirement appears particularly challenging given the increasing complexity of servers, in which requests may traverse not only a highly-tuned web server [43, 49, 54, 55] but also a number of other complex modules such as commercial databases, application servers or virtual machines for assembling dynamic content (e.g., Apache tomcat for executing Java Servlets and JavaServer pages), distributed cluster services [2, 23], and content delivery networks.

Separate prefetch infrastructure An intuitively simple way of avoiding server interference is to use separate servers to achieve complete isolation of prefetch and demand requests. In addition to the obvious strategy of providing separate demand and prefetch machines in a centralized cluster, a natural use of this strategy might be for a third-party "prefetch distribution network" to supply geographically distributed prefetch servers in a manner analogous to existing content distribution networks. Note that this alternative is not available to the one-connection architecture.

However, separate infrastructure needs extra hardware and hence may not be an economically viable solution for many web sites.

End-to-end monitoring A technique based on end-to-end monitoring estimates the overall load (or spare capacity) on the server by periodically probing the server with representative requests and measuring the response times of the replies. Low response times indicate that the server has spare capacity and high response times indicate that the server is loaded. Based on such an estimate, the monitor utilizes the spare capacity on the server by controlling the number and aggressiveness of prefetching clients.

An advantage of end-to-end monitoring is that it requires no modifications to existing servers. Furthermore, it can be used by both one- and two- connection prefetching architectures. The disadvantage of such an approach is that its scheduling precision is likely to be less than that of a local scheduler that has access to the internal state of the server and operating system. Moreover, an end-to-end monitor may not be responsive enough to bursts in load over fine time scales.

In the following subsections, we discuss issues involved in designing an end-to-end monitor in greater detail, present our simple monitor design, and evaluate its efficacy in comparison to server scheduling.

3.2 End-to-end Monitor Design

Figure 3 illustrates the architecture of our monitor-controlled prefetching system. The monitor estimates the server's spare capacity and sets a *budget* of prefetch requests permitted for an interval. The hint server adjusts the load imposed by prefetching on the server by ensuring that the sum across the hint lists returned to clients does not exceed the budget. Our monitor design must adress two issues: (i) budget estimation and (ii) budget distribution across clients.

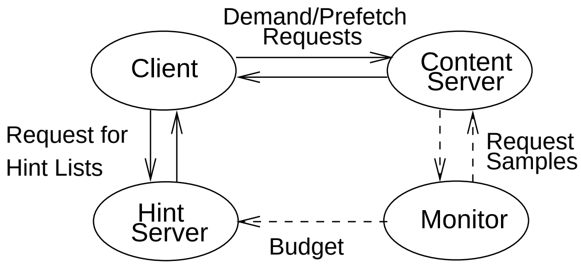


Figure 3. A Monitored Prefetching System

Budget estimation The monitor periodically probes the server with HTTP requests to representative objects and measures the response times. The monitor increases the budget when the response times are below the objects' *threshold* values and decreases the budget otherwise.

As probing is an intrusive technique, choosing an appropriate rate of probing is a challenge. A high rate makes the monitor more reactive to load on the server, but also adds extra load on the server. On the other hand, a low rate makes the monitor react slowly, and can potentially lead to interference to the demand requests. Similarly, the exact policy for increasing and decreasing the budget must balance the risk of causing interference against underutilization of spare capacity.

Budget distribution The goal of this task is to distribute the budget among the clients such that (i) the load due to prefetching on the server is contained within the budget for that epoch and is distributed uniformly over the interval, (ii) a significant fraction of the budget is utilized over the interval, and (iii) clients are responsive to changing load patterns at the server. The two knobs that the hint server can manipulate to achieve these goals are (i) the size of the hint list returned to the clients and (ii) the subset of clients that are given permission to prefetch. This flexibility provides a freedom to choose from many policies.

3.3 Monitor Prototype

Our prototype uses simple, minimally tuned policies for budget estimation and budget distribution. Future work may improve the performance of our monitor.

The monitor probes the server in epochs, each approximately 100 ms long. In each epoch, the monitor collects a response time sample for a representative request. In the interest of being conservative – choosing non-interference even at the potential cost of reduced utilization – we use an additive increase(increase by 1), multiplicative decrease (reduce by half) policy. AIMD is commonly used in network congestion control [30] to conservatively estimate spare capacity in the network and be responsive to congestion. If in five consecutive epochs, the five response time samples lie below a threshold, the monitor increases the budget by 1. While taking the five samples, if any sample exceeds the threshold, the monitor sends another probe immediately to check if the sample was an outlier. If even the new sample exceeds the threshold, indicating a loaded server, the monitor decreases the budget by half and restarts collecting the next five samples.

In our simple prototype, we manually supply the representative objects's threshold response times. However, it is straightforward because of the predictable pattern in which response times vary with load on server systems – a nearly constant value of response time for low load followed by a sharp rise beyond the "knee" for high load. As part of our future work, we intend to make the monitor automatically pick thresholds in a self-tuning manner.

The hint server distributes the current budget among client requests that arrive in that epoch. We choose to set the hint list size to the size of one document (a document corresponds to a HTML page and all embedded objects). Our policy lets clients to return quickly for more hints and thus be more responsive to changing load patterns on the server. Note that returning larger hint lists would reduce the load on the hint server, but it would reduce the system's responsiveness and its ability to avoid interference. We control the number of simultaneously prefetching clients, and thus the load on the server, by returning to some clients a hint list of zero size and a directive to wait until the next epoch to fetch the next hint list. For example, if B denotes the budget in the current epoch, and N the expected number of clients in that epoch, D the number of files in a document, and τ the epoch length, the hint server accepts a fraction $p = min(1, \frac{B \cdot \tau}{N \cdot D})$ of requests to prefetch on part of clients in that epoch and returns hintlists of zero length for other requests. Note that other designs are possible. For example, the monitor can integrate with the prefetch prediction algorithm to favor prefetching by clients for which the predictor can identify high-probability items and defer prefetching by clients for which the predictor identifies few high-value targets.

Since the hint server does not a priori know the number of client requests that will come in an epoch, it estimates that value with the number of requests that come in the previous epoch. If more than the estimated number of requests arrive in a epoch, the hint server replies with list of size zero and a directive to retry in the next epoch to those extra requests. If fewer clients arrive, some of the budget can get wasted. However, in the interest of avoiding interference, we choose to allow such wastage of budget.

In the following Section 3.3.1, we evaluate the performance of our prototype with respect to the goals of reducing interference and reaping significant spare bandwidth and compare it with the other resource management alternatives.

3.3.1 Evaluation

In evaluating resource management algorithms, we are mainly concerned with interference that prefetching could cause and less with the benefits obtained. We therefore abstract away prediction policies used by services by prefetching sets of dummy data from arbitrary URLs at the server. The goal of the experiments is to compare the effectiveness of different resource management alternatives in avoiding server interference against the ideal case (when no prefetching is done) with respect to the following metrics: (i) *cost:* the amount of interference in terms of demand response times and (ii) *benefit:* the prefetch bandwidth.

We consider the following resource management algorithms for this set of experiments:

1. No-Prefetching: Ideal case, when no prefetching is done or when we use a separate prefetching infrastructure.
2. No-Avoidance: Prefetching with no interference avoidance with fixed aggressiveness. We set the aggressiveness by setting *pfrate*, which is the number of documents prefetched for each demand document. For a given service, a given prefetch threshold will correspond to some average *pfrate*. We use fixed *pfrate* values of 1 and 5.
3. Scheduler: As a simple local server scheduling policy, we choose *nice*, the process scheduling utility in Unix. We again use fixed *pfrate* values of 1 and 5. This simple server scheduling al-

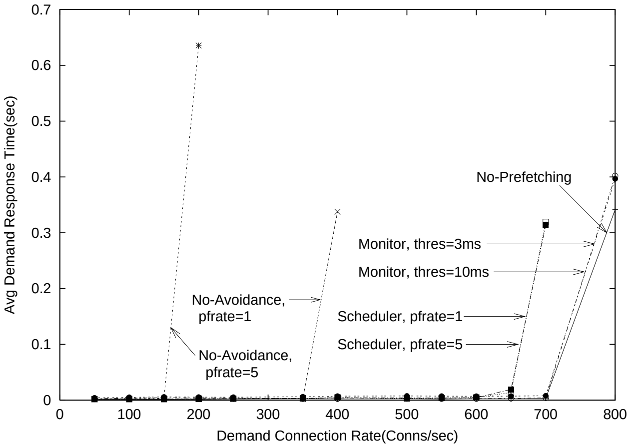


Figure 4. Effect of prefetching on demand throughput and response times with various resource management policies

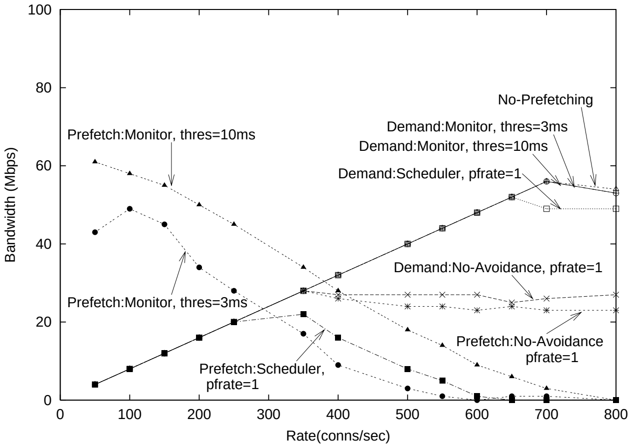


Figure 5. Prefetch and demand bandwidths achieved by various algorithms

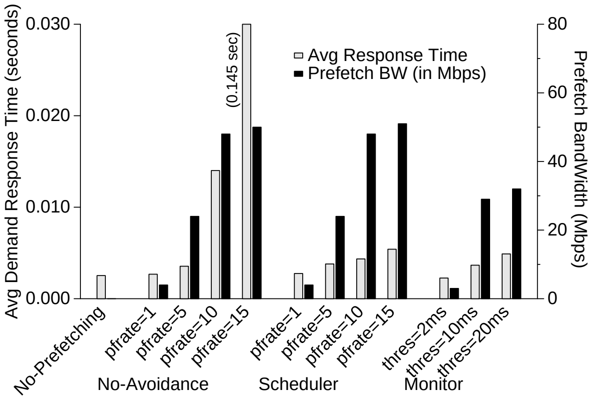


Figure 6. Performance of No-Avoidance, Scheduler and Monitor schemes on the IBM server trace

gorithm is only intended as a comparison; more sophisticated local schedulers may better approximate the ideal case.

4. Monitor: We perform experiments for two threshold values of 3ms and 10ms.

For evaluating algorithms 2 and 4, we set up one server serving both demand and prefetch requests. These algorithms are applicable in both one connection and two connection architectures. Our prototype implementation of algorithm 3 requires that the demand and prefetch requests be serviced by different processes and thus is applicable only to the two connection architecture. We use two different servers listening on two ports on the same machine, with one server run at a lower priority using the Linux *nice*. Note that the general local scheduling approach is equally applicable to the one-connection architecture with more intrusive server modifications.

Our experimental setup includes Apache HTTP server [1] running on a 450MHz Pentium II, with 128MB of memory. To generate the client load, we use httperf [38] running on four different Pentium III 930MHz machines. All machines run the Linux operating system.

We use two workloads in our experiments. Our first workload generates demand requests to the server at a constant rate. The second workload is a one hour subset of the IBM sporting event server trace, whose characteristics are shown in Figure 2. We scale up the trace in time by a factor of two, so that requests are generated at twice the original rate, as the original trace barely loads our server.

Constant workload Figure 4 shows the demand response times with varying demand request arrival rate. The graph shows that both Monitor and Scheduler algorithms closely approximate the behavior of No-Prefetching in not affecting the demand response times. Whereas, the No-Avoidance algorithm with fixed *pfrate* values significantly damages both the demand response times and the maximum demand throughput.

Figure 5 shows the bandwidth achieved by the prefetch requests and their effect on the demand bandwidth. The figure shows that No-Avoidance adversely affects the demand bandwidth. Conversely, both Scheduler and Monitor reap spare bandwidth for prefetching without much decrease in the demand bandwidth. Further, at low demand loads, a fixed pfrate prevents No-Avoidance from utilizing the full available spare bandwidth. The problem of too little prefetching when demand load is low and too much prefetching when demand load is high illustrates the problem with existing threshold strategies. As hoped, the Monitor tunes prefetch aggressiveness of the clients such that essentially all of the spare bandwidth is utilized.

IBM server trace In this set of experiments, we compare the performance of the four algorithms for the IBM server trace. Figure 6 shows the demand response times and prefetch bandwidth in each case. The graph shows that the No-Avoidance case affects the demand response times significantly as *pfrate* increases. The Scheduler and Monitor cases have less adverse effects on the demand response times.

These experiments show that resource management is an important component of a prefetching system because overly aggressive prefetching can significantly hurt demand response time and throughput while timid prefetching gives up significant bandwidth. They also illustrate a key problem with constant non-adaptive magic numbers in prefetching such as the threshold approach that is commonly proposed. The experiments also provide evidence of the effectiveness of the monitor in tuning prefetch aggressiveness of clients to reap significant spare bandwidth while keeping interference at a minimum.

4 Network Interference

Mechanisms to reduce network interference could, in principle, be deployed at clients, intermediate routers, or servers. For example, clients can reduce the rate at which they receive data from the servers using TCP flow control mechanisms [48]. However, it is not clear how to set the parameters to such mechanisms or how to deploy them given existing infrastructure. Prioritization in routers that provide differentiated service to prefetch and demand packets can avoid interference effectively [47]. However, router prioritization is not easily deployable in the near future. We focus on server based control because of the relative ease of deployability of server based mechanisms and their effectiveness in avoiding both self- and cross-interference.

In particular, we use a transport level solution at the server – TCP-Nice [51]. TCP-Nice is a congestion control mechanism at the sender that is specifically designed to support background data transfers like prefetching. Background connections using Nice operate by utilizing only spare bandwidth in the network. They react more sensitively to congestion and backoff when a possibility of congestion is detected, giving way to foreground connections. In our previous study [51], we provably bound the network interference caused by Nice under a simple network model. Furthermore, our experimental evidence under wide range of conditions and workloads shows that Nice causes little or no interference and at the same time reaps a large fraction of the spare capacity in the network.

Nice is deployable in the two connection context without modifying the internals of servers by configuring systems to use Nice for all connections made to the prefetch server. A prototype of Nice runs on Linux currently, and it should be straight-forward to port Nice to other operating systems. The other way to use Nice in non-Linux environments is to put a Linux machine running Nice in front of the prefetch server and make the Linux machine serve as a reverse proxy or a gateway.

It appears to be more challenging to use Nice in the one connection case. In principle, the Nice implementation allows flipping a connection's congestion control algorithm between standard TCP (when serving demand requests) and Nice (when serving prefetch requests). However, using this approach for prefetching faces a number of challenges: (1) Flipping modes causes packets already queued in the TCP socket buffer to inherit the new mode. Thus, demand packets queued in the socket buffer may be sent at low-priority while prefetch packets may be sent at normal-priority, thus causing network interference. Ensuring that demand and prefetch packets are sent in the appropriate modes would require an extension to Nice and a fine-grained coordination between the application and the congestion control implementation. (2) Nice is designed for long network flows. It is not clear if flipping back and forth between congestion control algorithms will still avoid interference and gain significant spare bandwidth. (3) HTTP/1.1 pipelining requires replies to be sent in the order requests were received, so demand requests may be queued behind prefetch requests, causing demand requests to perceive increased latencies. One way to avoid such interference may be to quash all the prefetch requests queued in front of the demand request. For example, we could send a small error message (eg. HTTP response code 307 – "Temporary Redirect" with a redirection to the original URL) as a response to the quashed prefetch requests.

Based on these challenges, it appears simpler to use the two connection architecture when the network is a potential bottleneck. A topic for future work is to explore these challenges and determine if a deployable one connection architecture that avoids network interference can be devised.

5 Client Interference

Prefetching may interfere with the performance of a client in at least two ways. First, prefetch requests consume processing cycles and may, for instance, delay rendering of demand pages. Second, prefetched data may displace demand data from the cache and thus hurt demand hit rates for the prefetching service or other services.

As with the interference at the server discussed above, interference between client processes could, in principle, be addressed by modifying the client browser (and, perhaps, the client operating system) to use a local processor scheduler to ensure that processing of prefetch requests never interferes with processing of demand requests. Lacking that option, we resort to a simpler approach: as described in Section 6, we structure our prefetch mechanism to ensure that processing prefetch requests does not begin until after the loading and rendering of the demand page, including all inline images and recursive frames. Although this approach will not help reduce cross-interference with other applications at the client, it may avoid a potentially common case of self-interference of the prefetches triggered by a page delaying the rendering of that page.

Similarly, a number of storage scheduling algorithms exist that balance caching prefetched data against caching demand data [6, 8, 31, 44]. Unfortunately, all of these algorithms require modifications to the cache replacement algorithm.

Because we assume that the client cannot be modified, we resort to two heuristics to limit cache pollution caused by prefetching. First, in our system, services place a limit on the ratio of prefetched bytes to demand bytes sent to a client. Second, services can set the `Expires` HTTP header to a value in the relatively near future (e.g., one day in the future) to encourage clients to evict prefetched document earlier than they may otherwise have done. These heuristics have an obvious disadvantage: they resort to magic numbers similar to those in current use, and they suffer from the same potential problems: if the magic numbers are too aggressive, prefetching services will interfere with other services, and if they are too timid, prefetching services will not gain the benefits they might otherwise gain. Fortunately, there is reason to hope that performance will not be too sensitive to this parameter. First, disks are large and growing larger at about 100% per year [14] and relatively modest-sized disks are effectively infinite for many client web cache workloads [52]. So, disk caches may absorb relatively large amounts of prefetch data with little interference. Second, hit rates fall relatively slowly as disk capacities shrink [5, 52], which would suggest that relatively large amounts of polluting prefetch data will have relatively small effects on demand hit rate.

Figure 7 illustrates the extent to which our heuristics can limit the interference of prefetching on hit rates. We use the 28-day UCB trace of 8000 unique clients from 1996 [22] and simulate the hit rates of 1 MB, 10 MB and 30 MB per-client caches. Note that these cache sizes are small given, for example, Internet Explorer's defaults of using 3% of a disk's capacity (e.g., 300 MB of a 10 GB disk) for web caching. On the x-axis, we vary the number of bytes of dummy prefetch data per byte of demand data that are fetched after each demand request. In this exper-

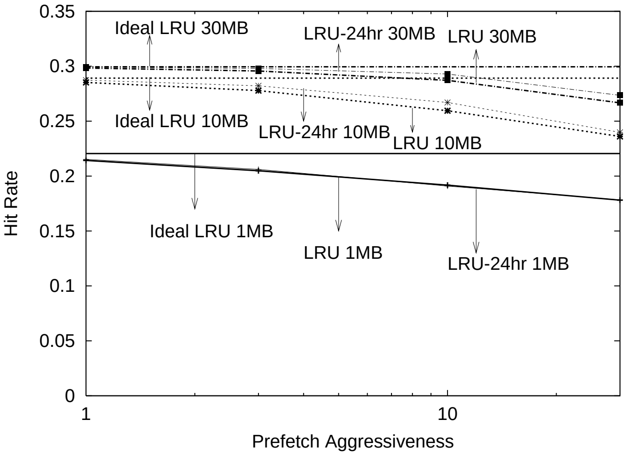


Figure 7. Effect of prefetching on demand hit rate

iment, 20% of services use prefetching at the specified aggressiveness and the remainder do not, and we plot the demand hit rate of the *non*-prefetching services. Ideally, these hit rates should be unaffected by prefetching. As the graph shows, hit rates fall gradually as prefetching increases, and the effect shrinks as cache sizes get larger. For example, if a client cache is 30 MB and 20% of services prefetch aggressively enough that each prefetches ten times as much prefetch data as the client references demand data, demand hit rates fall from 29.9% to 28.7%.

6 Prefetching Mechanism

Figure 8 illustrates the key components of the one and two connection architectures. The one-connection mechanism consists of an unmodified client, a content server that serves both demand and prefetch requests, a munger that modifies content on the content server to activate prefetching and a hint server that gives out hint lists to the client to prefetch. The hint server also includes a monitor that probes the content server and estimates the spare capacity at the server and accordingly controls the number of prefetching clients.

The two-connection prototype, along with the components above, also consists of a prefetch server that is a copy of the demand server (running either on a separate machine or on a different port on the same machine) and a front-end that intercepts certain requests to the demand server and returns appropriate redirection objects as described later, thereby obviating any need to modify the original demand server.

In the following subsections, we describe the prefetching mechanisms for the one and two connection architectures.

6.1 One-connection Prefetching Mechanism

Content modification The munger augments each HTML document with pieces of JavaScript and HTML code that cause the client to prefetch.

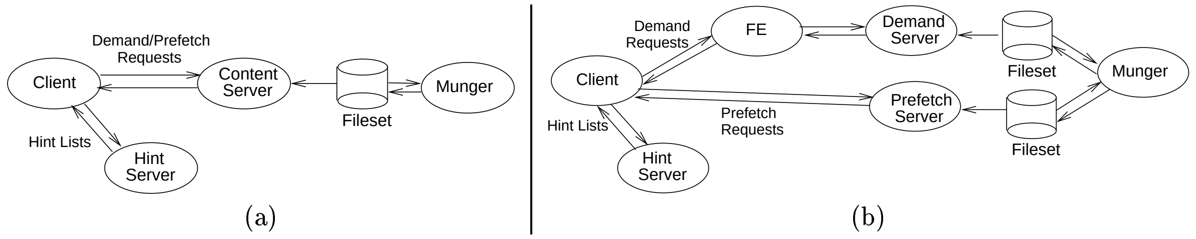


Figure 8. Prefetching mechanisms for (a) one connection and (b) two connection architectures.

On demand fetch

1. Client requests an augmented HTML document.
2. When an augmented HTML document (Figure 9) finishes loading into the browser, the `pageOnLoad()` function is called. This function calls `getPfList()`, a function defined in `pfalways.html` (Figure 10). The file `pfalways.html` is loaded within every augmented HTML document. `pfalways.html` is cacheable and hence does not need to be fetched everytime a document gets loaded.
3. `getPfList()` sends a request for `pflist.html` to the hint server with the name of the enclosing document, the name of the previous document in history (the enclosing document's referer) and TURN=1 as extra information embedded in the URL.
4. The hint server receives the request for `pflist.html`. Since the client fetches a `pflist.html` for each HTML document (even if the HTML document is found in the cache), the client provides the hint server with a history of accesses to aid in predicting hint lists. In Figure 10, PFCOOKIE contains the present access (`document.referrer`) and the last access (`prevref`) by the client. The hint updates the history and predicts a list of documents to be prefetched by the client based on that client's history and the global access patterns. It puts these predictions into the response `pflist.html` such as shown in 11, which it returns to the client.
5. `pflist.html` replaces `pfalways.html` on the client. After `pflist.html` loads, the `preload()` function in its body preloads the documents to be prefetched from the prefetch server (which is same as the demand server in the one connection case).
6. After all the prefetch documents are preloaded, the `myOnLoad()` function calls `getMore()` that replaces the current `pflist.html` by fetching a new version with `TURN=TURN`+1.

Steps 5 and 6 repeat until the hint server has sent everything it wants, at which point the hint server returns a `pflist.html` with no `getMore()` call. When there is not enough budget left at the server, the hint server sends a `pflist.html` with no files to prefetch and a delay, after which the `getMore()` function gets called. The information TURN breaks the (possibly) long list of prefetch suggestions into a "chain" of short lists.

```
<HTML> <HEAD> <! -- existing header goes here -- >
<SCRIPT LANGUAGE="JavaScript">
function pageOnLoad() {
 myiframe.getPFlist(document.referrer);
} </SCRIPT> </HEAD> <BODY>

<! -- existing body goes here -- >
if(null == window.onload) {
 window.onload = pageOnLoad();}
else {
 var origfn = window.onload;
 window.onload = function(){origfn();pageOnLoad();};}

<IFRAME SRC="pfalways.html" name="myiframe"
        width=0 height=0 frameborder=0>
</IFRAME> </BODY> </HTML>
```

Figure 9. Augmentation of HTML pages

```
<HTML> <HEAD> <SCRIPT LANGUAGE="JavaScript">
function getPFList(var prevref) {
 document.location="HINT-SERVER/pflist.html+PCOOKIE="
        + document.referrer + "+" + prevref + TURN=1;
 document.close();
} </SCRIPT> </HEAD> </HTML>
```

Figure 10. pfalways.html

On demand fetch of a prefetched document The client browser fetches it from the cache as if it is a cache hit.

6.2 Two-connection Prefetching Mechanism

The two-connection prototype employs the same basic mechanism for prefetching as the one-connection prototype. However, since browsers identify cached documents using both the server name and document name, documents fetched from prefetch server are not directly usable to serve demand requests. In order to fix this problem, we modify step 6 such that before calling `getMore()`,

6.a The `myOnLoad()` function (Figure 11) requests a *wrapper* (redirection object) from the demand server for the document that was prefetched.

```
<HTML> <HEAD> <SCRIPT LANGUAGE="JavaScript">

function myOnLoad() { //exeutes after body loads
 preload("DEMAND-SERVER/c.html"); //For two-conn only
 getMore() ;
}
function getMore() {
 document.location="HINT-SERVER/pflist.html +
                   PCOOKIE=" + document.referrer +
                   "+" + prevref + "+" + "TURN=2";
 document.close();
}

var myfiles=new Array()
function preload(){
 for (i=0;i<preload.arguments.length;i++){
  myfiles[i]=new Image() ;
  myfiles[i].src=preload.arguments[i]  ;
 }
} </SCRIPT> </HEAD>

<BODY onload="myOnLoad()">
<SCRIPT LANGUAGE="JavaScript">
preload("PREFETCH-SERVER/a.jpg",
        "PREFETCH-SERVER/b.jpg",
        "PREFETCH-SERVER/c.html");
</SCRIPT> </BODY> </HTML>
```

Figure 11. An example pflist.html returned by the hint server

```
<HTML> <SCRIPT LANGUAGE="JavaScript">
if (document.referrer.indexOf ("pflist") < 0)
          document.location="PREFETCH-SERVER/c.html";
document.close();
</SCRIPT> </HTML>
```

Figure 12. Wrapper for c.html, stored in cache as DEMAND-SERVER/c.html

6.b The frontend intercepts the request (based on the referer field) and responds with the wrapper (Figure 12) that loads the prefetched document in response to a client's demand request.

The prefetch server serves a modified copy of the content on the demand server. Note that the relative links in a webpage on the demand server point to pages on demand server. Hence, all relative links in the prefetch server's content are changed to absolute links, such that when client clicks on a link in the prefethed web page, the request is sent to the demand server. Also, all absolute links to inline objects in the page are changed to be absolute links to the prefetch server, so that prefetched inline objects are used. Since prefetch and demand servers are considered as different domains by the client browser, JavaScript security models [40] prevent scripts in prefetched documents to access private information of the demand documents and vice versa. However, to fix this problem, JavaScript allows us to explicitly set the `document.domain` property of each HTML document to a common suffix of prefetch and demand servers. For example, for servers `demand.cs.utexas.edu` and `prefetch.cs.utexas.edu`, all the HTML documents can set their `document.domain` property to `cs.utexas.edu`.

On demand fetch of a prefetched document: (i) a hit results for the wrapper in the cache, (ii) at the loading time, the wrapper replaces itself with the prefetched document from the cache, (iii) inline objects in the prefetched document point to objects from the prefetch server and hence are found in the cache as well, and (iv) links in the prefetched document point to the demand server.

This mechanism has two limitations. First, prefetched objects might get evicted from the cache before their wrappers. In such a case, when the wrapper loads for a demand request, a new request will be sent to the prefetch server. Since sending a request to the prefetch server in response to a demand request could cause undesirable delay, we reduce such occurrences by setting the expiration time of the wrapper to a value smaller than the expiration of the prefetched object itself. Second, but not a significant limitation is that some objects may be fetched twice, once as demand and once as prefetch objects as the browser cache considers them as different objects.

6.3 Prediction

For our experiments, we use prediction by partial matching [11] (PPM-n/w) to generate hint lists for prefetching. The algorithm uses a client's n most recent requests to the server for non-image data to predict URLs that will appear during a subsequent window that ends after the w'th non-image request to the server. Our prototype uses n=2 and w=10.

In general, the hint server can be made to use any prediction algorithm. It can be made to use standard algorithms proposed in the literature [17, 18, 24, 42] or others that utilize more service specific information such as a news site that prefetches stories relating to topics that interest a given user.

6.4 Alternatives

We explored other alternatives for prefetching in the two-connection architecture. We could have used a Java Applet instead of the JavaScript in Figure 9. One could also use a zero-pixel frame that loads the prefetched objects instead of JavaScript. The refresh header in HTTP/1.1 could be exploited to iteratively prefetch a list of objects by setting the refresh time to a small value.

As an alternative to using wrappers, we also considered maintaining state explicitly at the client to store information about whether a document has already been prefetched. Content could be augmented with a script to execute on a hyperlink's `onClick` event that checks this state information before requesting a document from the demand server or prefetch server. Similar augmentation could be done for inline objects. Tricks to maintain state on the client can be found in [45].

7 Prototype and Evaluation

Our prototype uses the two connection architecture whose prefetching mechanism is shown in Figure 8(b). We use Apache 2.0.39 as the server, hosted on a 450MHz Pentium II, serving demand requests on one port and prefetch requests on the other. As an optimization, we implemented the frontend as a module within the Apache server rather than as a separate process. The hint server is implemented in Java and runs on a separate machine with 932 MHz Pentium III proessor, and connects to the server over a 100 Mbps LAN. The hint server uses prediction lists generated offline using the PPM algorithm [42] over a complete 24 hour IBM server trace. The monitor runs as a separate thread of the hint server on same machine. The content munger is also written in Java and modifies the content offline (as shown in Figure 9). We have successfully tested our prefetching system with popular web browsers inluding Netscape, Internet Explorer, and Mozilla. [1]

7.1 End to End Performance

In this section, we evaluate NPS under various setups and evaluate the importance of each component in our system. In all setups, we consider three cases: (1) No-Prefetching, (2) No-Avoidance scheme with fixed *pfrate*, and (3) NPS (with Monitor and TCP-Nice). In these experiments, the client connects to the server over a wide area network through a commercial cable modem link. On an unloaded network, the round trip time from the client to the server is about 10 ms and the bandwidth is about 1 Mbps.

We use httperf to replay a subset of the IBM server trace. The trace is one hour long and consists of demand accesses made by 42 clients. This workload contains a total of 14044 file accesses of which 7069 are unique; the demand network bandwidth is about 92 Kbps. We modify httperf to simulate the execution of JavaScript as shown in Figures 9, 10 and 11. Also, we modify httperf to implement a large cache per client that never evicts a file that is fetched or prefetched during a run of an experiment. In No-Avoidance case, we set the pfrate to 70, i.e. it gets a list of 70 files to prefetch, fetches them and stops. This pfrate is such that neither the server nor the network becomes a bottleneck even for the No-Avoidance case. For NPS, we assume that each document will consist of ten files (a document is a HTML page along with the embedded objects). Thus the hint server gives out hint lists of size 10 to the requesting clients. Note that many of the files given as hints could be cache hits at the client.

Unloaded resources In this experiment, we use the setup explained above. Figure 13(a) shows that when the resources are abundant, both No-Avoidance and NPS cases significantly reduce the average response times by prefetching. The graph also shows the bandwidth achieved by No-Avoidance and Nice.

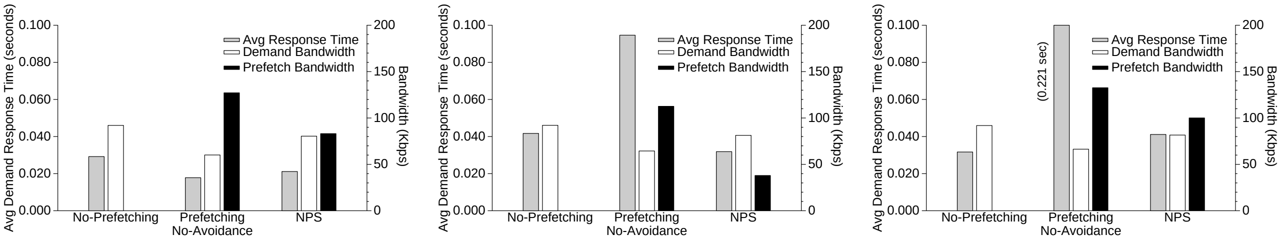
Loaded server This experiment demonstrates the effectiveness of the monitor as an important component of NPS. To create a loaded server condition, we use a client machine connected on a LAN to the server running httperf that replays a heavier subset of the IBM trace and also prefetches like the WAN client. Figure 13(b) plots the average demand response times and the bandwidth used in the three cases. As expected, even though the server is loaded, the clients prefetch aggressively in the No-Avoidance case, thus causing the demand response times to increase by more than a factor of 2 rather than decrease. NPS, being controlled by the monitor, prefetches less data and hence avoids any damage to the demand response times. NPS in fact benefits from prefetching, as shown by the decrease in the average demand response time.

Loaded network This experiment demonstrates the effectiveness of TCP-Nice as a building block of NPS. In order to create a heavily loaded network with little spare capacity, we set up another client machine running httperf that shares the cable modem connection with the original client machine, replays the same trace, and also prefetches like the original client. Figure 13(c) plots the average demand response times, demand bandwidth, and prefetch bandwidth in all three cases. The results show that when the network is loaded, No-Avoidance causes significant interference to demand requests, thereby increasing the average demand response times by a factor of 7. Although NPS doesn't show any improvements, it contains the increase in demand response times to less than 30%, which shows the effectiveness of TCP-Nice in avoiding network interference. The damage is because TCP-Nice is primarily designed for long flows.

8 Related Work

Several studies have published promising results that suggest that prefetching (or pushing)

[1] Source code for NPS prototype can be downloaded from *http://www.cs.utexas.edu/users/rkoku/RESEARCH/NPS/*



(a) Unloaded Resources (b) Loaded Server (c) Loaded Network

Figure 13. Effect of prefetching on demand response times.

content could significantly improve web cache hit rates by reducing compulsory and consistency misses [12, 17, 24, 25, 32, 33, 42, 52]. However, existing systems either suffer from a lack of deployability or use threshold-based magic numbers to address the problem of interference. Several existing commercial client-side prefetching agents that require new code to be deployed to clients are available [39, 27, 53]. At least one system makes use of Java applets to avoid modifying browsers [20]. It is not clear however, what, if any, techniques are used by these systems to avoid self- and cross-interference.

Duchamp [17] proposes a fixed bandwidth limit for prefetching data. Markatos [36] adopts a popularity-based approach where servers forward the N most popular documents to clients. Many of these studies [17, 29, 52] propose prefetching an object if the probability of its access before it gets modified is higher than a threshold. The primary performance metric in these studies is increase in hit rate. However, the right measures of performance are end-to-end latency when many clients are actively prefetching, and interference to other applications.

Davison et. al [16] propose using a connectionless transport protocol and using low priority datagrams (the infrastructure for which is assumed) to reduce network interference. Servers speculatively push documents chunked into datagrams of equal size and (modified) clients use range requests as defined in HTTP/1.1 for missing portions of the document. Servers maintain state information for prefetching clients and use coarse-grained estimates of per-client bandwidth to limit the rate at which data is pushed to the client. Their simulation experiments do not explicitly quantify interference and use lightly loaded servers in which only a small fraction of clients are prefetching. Crovella et. al [12] show that a window-based rate controlling strategy for sending prefetched data leads to less bursty traffic and smaller queue lengths.

In the context of hardware prefetching, Lin et. al [34] propose issuing prefetch requests only when bus channels are idle and giving them low replacement priorities so as to not degrade the performance of regular memory accesses and avoid cache pollution. Several algorithms for balancing prefetch and demand use of memory and storage system have been proposed [6, 8, 31, 44]. Unfortunately, applying any of these schemes in the context of Web prefetching would require modification of existing clients.

9 Conclusion

We present a prefetching mechanism that (1) systematically avoids interference and (2) is deployable without any modifications to the HTTP/1.1 protocol, existing clients, existing servers, or existing networks.

References

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

[2] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Measurement and Modeling of Computer Systems*, pages 90–101, 2000.

[3] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI*, 1999.

[4] C. Bouras and A. Konidaris. Web Components: A Concept for Improving Personalization and Reducing User Perceived Latency on the World Wide Web. In *The 2nd International Conference on Internet Computing*, 2001.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE Infocom*, 1999.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *SIGMETRICS*, 1995.

[7] B. Chandra. Web Workloads Influencing Disconnected Service Access. Master's thesis, University of Texas at Austin, May 2001.

[8] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Razzaq, and A. Sewani. Resource Management for Scalable Disconnected Access to Web Services. In *WWW10*, May 2001.

[9] X. Chen and X. Zhang. Coordinated Data Prefetching by Utilizing Reference Information at Both Proxy and Web Servers. In *PAWS 2001*.

[10] J. Cho and H. Garcia-Molina. Synchronizing a Database to Improve Freshness. In *2000 ACM International Conference on Management of Data*, May 2000.

[11] J. Cleary and I. Witten. "Data compression using adaptive coding and partial string matching". *IEEE Trans. Commun.*, 1984.

[12] M. Crovella and P. Barford. The Network Effects of Prefetching. In *Proceedings of IEEE Infocom*, 1998.

[13] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *SIGMETRICS*, May 1996.

[14] M. Dahlin. Technology trends data. http://www.cs.utexas.edu/users/dahlin/techTrends /data/diskPrices/data, January 2002.

[15] B. Davison. Assertion: Prefetching with GET is Not Good. Web Caching and Content Distribution Workshop, June 2001.

[16] B. D. Davison and V. Liberatore. Pushing Politely: Improving Web Responsiveness One Packet at a Time (Extended Abstract). *Performance Evaluation Review*, 28(2):43–49, September 2000.

[17] D. Duchamp. Prefetching Hyperlinks. In *Second USITS*, October 1999.

[18] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web Prefetching between Low-Bandwidth Clients and Proxies: Potential and Performance, 1999.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. HTTP/1.1. Technical Report RFC-2616, IETF, June 1999.

[20] Fireclick. Netflame. http://www.fireclick.com.

[21] J. Gray and P. Shenoy. Rules of Thumb in Data Engineering. In *Proceedings of the 16th International Conference on Data Engineering*, pages 3–12, 2000.

[22] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *USITS97*, Dec 1997.

[23] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable Distributed Data Structures for Internet Service Construction. In *OSDI*, 2002.

[24] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[25] J. S. Gwertzman and M. Seltzer. The Case for Geographical Push-Caching. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 1995.

[26] IBM. Websphere. http://www.ibm.com/websphere.

[27] IMSI Net Accelerator. http://nct.digitalriver.com/fulfill/0002.3.

[28] Intel. N-tier Architecture improves scalability and ease of integration. http://www.intel.com/eBusiness/pdf /busstrat/industry/wp012302.pdf.

[29] Q. Jacobson and P. Cao. Potential and Limits of Web Prefetching Between Low-Bandwidth Clients and Proxies. In *Third International WWW Caching Workshop*, 1998.

[30] V. Jacobson. "Congestion avoidance and control". In *Proceedings of the ACM SIGCOMM '88 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1988.

[31] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. R. Karlin, and K. Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *OSDI*, pages 19–34, 1996.

[32] M. Korupolu and M. Dahlin. Coordinated Placement and Replacement for Large-Scale Distributed Caches. In *Workshop On Internet Applications*, June 1999.

[33] T. M. Kroeger, D. E. Long, and J. C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *USITS*, 1997.

[34] W.-F. Lin, S. Reinhardt, and D. Burger. Designing a Modern Memory Hierarchy with Hardware Prefetching. In *IEEE Transactions on Computers special issue on computer systems*, volume Vol.50 NO.11, November 2001.

[35] C. Lumb, J. Schindler, G. R. Ganger, E. Riedel, and D. F. Nagle. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives. In *OSDI 2000*.

[36] E. Markatos and C. Chronaki. A Top-10 Approach to Prefetching on the Web. In *INET 1998*.

[37] J. C. Mogul. Network Behavior of a Busy Web Server and its Clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, California, 1995.

[38] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[39] Naviscope. http://www.naviscope.com.

[40] Netscape Communications Corporation. JavaScript Security. http://developer.netscape.com/docs/manuals/ communicator/jsguide4/sec.htm.

[41] A. Odlyzko. Internet Growth: Myth and Reality, Use and Abuse. *Journal of Computer Resource Management*, pages 23–27, 2001.

[42] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World-Wide Web Latency. In *Proceedings of the SIGCOMM*, 1996.

[43] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *USENIX Annual Technical Conference*, 1999.

[44] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *SOSP*, 1995.

[45] R. Rajamony and M. Elnozahy. Measuring Client-Perceived Resonse Times on the WWW. In *USITS*, 2001.

[46] Resonate Inc. http://www.resonate.com.

[47] RFC 2475. An Arhitecture for Differentiated services. Technical Report RFC-2475, IETF, June 1999.

[48] N. T. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, and B. N. Bershad. Receiver Based Management of Low Bandwidth Access Links. In *INFOCOM*, pages 245–254, 2000.

[49] C. S. Systems. http://www.cheetah.com.

[50] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth Constrained Placement in a WAN. In *Symposium on the Principles of Distributed Computing*, Aug 2001.

[51] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A Mechanism for Background Transfers. In *OSDI*, December 2002.

[52] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. The Potential Costs and Benefits of Long Term Prefetching for Content Distribution. In *Sixth Web Caching and Content Distribution Workshop*, June 2001.

[53] Wcol. http://shika.aist-nara.ac.jp/products/wcol/wcol.html.

[54] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.

[55] Zeus Technology. http://www.zeus.com.

[56] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, AT&T Center for Internet Research at ICSI, http://www.aciri.org/, May 2000.