Web Prefetching Using Partial Match Prediction

by

Themistoklis Palpanas

Graduate Department of Computer Science
University of Toronto
Technical Report CSRG-376

Web Prefetching Using Partial Match Prediction

Themistoklis Palpanas

Technical Report CSRG-376

Department of Computer Science

University of Toronto 1998

Abstract

Web traffic is now one of the major components of Internet traffic, which corresponds to the explosive growth that this medium is experiencing. One of the main directions of research in this area is to reduce the time latencies users experience when navigating through Web sites. Caching is already being used in that direction, yet, the characteristics of the Web cause caching in this medium to have poor performance. Therefore, *prefetching* is now being studied in the Web context. When prefetching is employed, Web pages that the user is likely to access in the near future are transfered without being requested. Thus, when prefetching is appropriately employed, the performance of the cache can be significantly improved, which will in turn have a direct impact on the delays perceived by the users.

This study investigates the use of partial match prediction, a technique taken from the data compression literature, for prefetching in the Web. The main concern when employing prefetching is to predict as many future requests as possible, while limiting the false predictions to a minimum. The simulation results suggest that a high fraction of the predictions are accurate, so that additional network traffic is kept low. Furthermore, the simulations show that prefetching can substantially increase cache hit rates.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web [BLCL$^+$94] usage has proliferated during the last years. It is being used for entertainment, commercial, or educational purposes by millions of people around the world, and for some of them it has become an indispensable tool for their work. It gives access to a tremendous variety of information, which can be in text, image, audio, or video format, and can be physically located anywhere in the world.

Nowadays, apart from researchers and educational institutions, commercial organizations, public and private industries use the Web to disseminate information, advertise products, provide support, and deliver services. There already exist companies [Ama98] that rely solely on the Web for interacting with their customer base, and it seems that more and more companies will adopt the same commerce paradigm in the near future. Web traffic is now one of the major components of Internet traffic [Pax94], and accounts for much of the explosive growth that Web is experiencing.

Although it is a relatively new domain, it has already attracted the attention of the research community. One of the main directions of research is to reduce the time latencies users experience when navigating through Web sites. There are several factors that may contribute to these latencies, such as:

- server configuration
- server load
- client configuration

- document to be transferred
- network characteristics

The work in the direction of reducing page retrieval delays originated from relevant research in operating systems, where the aim is to reduce file system latencies [BAD$^+$92, KTP$^+$96]. Caching is a technique that is already being used in the Web domain. The cache, which is realized in local disk storage, is used for storing Web pages that were recently accessed. Caches reduce latencies, because they allow fast retrievals of potentially frequently accessed documents.

1.1 The HTTP Protocol

This section gives a brief overview of the transfer protocol used by the Web. The issues discussed here provide clarifications and serve as a motivation for the rest of this work.

1.1.1 HTTP/1.0

The original version of the protocol, HTTP/1.0 [BLFF95], is still the most widely employed in the Web. In HTTP, clients are sending requests concerning a specific resource to servers, and servers are responding to these requests.

HTTP is layered over the Transmission Control Protocol (TCP) [Pos81], thus, inheriting some of the latter's characteristics. Figure 1.1 depicts the flow of packets over time, that are exchanged between a client and a server, for a request to be served. In order for a client request to be served, a number of steps should be performed (assume a request for an HTML document with embedded images):

1. the client opens a new TCP connection with the server, and then sends the request for the HTML document;
2. the server sends the HTML file (i.e., the text without the images), and closes the connection;
3. the client parses the HTML file;

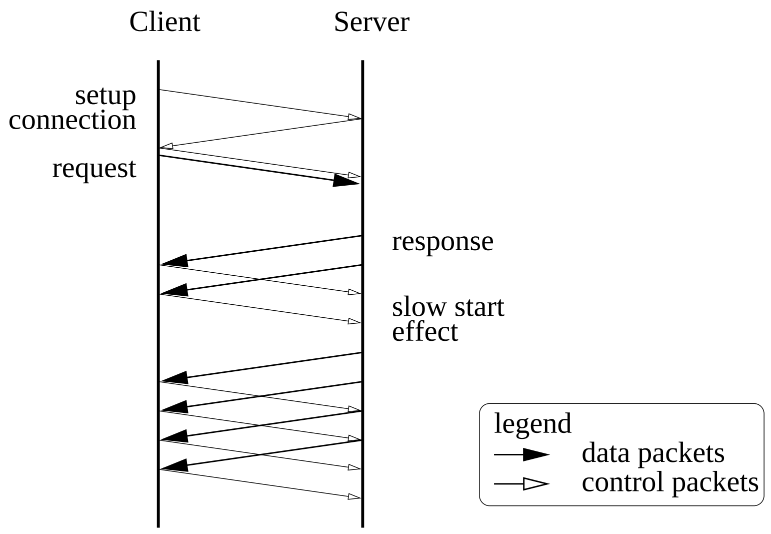


Figure 1.1: Establishing an HTTP connection between a client and a server.

4. the client opens a new TCP connection, and then requests the first of the embedded images;
5. the server sends the requested image, and closes the connection; and
6. the client continues until the whole document is received.

1.1.2 HTTP/1.1

It is obvious that there are inefficiencies in the way that HTTP/1.0 was defined. Web traffic is usually characterized by bursts of numerous, but small requests [AW97, WAS$^+$96]. The frequent need for opening new connections and tearing them down, which imposes a significant overhead, and the slow-start algorithm that TCP implements for congestion avoidance [Jac88], are not well-suited for the short-lived connections required for the Web.

The HTTP/1.1 protocol [FGM$^+$97] tries to solve the aforementioned problems. It uses long-lived TCP connections (*persistent connections*) to serve multiple HTTP requests. This allows a client to make several requests in a short amount of time, using the same connection. Therefore, the time latencies are reduced.

1.2 Motivation

The main concern in Web performance is to reduce the delays associated with page retrievals. The introduction of the HTTP/1.1 protocol is a major step in this direction. Simulations and working prototypes implementing the new protocol indicate the benefits derived from its usage [PM94, NGBS$^+$97].

However, time latencies remain significant despite these improvements. Ideally, we would like all the requested documents to be already in the cache at the time of their request. In this case, the delays would be minimal and rather negligible. Unfortunately, recent studies [KLM97, DFKM97] indicate that the benefits from caching are quite limited. The vast number of documents available in the Web, the quick rate of their change, and the diverse needs across users and over time, are the main factors that cause Web caching to perform poorly.

Therefore, another method that was first applied in operating systems, *prefetching*, is now being studied in the Web context. This method does not reduce the page retrieval latency. It rather tries to hide it from the user's point of view. It is often the case that the user spends a significant amount of time processing a page, while the I/O system is idle. The computer could exploit this time to fetch other pages that the user might request, in the background.

When prefetching is employed, Web pages that the user is likely to access in the near future are transferred to her cache without being requested. If the user does request one of the prefetched pages, it will already be in the local cache, thus, reducing the latency to minimum. Otherwise, the transmission of these pages was superfluous (since they were transferred, but never requested), and the bandwidth used was wasted.

Prefetching is complementary to the caching mechanism. When making good predictions about future requests, the cache receives pages that are highly likely to be accessed soon, and can serve the clients requesting them promptly. Thus, the performance of the cache can be significantly improved, which will in turn have a direct effect in reducing the delays perceived by the users.

1.3 Limitations

Prefetching has the potential to alleviate the problem of latencies in the Web, but it certainly is not a panacea. Actually, it is quite clear that this technique has certain limitations which derive from the special characteristics of the Web. Prefetching can only be advantageous when clients navigate in a site with a purpose, i.e., not aimlessly hopping from one page to another, and when the overall access patterns in servers do not vary extremely.

Yet, studies [AW97, DFKM97] indicate that Web documents tend to change frequently. Still, many of the documents residing in the same site are accessed only once. Moreover, users do not spend much time in each site, and may actually leave immediately after the first access.

Obviously, the aforementioned cases restrict the benefits expected from prefetching. However, this technique is still valuable, and can deliver significant performance improvements when appropriately employed, as this thesis indicates.

1.4 Overview

This work investigates the applicability of multiple high-order Markov models for doing prefetching in the Web. Specifically, an algorithm that is also employed in the compression context, *Prediction by Partial Match*, is used to predict the users' future requests to a particular Web site. A trace-driven simulation emulates the network traffic with the additional (i.e., the prefetched pages) workload, the behaviour of the clients' browsers and their caches under the new scheme, and explores the characteristics of the prediction algorithm. This study evaluates the costs and benefits of employing this technique, when different parameters of the model vary. The results show that the algorithm can be efficiently tailored to suit the new environment, and deliver considerable benefits to the users.

The remainder of this document is organized as follows. Chapter 2 presents the related work. The theoretical model, and the implications of employing it to the Web environment are discussed in Chapter 3. Chapter 4 describes the simulation framework, the assumptions made, and the techniques used. The results of the simulations are presented and discussed in Chapter 5. The last chapter summarizes the work, and suggests directions for future research.

Chapter 2

Literature Review

One of the main directions of research in the Web community is to reduce the time latencies users experience when navigating through sites. The work in this direction originated from relevant research in operating systems, where the aim is to reduce file system latencies. Previous work in both communities indicates the need to assist the caching technique with other methods in order to achieve better performance, with the emphasis given to prefetching.

2.1 Operating Systems Context

2.1.1 Caching

Caching has been used successfully for many years, yielding significant improvements in I/O latency [BAD$^+$92]. It tries to reduce time latencies induced by the file system by having the requested file pages available in memory.

Yet, the benefits from exploiting a cache are limited. Ousterhout et al. present [OCH$^+$85] a trace-driven simulation which indicates that the relative benefit of caching decreases as cache size increases. The experiments show that beyond a certain point response times cannot be substantially reduced any further, no matter how large the cache is.

Moreover, Ousterhout [Ous90] shows that applications are still I/O bound, despite the presence of the cache. It is precisely the requests that cannot be satisfied by the cache

(cache misses) that are the bottleneck.

2.1.2 Prefetching

In order to improve performance, the ordinary caching algorithms should be supplemented with other methods, thereby enabling a further increase in cache hits, and thus, a decrease in average waiting time for the user.

Prefetching is a concept that helps in that direction. It enables the cache to satisfy more requests by predicting sufficiently ahead of time what these requests will be.

A naive method for doing prefetching is to have the application inform the operating system of its future requirements [PGS93]. When this method works, it is very efficient since the application knows exactly which files it will need and at which point in the future. However, the problems this approach has to overcome are rather significant. First of all, applications must be modified to use this technique. So old applications will have to be rewritten, or otherwise they will not benefit at all. Then, the mere task of programming such an application will become extremely cumbersome. The programmer will have to learn and use a set of complex directives to manipulate low level functions with a strong dependence on the physical resources. Furthermore, just the knowledge of future requests is not enough. The prefetching should occur a significant time before a page is actually needed (though, not too much in advance) to ensure that the file will be available when requested. This poses an additional difficulty in the programming task.

One way to overcome the aforementioned problems is by instrumenting the compiler to provide all the necessary prefetching information to the operating system [MDK96]. The compiler can add instructions in the code for prefetching pages that will soon be accessed, as well as for evicting pages that will no longer be useful, thus, enabling other applications to use the freed space. This approach was tested against the NAS parallel benchmark suite [BBLS91], and resulted in execution speed-ups of twofold and in one case threefold. However, the experiments were performed on a system with multiple disks, which significantly increased the available I/O bandwidth. The effects of varying file system bandwidth were not tested. In addition, the scope of that work is currently limited to numeric applications, so it is expected to be beneficial only in highly restricted settings.

A recent study [KTP+96] has tried to explore the effectiveness of combined caching and prefetching techniques using a trace-driven simulation. Assuming a single-process system with multiple disks and full advance knowledge of future requests, combined caching and prefetching algorithms enable some of the existing strategies to reduce I/O latency to zero. Then the application becomes CPU bound. When restricted to one disk though, the stall time (application waiting for file system operations) remains significant.

An additional problem that cannot be tackled by the prefetching methods presented above arises when related file accesses span multiple executables. Typically, each application can only be aware of its own file access patterns. Yet, it is frequently the case that a series of different applications are executed repeatedly (e.g. commands of a shell script or a makefile). In order to capture such relationships, long-term history information on accesses across applications is exploited. These techniques assume that future request patterns in an operating system will be similar to past ones. As a consequence, in contrast to the previous methods, this one makes predictions without using any application specific knowledge at all. Therefore any application can benefit without having to be rewritten, but predictions are in general fewer or less accurate.

Griffioen and Appleton [GA96] propose a predictive cache which prefetches file pages based on a *probability graph* of past accesses. The nodes in the graph represent accessed files, while a directed weighted arc from node A to node B expresses a metric for the probability of requesting file B within w requests after file A. A trace-driven simulation revealed improvements in the time an application has to wait for read operations to complete (*read complete time*) ranging from 22% to 51% compared with the non-prefetching case. Similar results yielded a prototype implementation of the system, tested against a synthetic workload. However, the overall improvement in execution time ranged from 1% to 8%, indicating that only a small fraction of the prefetching benefits is reflected in the total run time. This is due in part to the overhead prefetching imposes on the system. The dominant factor, though, is the ratio of the read complete time to the total execution time.

Curewitz et al. [CKV93] claim that data compression techniques can be successfully used for prefetching. An implementation of this idea, based on *Prediction by Partial Match (PPM)*, is presented by Kroeger and Long [KL96]. The advantage of this method is that the prediction of future requests depends on the k previous accesses, thus resulting in better predictions. The major drawback is the algorithm's memory requirements. In the trace-driven simulation, cache-hits increase by an average of 25% over a non-prefetching cache. The simulation shows that on average a 4MB predictive cache has a higher hit rate than a

90MB simple *Least Recently Used (LRU)* cache. Results on time latencies are not reported in this paper.

Lei and Duchamp [LD97] employ *access trees* to make predictions. Access trees record all files accessed during one execution of each program. When an application is re-executed, the current activity is compared against the saved access trees. If a similarity is detected, the files in the access tree are prefetched. The algorithm does not force any prefetches unless a sufficiently similar pattern exists in past access trees. The trace-driven simulation shows a substantial reduction in applications latency, despite the significant CPU overhead. However, the trace used was not based on a real workload.

The problem of predictive caching emerges also in *near-line tertiary storage libraries* [KW97a], although in this case time scales are different. The decision to be made is which large multimedia files should be prefetched from slow tertiary storage to fast disks. A continuous-time Markov-chain model is used to predict what the future accesses will be within time t. A trace-driven simulation of synthetic workloads reveals time latency reductions of up to a factor of two.

2.2 World Wide Web Context

Many of the initial ideas employed on the Web were borrowed from similar work in operating systems. Indeed, it is only recently that the peculiarities of the Web domain started to become apparent [WAS$^+$96, Mar96, DFKM97].

The characteristics of Web resources are explored by Douglis et al. [DFKM97]. The trace used in the study showed that 69% of the accesses and 60% of the bytes transfered involved images. The same figures for HTML documents are 20% and 21% respectively. The analysis of the trace indicates that 22% of the resources were accessed more than once, but they accounted for 50% of the requests. From this 50%, 13% were to resources that had been modified since the previous request. Unlike another study [Bes96], which was performed in the educational environment, this trace suggests that the smaller and most frequently referenced resources tend to change more often. It was also observed that pages in the educational domain are modified noticeably less often than pages in other domains. The above results manifest the diversity and changing nature of the Web.

2.2.1 Caching

Williams et al. [WAS+96] investigate the operation of proxy caches, i.e., caches "near" the clients. Their study shows that *SIZE*, a policy where the largest document is removed first, always performs better than *LRU* (and at within 10% of the optimal) in terms of document cache hits (as opposed to byte cache hits). When a single large document is evicted from the cache, space is freed for many small ones. Moreover, the majority of the requests are for small documents. Nevertheless, *SIZE* proves to be the worst policy when byte cache hits are measured since large documents tend not to be cached.

In accordance with the aforementioned results, Lorenzetti et al. [LRV97] propose a new replacement policy, *Lowest Relative Value (LRV)*. It takes into account the size of the document, the number of past references, and the time of the last reference. Such a cost model enables the algorithm to maximize both the document and byte cache hits. The implementation of the algorithm, however, requires some parameters that may only be determined through experiments, and could vary with time. Trace-driven simulations show that *LRV* performs as well as, or better than the best policy considered so far. The simulations do not account for time latencies.

A cache which resides on the server, *Main Memory Web Cache*, is proposed by Markatos [Mar96]. This kind of cache can substantially alleviate the server load by having the majority of the requests serviced from the memory instead of the file system. A trace-driven simulation consisting of five diverse traces shows that even a small amount of main memory (512KB) is enough for the cache to experience a document hit rate of more than 60%. The algorithm used implements an *LRU* removal policy, but has also a limit for the maximum allowed document size that can be cached. The limit is dynamically adapted so as to maximize the document cache hit rate. These traces demonstrate that a significant percentage of the requests involve a small number of documents, and even a small cache can significantly reduce the server load. However, no figures are provided for the modification frequency of the pages. Note that a high frequency would result in poorer performance.

Additional, more subtle parameters of the proxy cache operation are addressed in other studies. Krishnamurthy and Wills [KW97b] explore the cache coherency problem, i.e., assuring the validity of cached resources, and present a new technique, the *Piggyback Cache Validation (PCV)*. The cache, instead of sending explicit validation requests to the server to find out the expiration times of the cached documents one at a time, piggybacks a list of documents from that server whenever a communication occurs. A trace-driven simulation for different cache sizes shows that among other policies currently employed, *PCV* results in fewer requests sent to the server, while maintaining a close-to-strong coherent cache. Furthermore, it yields the lowest costs in terms of response time, bandwidth used, and number of validation messages. By extending this technique, the server could piggyback resource invalidations back to the cache, which in turn could free some space sooner.

The work of Shim et al. [SSV98] suggests a unified algorithm for cache replacement and consistency. The algorithm uses a cost function in order to decide which documents to replace in the cache. The factors determining the cost of a document are the size, the retrieval latency, the frequency of accesses, and the frequency of validation checks for that document. Trace-driven simulations [SSV97, SSV98] indicate that the new algorithm achieves a better hit ratio than LRU, while improving the cache consistency level over the current approach (i.e., for documents whose modified status is unknown, validate with the server upon request). However, no results are reported for the byte hit ratio. Furthermore, although the algorithm needs to keep track of several parameters, its complexity is not analyzed.

The benefits from *delta encoding* (transmitting only the difference between two versions of the same document) and *data compression* for the transfer of Web documents are investigated by Mogul et al. [MDFK97]. The trace-driven simulation uses an unlimited-size cache and a filtered trace (only status-200 responses, with no images). It demonstrates that the response body-bytes saved are at least 98.5% for half of the delta-eligible responses, or 30% for all responses. Latency is reduced by 12%. However, the latency benefits tend to decrease as the bandwidth increases. Apart from the computation overhead of this method, there is a space overhead, since the server needs to keep several old versions for each file. The study suggests, though, that these overheads are minimal.

2.2.2 Prefetching

Bestavros proposes two server-initiated protocols [Bes96]. The first one is a hierarchical data dissemination mechanism that reduces network traffic by propagating Web documents from servers to server proxies that are closer to clients. The second protocol, a *speculative service*, sends to the client —along with the requested document— a number of other documents that the server predicts will be requested in the near future. The predictions are produced by the closure of the matrix whose *(i,j)* element represents the probability that document j will be requested within a certain time window after the request of document i. The trace-driven simulation reveals that a slight increase in network traffic yields significant reductions on server load, service time, and client cache miss rate. These results however, were obtained by allowing a document to predict the request of its images, thus distorting the figures.

Previous work on operating systems [GA96] has been the basis of Padmanabhan and Mogul's research [PM96] on prediction of future requests. A trace-driven simulation along with a linear model for the network is used to test the algorithm. The results indicate that the benefits from prefetching are significant, even for slow (i.e., modem) connections. Prefetching can reduce the average latency by up to 30% with a 25% increase in network traffic. In contrast, a doubling of the bandwidth reduces latency only by 20%.

A cooperation of the client and the server is proposed by Wills and Sommers [WS97]. The trace-driven simulation suggests that depending on the client's browsing habits, client information can assist the server's suggestions for prefetching. The server uses the prediction model presented by Bestavros [Bes96].

A recent study attempts to determine bounds in the performance of proxy caching and prefetching [KLM97]. The authors performed a trace-driven simulation with infinite-size cache and complete knowledge of future requests. The external latency between the proxy cache and the server accounts for 77% of the total latency. Caching achieved a reduction of 26%, prefetching 57%, and the combination of both 60%. Although these figures are quite different for the second trace used, they exhibit the same trends. The weak association of proxies and latency reduction is confirmed by a real-case study of cache performance [MRG97]. The simulation also shows that the available bandwidth for prefetching has little effect on latency reduction for which the dominant factor is how far in advance of the actual need the data can be prefetched. The study, however, does not comment on the number of single accesses to servers that inherently cannot be prefetched.

Crovella and Barford [CB98] discuss the effects of prefetching on the network. A trace driven simulation indicates that straightforward approaches to prefetching increase the burstiness of traffic. Instead, the authors propose a transport rate control mechanism, which is intended to reduce the bursting effects by using as little bandwidth as possible while ensuring that the documents are delivered on time. The simulation denotes that rate-controlled prefetching significantly improves network performance compared not only with the straightforward approach, but also with the non-prefetching case.

2.2.3 Web Mining

Web mining is a relatively new area of research, which involves identifying common or unexpected *access patterns*. The discovery of common patterns may provide suggestions for better marketing decisions, while the recognition of unexpected patterns may help to improve the structure of the Web site. Prefetching and Web mining are associated in two ways. First, they try to recognize regular user access patterns. Second, they need a way of extracting and modeling the sequences of page references into logical units representing user sessions (i.e., only the pages that a user requested during a single visit in the Web site are grouped together). Note that Web mining is also used for extracting interesting content information from Web pages. However this particular aspect is not the focus of our work.

The problem of mining path traversal patterns in the Web is explored by Chen et al. [CPY96]. The interesting point in the proposed algorithm is that it differentiates between *forward* and *backward* (to objects already visited) traversals. Backward traversals are ignored, since they are considered to merely indicate that the user wants to reach a sibling page. It is only in the forward references that useful patterns are expected to be found.

Mobasher et al. [MJHS96] present a framework for Web Mining, and a system based on that framework, *Webminer*. Unlike the previous approach, this method does not consider all backward references to be useless, nor all forward references as meaningful. Instead, user references are clustered according to some criteria, which are primarily time constraints. Recently, a similar system was proposed by Spiliopoulou and Faulstich [SF98]. This tool focuses on the discovery of navigation patterns, and can be used to assist in the process of site reorganization.

2.2.4 Commercial Predictive Cache Products

The interest in the direction of reducing navigation latencies in the Web is not limited to research studies. Several commercial products, termed as *Web accelerators*, promise to enhance the performance of browsers [Con98, Pea98, Web98a, Rob98, Par98, Web98b].

They reside in the client side, and employ prefetching in order to serve potential future requests from the local cache, and thus, minimize latencies. However, the algorithms used are not sophisticated. They merely exploit the idle time of the client's network connection to prefetch all the links of the current page, or the user's popular pages. These products diminish browsing latencies [Fos98], yet, their simplistic approach results in a considerable increase in network traffic.

Chapter 3

Prediction Model

In order to do successful prefetching, a model that describes the users' request patterns is essential. Based on this model, the system can make predictions about future requests. It is obvious that when the model is more accurate, the predictions are more useful. Therefore, determining a model that can effectively capture the real world is of great importance.

Such models, that may be used to drive a prefetching system, can be found in the compression literature. Previous work [CKV93] indicates that the idea of applying techniques used in compression algorithms for making predictions is fruitful. Intuitively, both compression and prefetching work towards the same goal, which is to predict the next element in a particular sequence.

3.1 From Compression to Prefetching

Compressors are building a model of the data they are operating on in order to be efficient. In general, these models generate a probability distribution for the elements in the data, which is subsequently used to achieve effective coding. According to this scheme, data with high probability to occur are encoded with few bits, and infrequently occurring data with many bits. Therefore, an accurate model significantly improves the performance of the coding.

Likewise, in prefetching, the pages that are most likely to be requested next are prefetched. Thus, if a compressor can effectively compress a data sequence that means the model it is using accurately describes the data, and can be used for making predictions.

The idea of applying prediction techniques used in the compression context for doing prefetching was explored by Curewitz et al. [CKV93]. The results indicate that not only is it feasible, but also that the performance is remarkable. The simulation experiments use traces derived from Computer Aided Design applications, and from a set of benchmarks. When such prefetchers are employed, the cache experiences page fault rates (number of page faults divided by the length of the access sequence) reduced by 15%-60% compared to a pure Least Recently Used (LRU) cache.

3.2 Context Modeling

As advocated earlier, the prediction model that a compressor uses is crucial for its efficacy. Research in the compression community has shown that the algorithms employing *context modeling* (for general purpose compression) tend to achieve superior performance [BCW90].

Context models comprise a family of techniques that use the preceding few characters in order to calculate the probability of the next one. As an example, consider the domain of the English language, and the string "start". The simplest way to predict which character will follow this string is by selecting the one with the highest fixed probability. That is, when considering the probability of each letter irrespective of its position in the string. Note that in this case the previous characters are ignored. Therefore, the letters "i" and "h" are approximately equiprobable of occurring next.

However, the models can become much more elaborate. A more sophisticated way of computing the letter probabilities is to take into account the preceding symbols. Thus, the letter "i" is much more likely to follow the substring "rt" than "h" is.

Evidently, the predictions are more accurate when they are produced with respect to some context. For this example, the contexts in which the letter following "start" occurs are defined by the set $\{\emptyset, ``t", ``rt", ``art", ``tart", ``start"\}$. The length of a context is its *order*. So, if the m preceding symbols are used to determine the probability of the next character this is an *order-m* model. The order-0 model refers to the case where the preceding symbols are ignored.

Intuitively, the higher order the model is, the more accurate the predictions are, since more relevant information is taken into account. The problem that arises when using long contexts is that most of them will never have been seen, i.e., the specific sequences of symbols will be extremely rare. Consequently, the predictor uses the predetermined fixed probabilities. Alternatively, a lower order model can be employed. In both situations the performance is rather poor. In order to avoid the aforementioned problems, *blending* can be used. This is a strategy that allows the predictions of contexts of different lengths to be combined into a single probability.

3.2.1 Prediction by Partial Match

Prediction by Partial Match (PPM) [CW84] is one of the context models that have been proposed in the literature. A PPM compressor uses multiple high-order Markov models to describe the dataset. That means it keeps track of the probability of a symbol occurring, given that a specific sequence of symbols has already been seen. The algorithm uses the high-order contexts to make predictions, when these contexts are available. Otherwise, the predictions are based on the lower orders. These predictions subsequently drive an arithmetic coder which performs the actual compression.

The work presented herein is based on the prediction technique used by the PPM compressor. Detailed description of the prediction mechanism of the PPM algorithm and its implementation in the Web context is given in the following paragraphs.

3.3 Prefetching in the Web

In the Web context the focus is on constructing a model for the users' page access sequences. Thus, the "alphabet" in this case is comprised by individual Web documents, each specified by its name in the Web name space. This model should describe the way users are navigating in the Web, and should capture access patterns that are occurring frequently.

3.3.1 Types of Prefetching

Prefetching can take various forms and can be achieved by instrumenting the client, the server, or both. Depending on which part is more involved in the process, the different flavors of prefetching can be classified in the following categories:

Client-based: The prefetching model is based on the navigation patterns of certain users, since the relevant information is gathered in the client side. Prefetching can either be done on a per-user basis (i.e., a separate model for each user), or for a community of users (i.e., one model based on a proxy server). Client-based prefetching concentrates on the navigation behaviour of *specific* users *across* Web servers.

Server-based: The prefetching process is coupled with the server. It models the access patterns of *all* the users accessing the *specific* server. The advantage of this approach is that the prefetcher benefits by merging the reference sequences (concerning the same pages) of a great number of clients.

Combined client and server: This is a combination of the other two methodologies. The client and the server combine their knowledge in order to make more accurate predictions.

This study elaborates on the server-based approach. It proposes, as a prefetching algorithm, an adaptation of a compression technique to the Web domain and investigates the peculiarities of this transformation.

3.3.2 Building the Model

In this work, the prefetching method is based on the PPM algorithm. The elements of the model are access events to particular Web pages, and the contexts correspond to sequences of such events. Note that all the Web pages reside in the same server (where the prefetching algorithm is running), while requests for those pages may originate from any client in the Internet. Therefore, if A, B, and C are three pages (all of them residing in the same Web server) the context $\{ABC\}$ denotes that these three pages have been accessed in that specific order by at least one of the clients.

An *order-m* prefetcher maintains $m+1$ Markov predictors[1] which correspond to contexts of length 0 to m. One way of describing this model is by using a trie [Knu73]. Tries are special forms of tree data structures, and are typically used to store dictionary of words.

[1]An *order-k Markov predictor* is a scheme which calculates the conditional probability p of accessing page P, given that the previous accesses were to pages $\{P_1, P_2, \ldots, P_k\}$ in that order. More formally, the predictor computes the probability $p(P|P_kP_{k-1}\ldots P_1)$.

Each node in a trie represents the sequence of events that can be found by traversing the trie from the root to that node, or in other words a context of length equal to the depth of the node. Thus, contexts of different lengths can be mingled in a single structure since any context of length j contains all the information required for the corresponding context of length $j-1$.

In order to facilitate updates an additional data structure keeps track of the *current* context of length j, for $0 \leq j \leq m$. A length-j current context embodies the j last events of the access sequence. Then, a new event (i.e., the next page access in the sequence) is added to the model in the following fashion:

1. For each current context of length j check whether any of the child nodes represents the new event.

2. If such a node exists (i.e., the same sequence has been seen before) then set the current context of length $j+1$ to this node, and increment the number of occurrences of this context. Otherwise, create a new child node, and proceed with the same operations. Note that during this step the context of a higher length is involved.

The algorithm for constructing the prediction model is depicted in Figure 3.1. Observe that in order to update the model with a sequence of accesses originating from a different client, the algorithm first resets all the current contexts, and then begins inserting the new sequence.

An example describing the above procedure will give an insight into the way the model is constructed. The example is illustrated in Figure 3.2. Assume that the model of order 2 is initially empty, and that the sequence of accesses $ABACDBCA$ is introduced, where $\{A, B, C, D\}$ is a subset of the Web pages available in a specific server. In the very first step the only available context is the order-0 context, which is represented by the dotted square surrounding the root node Λ (Figure 3.2.a). Thus, the element A, the first element in the sequence, becomes the child of Λ, and at the same time the new order-1 context. The number label associated with the node counts the occurrences of this context. The new order-0 context is (and will always be) the root node. Again, the dotted squares show the current context of each order (Figure 3.2.b). When element B comes, all contexts are updated in turn. Since there is no child labeled B in either of the contexts seen so far, two child nodes are created, one for each of the existing contexts. The model, with the updated

Objective: Build a prediction model based on the access patterns of the users.
Input: The trie structure T, representing the prediction model of order m constructed so far, and a set S of events deriving from the same user.
Output: The updated prediction model.

```
current_context[0]:=root node of T;
for length j=1 to m
   current_context[j]:=NULL;

for every event R in S
   for length j=m down to 0 {
      if current_context[j] has child-node C representing event R {
         node C occurrence_count:=occurrence_count+1;
         current_context[j+1]:=node C;
      }
      else {
         construct child-node C representing event R;
         node C occurrence_count:=1;
         current_context[j+1]:=node C;
      }
      current_context[0]:=root node of T;
   }
```

Figure 3.1: Algorithm for building the prediction model.

current contexts, is depicted in Figure 3.2.c. This process continues in a similar fashion until the entire sequence is exhausted (Figure 3.2.e).

As the example portrays, an *order-m* model is represented by a trie of height $m + 1$. The leaves cannot correspond to a context since there is no other event following them.

Note also that not all branches of the trie need to be of equal height. There may exist subtrees which are shorter than others. However, this would be quite unlikely to occur, especially for relatively short contexts and after a large number of sequences has been seen.

In order to add a sequence of page accesses from a different client, the current contexts of all lengths are reset, i.e., they refer to the root node. Then the model is updated as in the previous case. Figure 3.3 depicts the new state of the model, after the sequence $DCABD$

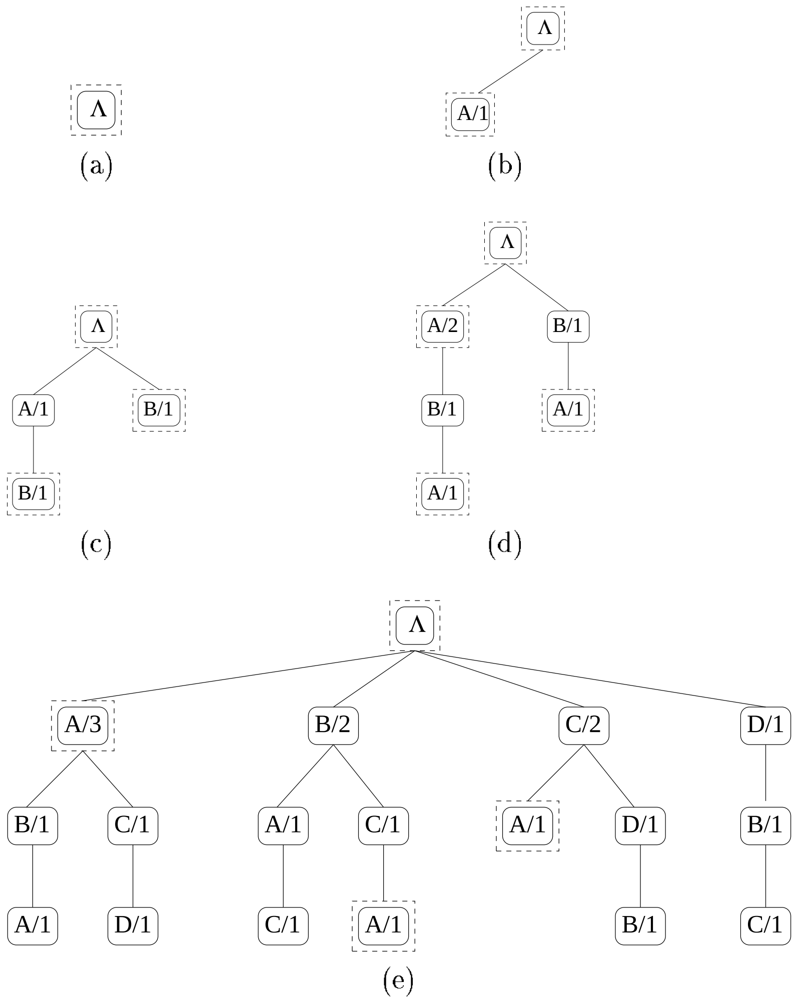


Figure 3.2: Structure of the model when inserting the sequence $ABACDBCA$.

has been appended.

The algorithm described above makes only a single pass over the data, does not require any bulk processing, and there is no need for preprocessing either. Therefore, it can operate in real-time. The model can be updated as information on new user requests comes in, and at the same time provide predictions concerning the future accesses.

3.3.3 Making Predictions

The prefetcher, based on the model describing the access patterns of the users, can make predictions for future requests. The prediction algorithm can take several different forms.

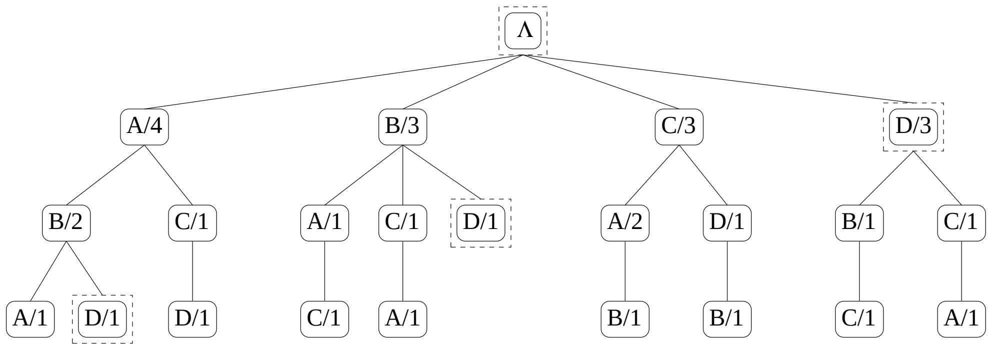


Figure 3.3: The model when the sequence $DCABD$ of a different user has been processed.

Yet, all variations depend on calculating and comparing the probabilities of some events occurring next. These probabilities are derived from the number of times each context has been observed. The following alternatives outline some variations of the algorithm:

- Select events from the highest possible order only.
- Combine predictions from several orders.
- Assign different levels of trust to various context orders.
- Choose up to k events to prefetch, where $1 \leq k \leq k_{max}$.

The fewer limitations the algorithm has, the more accurate predictions it will produce. In the extreme case it would prefetch all the possible events as indicated by the model. However, this is unrealistic since the considerable increase in network traffic would be a restrictive factor. Thus, there is a tradeoff between the expected correct prefetches and the available bandwidth.

When a j-order context is used, the prefetcher is actually making predictions based on the previous j accesses of that client. Therefore, the model is implicitly narrowing down the possible events that will follow. An interesting point is that in the case where predictions from different contexts are merged, subtle variations in the pattern sequences can be captured. In the model of Figure 3.3 for example, assume that a specific user has already seen page D followed by B, and that the next two accesses will be for pages A and C. Both of these pages can be prefetched, since they are predicted by the contexts $\{B\}$ and $\{DB\}$ respectively. Consequently, no matter whether the actual request sequence is

$\{DBAC\}$ or $\{DBCA\}$, it can potentially be predicted by the model.

3.4 Summary

This thesis explores the employment of multiple high-order Markov models for doing prefetching in the Web environment. The algorithm takes as input the sequences of page requests of all the clients accessing the Web server, and constructs a prediction model. Based on this model and the current request of a specific user, the prefetching engine, which resides on the server, decides which pages to prefetch (if any). Then, the server responds to the client by sending the requested page along with the pages instructed by the prefetcher.

A trace-driven simulation is used to study the operation of a Web server under the proposed scheme. The different ways of generating predictions, which were mentioned earlier, are tested independently in order to assess their effect on the behaviour of the algorithm, and in combination so as to estimate the power of the prefetching process.

Chapter 4

Simulation Model

In order to estimate the performance of the algorithm a simulation model is used. In this case, a real-world evaluation of the system would be unrealistic. This task would require not only the instrumentation of the Web server's software, but also changes in the client side, so as to achieve compatibility and cooperation. The simulation is trace-driven, i.e., a real Web server log file is employed to emulate the requests of the clients, and to assess the operational behaviour of the prefetching system.

4.1 Overall Architecture

An overview of the architecture of the simulator is depicted in Figure 4.1. The core of the system is the *dispatcher* which interacts with the rest of the components, namely the *log file*, the *prediction engine*, and the *clients*. Each one of those is further discussed in the following paragraphs.

4.1.1 The Log File

This is the file logging information about user accesses to the Web server of the Computer Science Department of the University of Toronto [Dep98], as well as the actions that the server took in response to the requests. An HTTP request consists of the following elements:

Method: indicates the method to be performed on the resource (e.g., *GET, POST*, etc.)

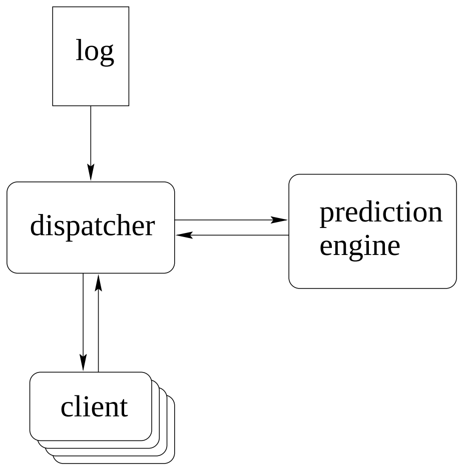


Figure 4.1: The architecture of the simulation model.

Uniform Resource Identifier (URI): uniquely identifies the resource upon which the request should be applied

Request Header Fields: act as request modifiers (e.g., may carry authentication information, or specify which documents to accept)

As a result of a request, the server performs the actions specified by the client. Subsequently, the server sends a response with the following elements:

Status Line: indicates the status of the served request (i.e., whether it succeeded or not, and why)

Response Header Fields: contains further information about the server and the requested resource

Entity: the actual file requested (if any), and some meta-information pertaining to it

The site is supported by the Apache HTTP Server v1.2 [Apa98] which logs the requests using the Common Log Format (CLF). The CLF is a standardized way of arranging the information in the log file. Each line in the file represents a different request, and is composed of several tokens separated by spaces. These tokens are:

host Identifies the client that issued the request to the server. This client may be a machine belonging to a specific user, or a proxy server acting on behalf of a large community of users.

ident The identity information reported by the client.

authuser The userid used when requesting a password protected document.

date The date and time of the request, in granularity of seconds.

request This is the actual request sent by the client. It includes the method (directing the server to perform a specific action) and the URI of the Web page the user is interested in.

status A three digit status code which is returned to the client. It conveys information on the outcome of the requested operation.

bytes The number of bytes send back to the user, not including the message headers.

If a token happens not to have a value, it is replaced by a hyphen ("-").

All the entries in the log are in chronological order, in correspondence to the time each request was received. The aforementioned tokens can be identified in the sample entries taken from an access log file, illustrated in Figure 4.2.

```
pd12.compuserve.com - - [27/Jun/1997:09:05:33 -0400] "GET /~ria/orian.html HTTP/1.0" 200 4195
gatekeeper.sunlife.com - - [27/Jun/1997:09:05:34 -0400] "GET /~gam/g6.html HTTP/1.0" 200 18675
clipper.ssb.com - - [27/Jun/1997:09:05:35 -0400] "GET /~ria/career/team.html HTTP/1.0" 200 29673
dp15.rfci.net - - [27/Jun/1997:09:05:35 -0400] "POST /~riedel/chatstart.cgi HTTP/1.0" 200 182
clipper.ssb.com - - [27/Jun/1997:09:05:35 -0400] "GET /~an/index.html HTTP/1.0" 404 176
pd12.compuserve.com - - [27/Jun/1997:09:05:45 -0400] "GET /~ria/rhockey.jpg HTTP/1.0" 200 1178
pd12.compuserve.com - - [27/Jun/1997:09:05:47 -0400] "GET /~ria/rhockey.gif HTTP/1.0" 200 1195
interpix.com - - [27/Jun/1997:09:05:51 -0400] "GET /~dt/siggraph/ HTTP/1.0" 200 13882
```

Figure 4.2: Sample entries from a Web access log file. The two missing tokens are the *ident* and *authuser*.

4.1.2 The Dispatcher

The dispatcher is the module responsible for the coordination of the other components of the system. It interacts with all three of them, and acts as a mediator in the communication among them. More specifically, the operation of this module can be broken down into the following tasks:

1. Read and parse the next request from the log until the end of the file. This is the actual request issued to the server by some user.

2. Forward the request to the prediction engine and get back the answer. The answer consists of the requested page, and zero or more pages to prefetch.

3. Send the requested page along with the prefetched pages chosen by the prediction engine, to the corresponding client.

4. When the simulation ends, gather and print the overall statistics.

Note that when the simulation begins, a fraction of the log file is used to build the prediction model. Thus, the prefetcher is not making any predictions during this phase, and it subsequently performs a *warm* start.

4.1.3 The Prediction Engine

This component implements the prefetching mechanism. It constructs and updates the prediction model according to the requests issued by the users, and offers predictions independently to each client.

As discussed earlier in this study, the traces pertaining to different clients should not be intermingled when building the model, since this would lead to a distorted view of the actual access patterns. In addition, the model should differentiate between discrete series of accesses even if they belong to the same user. The aforementioned observations are used in order to avoid the correlation of patterns that should not take place. In the former case the assumption is that sequences of requests issued by different users are independent, while in the latter case, requests from the same user which are far apart in time are considered independent as well.

The model tries to capture the above characteristics by introducing the notion of *browsing sessions*. A browsing session S_ϕ is a set of requests, $\{R_1, R_2, \ldots, R_i, \ldots, R_n\}$, where R_i represents an entry from the log file as described above. Each request R_i in the set is constrained to belong to the same client ϕ, i.e., $R_i.host = \phi$ for $1 \leq i \leq n$. In addition, there is a total ordering in the set, $R_i.time \leq R_{i+1}.time$ for $1 \leq i < n$, and a time window is used to group requests in sessions, $R_{i+1}.time - R_i.time \leq t^s_{max}$ for $1 \leq i < n$. Therefore, requests from discrete clients correspond to different browsing sessions, and a session ends when the relevant user has been idle for more than t^s_{max} time units[1]. A typical value for t^s_{max} would be in the order of few minutes.

The prefetcher maintains a list of active sessions as an intermediate processing step between getting the requests from the log file and updating the prediction model. This step is necessary in order to construct a model consistent with the actual user behaviour. There may be several active sessions simultaneously, yet, each one of them must be associated with a different client. When a browsing session ends, the corresponding sequence of accesses is inserted in the model.

The k most recent requests, which are the last k elements of each active browsing session, constitute the length-k context for the specific access sequence. This information is significant since it enables the prefetcher to exploit the high order prediction model and to make predictions targeted individually to each user.

It is important to note here that when building the model not all of the log file entries are taken into account. In particular, it is only the successful requests for *HTML* or *text* files that are processed. Obviously, requests for non-existent pages or requests that result in any other error do not contribute to the model. In addition, pages generated by *CGI programs*[2] cannot be prefetched since they usually involve user selections that normally take place just before the request of the specific page. Finally, all the requests for the inlined images are also ignored. It is evident that when a user issues a request for some document, subsequent requests for all the embedded images will follow. (Exceptions to this statement are the cases where the user has explicitly instructed the browser not to fetch any images, or when the browser operates in text-only mode, e.g., Lynx, both of which are rather rare). Thus, the prefetching system could send some document to a client along with all the inlined references, without having to incorporate any additional knowledge in the prediction model. Furthermore, counting the image requests in the simulation statistics would result in skewed figures since predicting the embedded images of the current request

[1]The s superscript in the time window length stands for the server, where the prediction engine is located. Later on, a similar time window will be introduced for the simulation of the clients.

[2]Common Gateway Interface (CGI) is the protocol that specifies the way an application may communicate with an HTTP server, and consequently interact with clients through Web pages. CGI programs are used to dynamically generate Web pages, based on the user input (e.g., the result of a user-specified database query).

is actually useless and unnecessary.

The prediction engine selects which pages to prefetch based on the occurrence count associated with each node in the model. The occurrence count stores the number of times that the specific context has been seen. Thus, comparing these counts provides a mean of selection among different pages. In addition, the occurrence count of a certain node divided by the count of the father node yields the percentage of times that this particular path was followed among all the possible ones. This fraction also enables the prediction engine to set a *confidence* threshold, below which no pages are selected. This is a useful concept since it does not allow pages belonging to infrequent access patterns to be prefetched.

As discussed in the previous chapter, the strategy of combining predictions from contexts of multiple lengths may prove useful. The outline of such an algorithm is given in Figure 4.3. Each context of the model produces predictions independently, and may be associated with a different level of confidence. Then, all the predictions are merged into a single set, and duplicates are removed.

Observe that the context of length 0 is not used to derive predictions. This context contains no information regarding the past requests. It rather tries to express the unconditional probability of accessing a page. Nevertheless, the extremely large set of pages in a Web site renders this option unprofitable.

4.1.4 The Client

This module simulates the client side of the system. It receives the page actually requested, as well as any additional pages sent by the prefetcher. The client retains a cache, so as to store the prefetched pages. These are pages that the client will potentially access in the future, and if they are not stored until then their transmission was useless. Therefore, the absence of cache is equivalent to the non-prefetching case.

At any time during the simulation there may exist many instances of the client module running, each one associated with a different client. These instances correspond to the *active* clients. A client C_ϕ is termed active if the last request it issued to the Web server is within a window of t^c_{max} time units of the present (simulation) time. Formally, let client C_ϕ be associated with the requests $\{R_1, \ldots, R_n\}$, issued at times $R_1.time, \ldots, R_n.time$. This client is considered active if $time_{present} - R_n.time \leq t^c_{max}$.

Objective: Predict the next request, given the prediction model, the previous requests, and the confidence threshold for each order of the model.
Input: The trie structure T, representing the prediction model of order m; a set of the last k requests, R_i, $0 \leq i \leq k \leq m$; and the confidence threshold for each order, $c_{th}[j]$, $0 \leq j \leq m$.
Output: A set P of predicted pages.

(initialization is not actually a separate phase; current contexts are updated as new requests arrive)

```
for length j=1 to k
   current_context[j]:=node of depth j, representing the access
                       sequence {R[k-j+1],..., R[k]};


P:=NULL;
for length j=k down to 1
   for every child-node C of current_context[j]
      if (occurrence_count of C)/(occurrence_count of parent) >= c_th[j]
         P:=P+C;
remove duplicates from P;
```

Figure 4.3: Algorithm for making predictions, using multiple orders and different confidence thresholds.

When a client becomes inactive its cache is emptied of all its contents. (In the case of this simulation the interaction of the clients with only a particular Web server offering prefetching functionality is examined. Therefore, all the contents correspond to pages belonging to the same Web site.) Setting the t^c_{max} parameter to infinity will prevent the client cashes from ever being flushed.

The intuition for this scheme is that pages originating from a particular server do not have infinite lifetime in the client's cache. As the user navigates in the Web new pages from other servers replace the old ones. This behaviour is simulated by emptying the cache at specific time intervals.

It is usually the case that the number of prefetched pages is smaller than the size of the cache. Thus, a replacement policy should be applied, for deciding which pages to evict from the cache to make space for the incoming ones. There are no constraints in choosing such a policy, and the existing algorithms need only minor adaptations to operate in the new environment. In this study, the cache is implemented using the LRU replacement policy. The prefetched pages are put in the cache as if they were demand fetched, they are marked as most recently used, and they replace the least recently used pages. The maximum size of the cache is determined either in terms of storage size (i.e., in bytes), or in terms of number of Web pages. Note that the caches in the Web store documents in their entirety, unlike to the operating systems context where usually the cache operates on blocks of files.

4.2 Restrictions of the Environment

There are certain factors in the Web environment that impose limitations on the performance of the algorithm. These are inherent characteristics of the Web. Some of them can be remedied (either with a change in the protocol, or by incurring an extra cost in the system), while others are fixed.

Host Name and User Correspondence. There does not exist an one-to-one correspondence between the host identifier listed in the access log file and the actual users at that host. It may be the case that more than one user, or even worse a community of users (i.e., when a proxy server is employed), are serviced by the same machine. Ways of addressing the problem of user identification have been proposed [Pit97], yet, they are rather cumbersome, and have not become a common practice. Thus, it is unavoidable for the prediction model to erroneously mix some distinct browsing sessions into a single one.

Client-side Caches. Both individual users' caches and proxy caches constitute sources of noise for the prediction model. User requests serviced by a cache cause the access patterns perceived by the server to be false. Consequently, the prediction model becomes inaccurate as well.

Huge Number of Web Pages. An important issue when operating in the Web domain is that the alphabet is no longer limited. On the contrary, the number of pages that a server offers is considerable (in the order of thousands or even tens of thousands). In addition, this set of pages may be changing dynamically, by adding new pages or removing old ones. Therefore, the prediction algorithm should be adapted to this context.

Memory Requirements. The extremely large size of the alphabet and its dynamic nature lead to one more concern, the amount of memory required to store the model. This study does not address the memory issue, yet, several techniques in the literature propose schemes for limiting the data structure size [Sto88, BB92]. These techniques specify an upper bound M for the amount of memory the model may consume. Then, the alternatives include freezing the model when it reaches this bound (it is no longer updated with new access patterns); discarding the model, and rebuilding it from scratch each time there is no more free memory; freezing the model at size $M/2$ and start building a new one, while making predictions from the old model; using an LRU strategy to decide which node to evict from the model in order to accommodate the new access patterns and at the same time maintain its size below M.

Furthermore, a careful implementation of the model, in terms of information representation, could result in significant memory savings. A major step in this direction would be the efficient encoding of the URIs of the Web pages. Evidently, mapping each URI to an integer (through a hashing function) is a much more concise description of a Web page compared to the naive approach of storing every URI as a string.

4.3 Prefetching in the Real World

The incorporation of prefetching functionality into the existing infrastructure of the Web requires few changes. The architecture of the system with prefetching is depicted in Figure 4.4. The figure illustrates a set of clients accessing pages from a single server, through the Internet.

The major change in the system is the prediction engine module, which is running as a separate process, in parallel with the server's HTTP daemons. Note that the server may spawn several processes (daemons) to service the client requests, yet, all of them interact with a single prediction process. However, an architecture where multiple copies of the prediction process operate in parallel on a single shared instance of the model should be feasible.

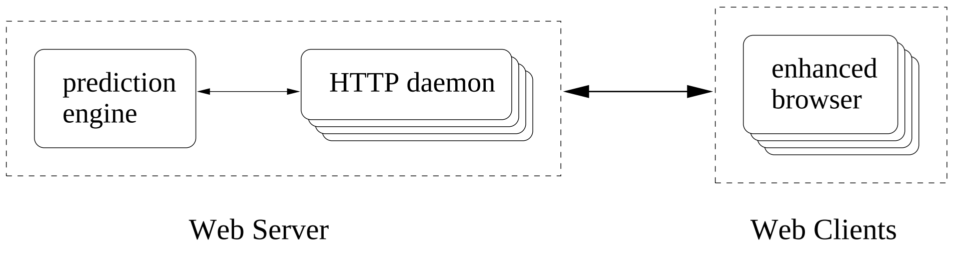


Figure 4.4: Proposed architecture of the system in a real implementation.

On the client side, the required changes involve instrumenting the browser to accept the additional prefetched pages, and store them in the cache. Although this approach demands minimal effort, the performance of the system improves when the client becomes *cooperative.* A cooperative client may assist the prefetching server in several ways:

- inform the server which pages are already residing in its cache
- communicate special characteristics of the browser (e.g., no support for images)
- decide whether prefetching is beneficial

The aforementioned cases contribute to a more informed prefetching which leads to network traffic savings, and more effective resource utilization. The additional information exchange between the client and the server can be piggybacked on the normal request and reply messages. Thus, the overhead incurred will be insignificant.

Chapter 5

Results

This chapter presents the results obtained from the simulation experiments. The simulations explore the effects of varying several parameters, and determine the impact of these changes on the performance of the prefetching system. Finally, the efficiency of the algorithm is estimated in terms of the number of successful predictions made versus the number of all possible predictions.

5.1 Experimental Setup

All the simulations presented herein were driven by the access log files pertaining to the Web server of the Department of Computer Science of the University of Toronto. The simulator was coded in the C++ programming language, and compiled by the GNU g++ v2.7.2 utility. Both the compilation and the experiments took place on a Sun SPARCstation-20 running the SunOS 5.4 operating system.

5.1.1 The Parameters of the Simulation

The experiments evaluate the effect of several parameters on the algorithm. These parameters are both simulation- and algorithm-specific:

Order of the prediction model: Specifies the maximum context length that the prediction algorithm will use. (In the experiments it ranges from 3 to 6.)

Confidence: Defined as the number of occurrences of the current node divided by the number of occurrences of the parent node. It limits the number of pages prefetched by requiring that the path from the current page to the predicted one is preferred among all the available paths. It is used by the prediction engine. (In the experiments it ranges from 0 to 0.9.)

Previous requests: Refers to the number of requests that a particular client has issued to the server so far. (In the experiments it ranges from 0 to 4 pages.)

Number of predictions: Sets an upper-bound on the number of predictions that each context can make. (In the experiments it ranges from 1 to 7 pages.)

Browsing session idle time: This is the t^s_{max} time window length. It is used when constructing the prediction model in order to differentiate among distinct browsing sessions of the same client. (In the experiments it ranges from 5 to 45 minutes.)

Client cache size: Determines the size of the client cache. It should be noted here that across all experiments the client cache size is formulated in terms of number of Web pages, rather than number of bytes. This approach is more intuitive for the interpretation of the results, without altering their significance. (In the experiments it ranges from 1 to 50 pages.)

Client cache idle time: Constrains the maximum time that a cache may hold the Web pages sent by the server, while the client has not issued a new request. It is the t^c_{max} time window length. (In the experiments it ranges from 45 minutes to 3 hours.)

The last two parameters are not associated with the prediction algorithm. However, they are useful for the modeling of the client environment which is a vital part of the simulation.

5.1.2 The Performance Metrics

Three quantities, expressed in percentages, are used to evaluate the performance of the prefetching system:

Usefulness of predictions: The number of prefetched pages that the users requested divided by the total number of requested pages.

Accuracy of predictions: The number of prefetched pages that the users requested divided by the total number of prefetched pages.

Network traffic: The volume of network traffic when prefetching is employed divided by that in the non-prefetching case.

A prefetching system aims at maximizing the first two metrics, and the same time at minimizing the last one. It is obvious that these objectives are conflicting. The more pages are prefetched, the more probable it is for some of them to be accessed. Though, at the same time the return of value is lower (i.e., accuracy is decreasing), and the increase in network traffic is high. Thus, there is a trade-off among these quantities that the prefetching algorithm should take into account. The experiments presented in the following paragraphs depict how the different parameters of the algorithm affect this trade-off.

5.1.3 Trace Characteristics

The log file used in the experiments spans a time period of five days, and contains 211,300 user requests for Web documents, originating from 14,007 unique hosts. As mentioned earlier, only the successful HTML and text requests are taken into account, leaving 86,000 requests. From these, the first 29,650 (i.e., the first two days) are used as training data in order to build the prediction model. The rest 56,350 requests are used to simulate a real system with prefetching. Web pages are prefetched only during the second phase, yet, the model is being updated throughout the entire simulation.

In order to appraise the navigation patterns of the users as perceived by the prediction model, it is useful to know how many pages they request from the Web server before they leave. This information will also help in the fine-tuning of the prefetching algorithm. Figure 5.1 depicts the average number of pages requested in a single session, as the maximum browsing session idle time varies. The two curves represent the results for the cases where only the distinct pages in each session are taken into account, or all of them. The standard deviation for these experiments is reported in Table 5.1.

The results indicate that users tend to request only a few pages during a single session, although there exist sessions with significantly higher numbers of requested pages. However, the large values of standard deviation suggest that a considerable portion of the sessions involve three or less requests. This observation implies that many users either are not really

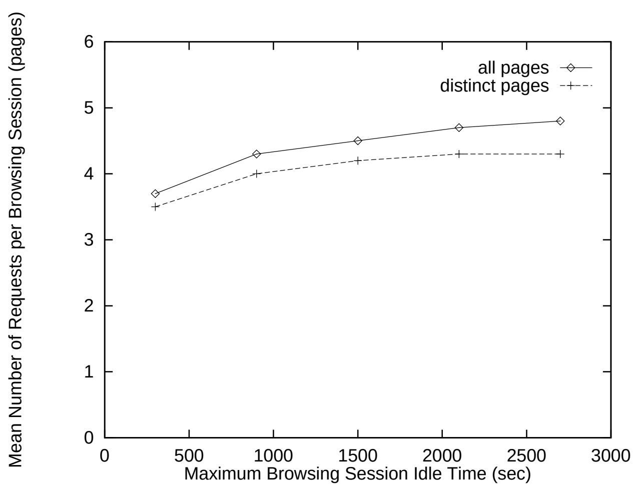


Figure 5.1: Mean number of page requests per browsing session, as a function of the maximum browsing session idle time.

all pages		*distinct pages*	
mean	std. dev.	mean	std. dev.
3.7	5.9	3.5	5.6
4.3	7.2	4.0	6.3
4.5	7.9	4.2	6.8
4.7	8.2	4.3	7.0
4.8	8.7	4.3	7.1

Table 5.1: Arithmetic mean and standard deviation for the number of page requests per browsing session. Results reported for maximum browsing session idle time set to 5, 15, 25, 35, and 45 minutes.

interested in navigating in the site, or they know exactly where to find the information they want. Therefore, prefetching will not be beneficial for them, and may be avoided.

Note that these results should not be interpreted as an exact description of the users' access behaviour. Several factors, such as the employment of proxy caches that hide a portion of the requests from the server, or the use of robots[1] by search-engines, distort the aforementioned figures. Nevertheless, they still represent the way the prediction model captures the access patterns of the users.

[1]Application programs that operate on the Web, and can semi-automatically crawl portions of the Web space in order to extract information.

5.2 Experimental Results

The subsequent paragraphs illustrate and discuss the outcome of the simulations. Each experiment investigates the influence of a single parameter at a time on the performance of the system. All three performance metrics are presented.

Unless otherwise noted, the simulations were executed assuming the following framework:

1. Prefetchers of order 3 to 6 were tested for each simulation.

2. It is the highest order context that makes predictions. In the cases where this is not possible (i.e., there is no context of respective length in the model matching the observed sequence), the responsibility moves on to the immediately lower order, until the context of unit length. No predictions are derived from the zero-order.

3. The confidence is set to 0. This allows prefetches to be made, even if they are not highly probable.

4. The previous requests parameter is set to 1. No pages are prefetched to clients that have just entered the Web server. Prefetching starts upon their second request.

5. Max predictions is set to 1. At most one page is prefetched along with each request.

6. The maximum browsing session idle time is 1,800 seconds (30 minutes). Any requests originating from a client that had been idle for at least that long are considered to constitute a new session.

7. The corresponding maximum idle time for client cache is set to 10,800 seconds (3 hours).

8. The client cache size is set to 50 pages, irrespective of the actual physical size of each individual page. Note though, that throughout the following experiments the maximum size of any prefetched page is restricted to under 60,000 bytes.

9. The experiments assume cooperative clients. Therefore, the prefetcher does not send to the client any pages that already reside in its cache.

5.2.1 Varying the Number of Previous Requests

As the previous requests parameter increases, the number of prefetched pages decreases (Figure 5.2). The prefetcher is only making predictions for clients that exhibit an interest in the Web site and do not leave after the first requests. This results in more accurate predictions, yet, the predicted requests are fewer.

5.2.2 Varying the Confidence

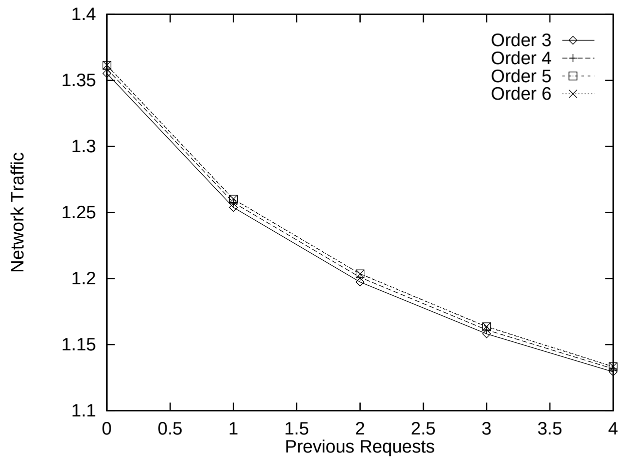
Similar patterns are observed when confidence increases (Figure 5.3). Larger values for this parameter cause fewer predictions to be made, thus, reducing the network traffic. However, the increment in prediction accuracy is significant since it is only the highly probable pages that are prefetched.

An interesting pattern occurs in the useful predictions metric, which is depicted in Figure 5.3.b. The lowest order seems to perform better than the rest for small values of confidence, and worse for the larger ones (though always having the best figures in accuracy). The reason is that higher orders offer predictions derived from larger contexts which convey more information, and are focused enough to surpass the confidence threshold. Nevertheless, these predictions are rather specialized, with a smaller supporting base, resulting in a little bit less accuracy.

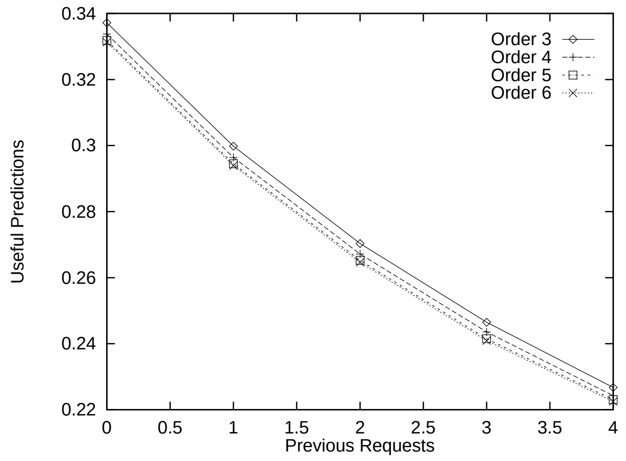
5.2.3 Varying the Number of Predictions

A way to predict as many requests as possible is by allowing the maximum predictions parameter to take large values, as represented in Figure 5.4. The number of useful predictions increase (note that prefetching every possible page would result in a perfect figure for this metric), but the return of value is minimal. The increase in traffic is dramatic, and the accuracy degrades quickly.

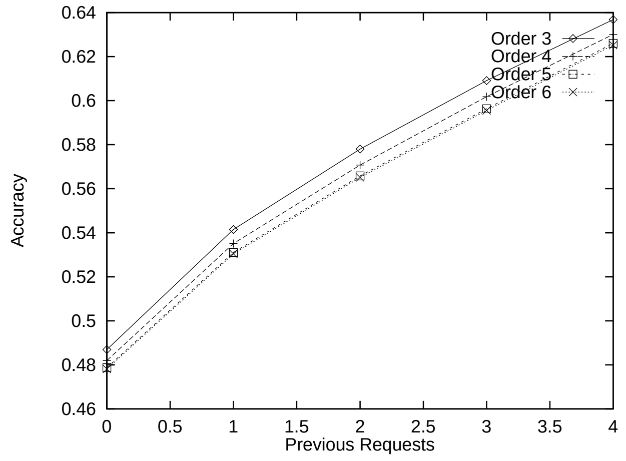
During this experiment the confidence is set to zero. Therefore, the prefetcher is actually forced to make many predictions, even though most of them are unlikely to be fruitful. Consequently, the higher orders, that retain a more restricted set of possible moves, predict more accurately (Figure 5.4.c).



(a)

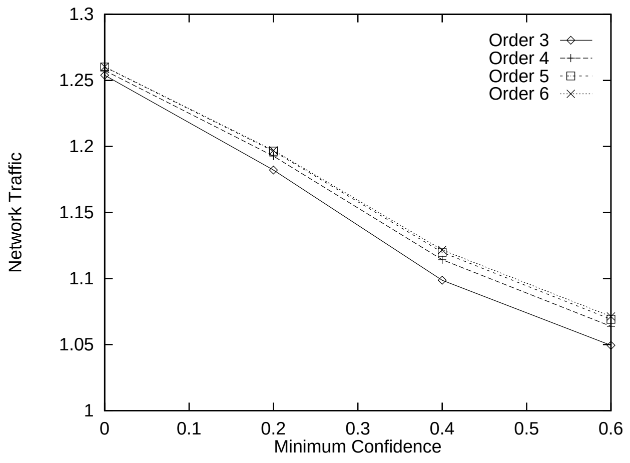


(b)

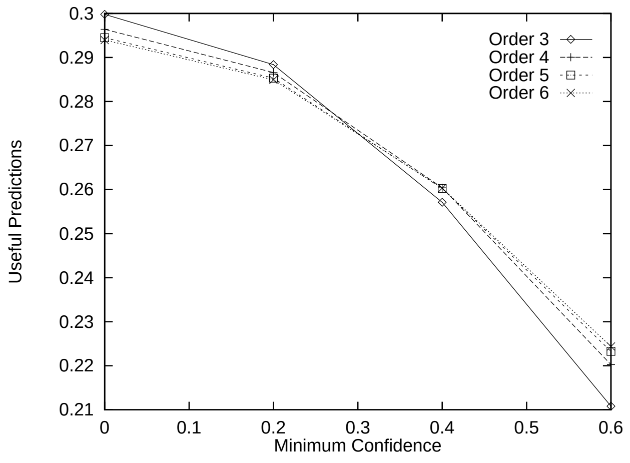


(c)

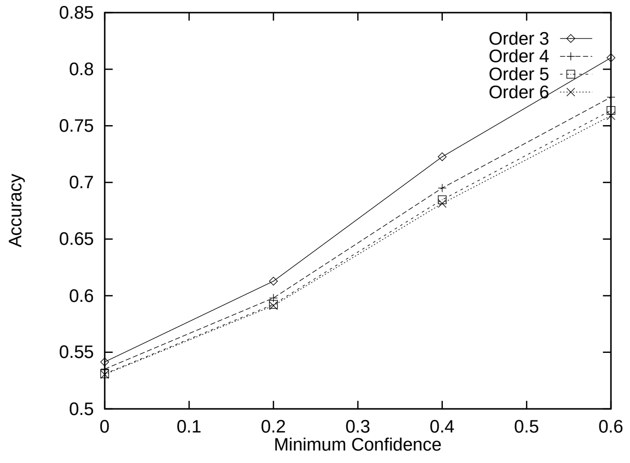
Figure 5.2: Varying the number of previous requests.



(a)

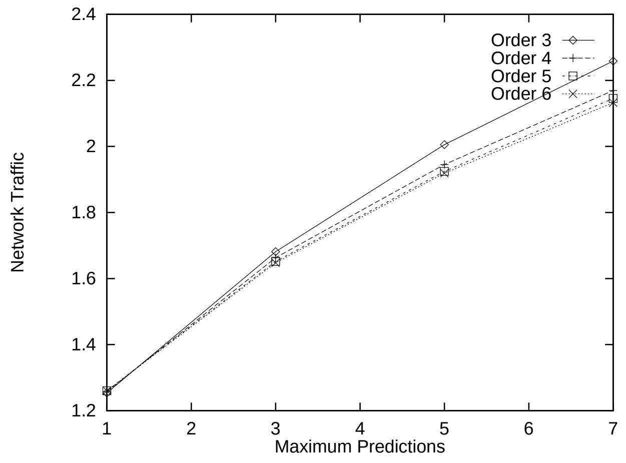


(b)

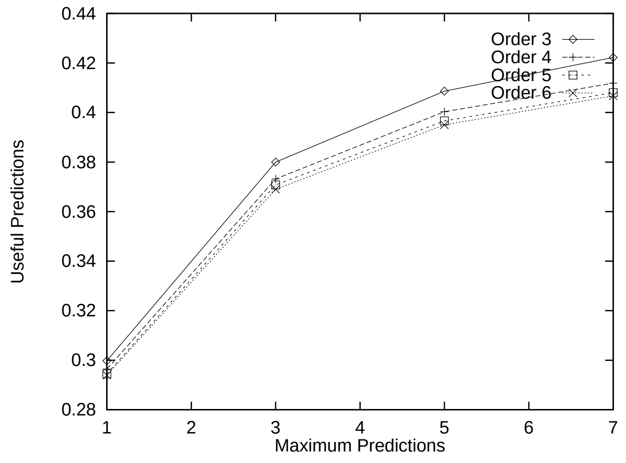


(c)

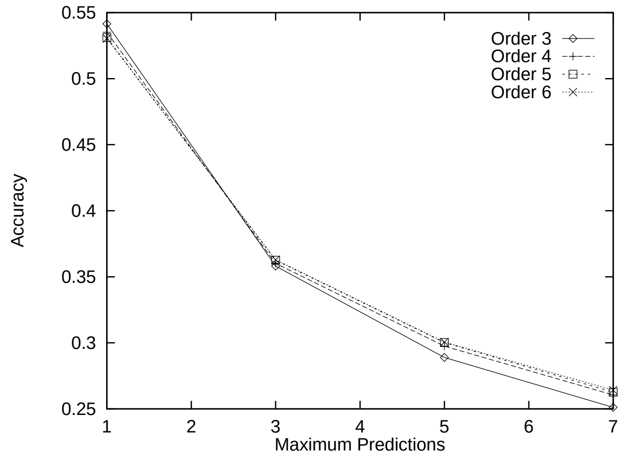
Figure 5.3: Varying the confidence.



(a)



(b)



(c)

Figure 5.4: Varying the number of predictions.

5.2.4 Varying the Browsing Session Idle Time

The experiment of varying the browsing idle time graphically depicts the effect of this parameter in the construction of the prediction model. The results of this simulations, where the confidence is set to 0.2, are illustrated in Figure 5.5. The best performance is achieved for small values of idle time, in the order of a few minutes.

Further increase of the threshold results in a slight improvement in the number of useful predictions at the cost of the accuracy. As the parameter moves to even larger values (i.e., above 30 minutes), more uncorrelated requests are considered to belong to the same session. This false classification causes a decline in the number of correct prefetches (Figure 5.6.b).

5.2.5 Varying the Client Cache Idle Time

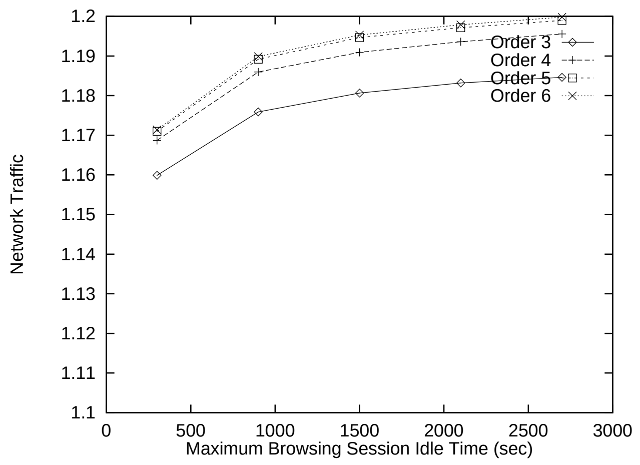
This parameter pertains to the client side of the simulation model, rather than the prefetching system per se. Nevertheless, it is useful to assess the way it affects the experiments. The results displayed in Figure 5.6 correspond to a confidence setting of 0.2.

As the maximum client cache idle time increases, network traffic and the number of useful predictions diminish. At the same time accuracy remains nearly constant, which suggests that the aforementioned reduction is due to an additional number of requests being serviced by the client cache, and not to a service degradation. When client caches in the simulation are left to live longer, it becomes more probable for a page already in the cache to be requested again. In the extreme case where the parameter is set to infinity (in conjunction with a sufficiently large size for the cache) most of the requested pages will end up residing in the cache.

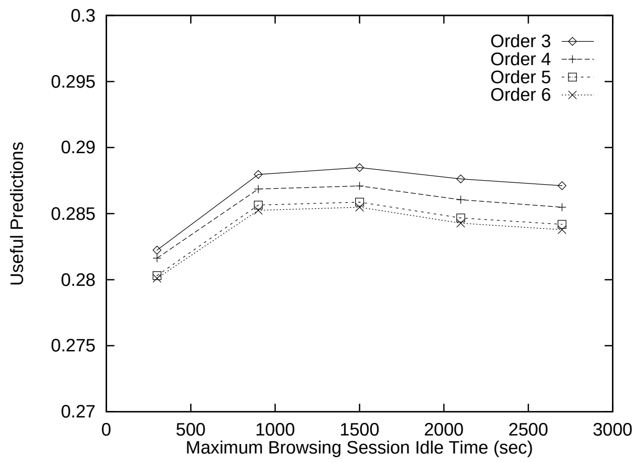
5.2.6 Varying the Client Cache Size

Simulations experimenting with different client cache sizes yield the results illustrated in Figure 5.7. The confidence threshold for these simulations is 0.2. Note that the size of 1 is the lowest value the parameter gets.

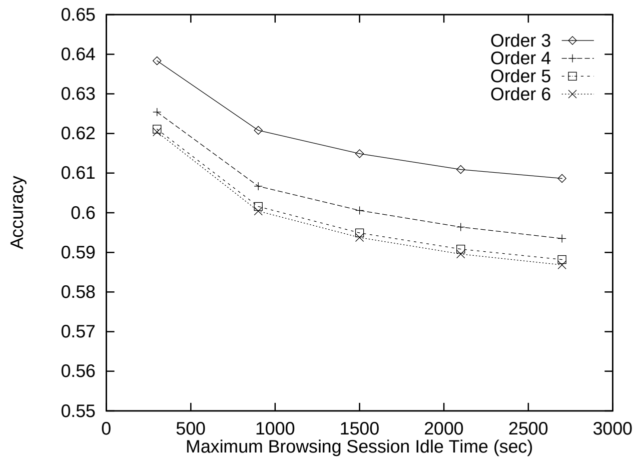
When the cache has room for a single page, any prefetched pages can only be used immediately after their arrival. Thus, the next request is the sole opportunity for the prefetched page to be used. Otherwise, it will be replaced in the cache by another page. As the cache size increases, the prefetched pages do not get evicted as often, and have the



(a)

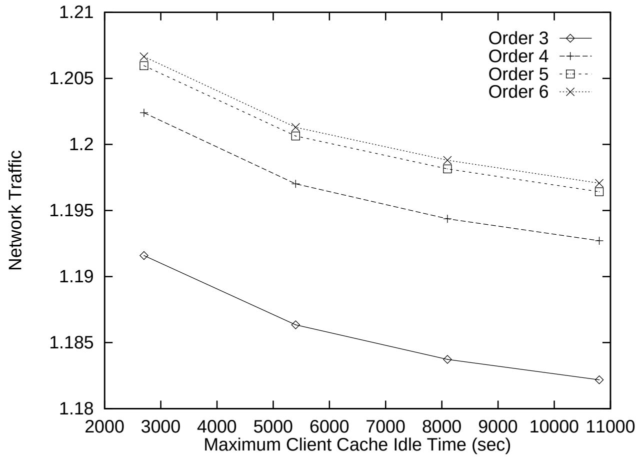


(b)

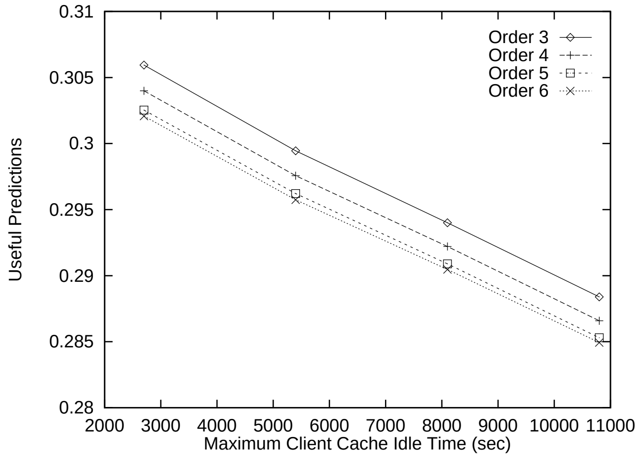


(c)

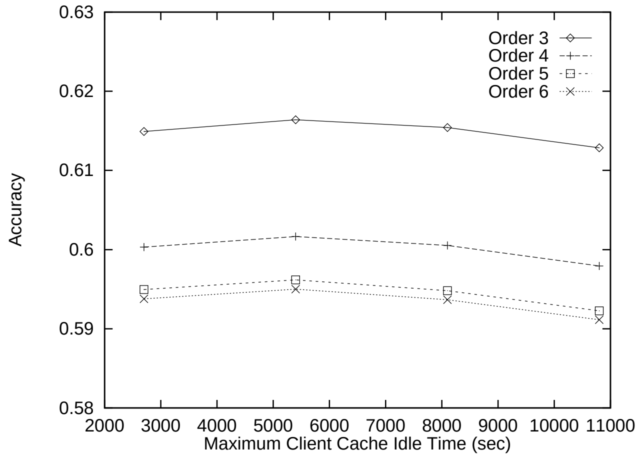
Figure 5.5: Varying the browsing session idle time.



(a)

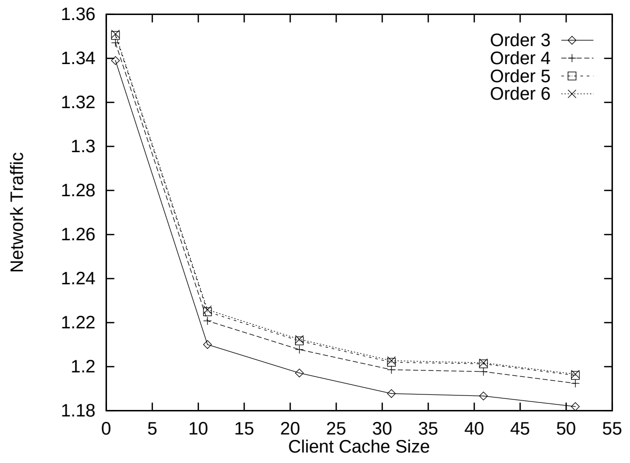


(b)

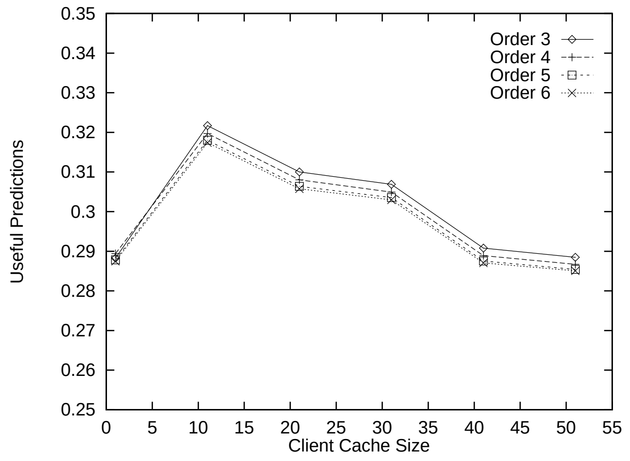


(c)

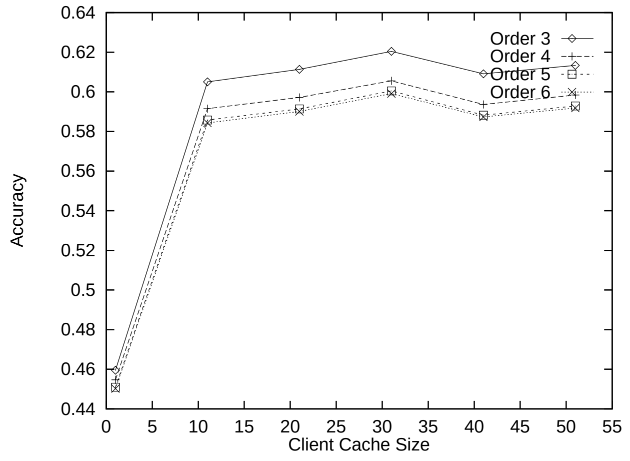
Figure 5.6: Varying the client cache idle time.



(a)



(b)



(c)

Figure 5.7: Varying the client cache size.

chance to be used in later requests as well. This causes the metrics in Figure 5.7.b and Figure 5.7.c to improve significantly.

However, a further increment in the size of the cache does not correspond to analogous improvements in the number of useful predictions (Figure 5.7.b). The explanation is twofold. First, the prefetched pages that are evicted from the cache before getting used when the cache size is smaller, are pages that are not going to be used either way (i.e., they are wrong predictions). Therefore they do not contribute anything to the statistics. Second, and more important, a large cache has the ability to service many of the requests of the client which reoccur. Consequently, the need for prefetching is diminished.

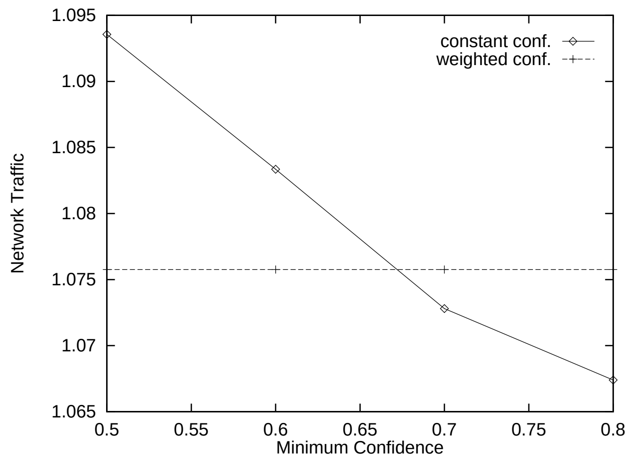
5.2.7 Combining Predictions

The representation of the prediction model, which stores multiple context orders in the same structure, offers the opportunity to combine the predictions from different orders. This experiment examines several variations of this approach. The parameters used for the maximum predictions, the client cache size, and the maximum client session idle time are 7 pages, 10 pages, and 2700 seconds respectively.

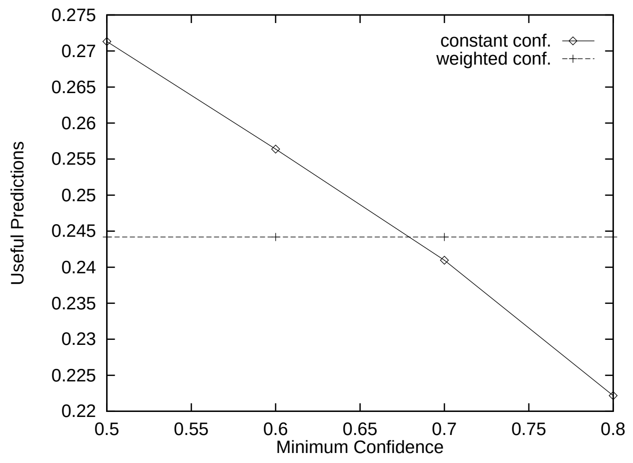
Figure 5.8 compares the use of a constant confidence value with *weighted* confidence, for a model of order 6. When weighted confidence is employed, a different confidence level is applied for making predictions from each order of the model. In contrast, a constant confidence value assigns the same weight to all orders, and treats them equally. However, higher order contexts embed more information pertaining to the access patterns. Thus, they can be trusted more by assigning them a smaller confidence. Note that in both cases predictions are made independently from each order of the model, and are then combined into a single answer set (i.e., the union operation is applied, and duplicates are discarded).

In the graphs of Figure 5.8 the horizontal lines represent the performance of the algorithm with weighted confidence which for the orders 2 to 6 takes the values 0.8 to 0.5 (reducing in steps of 0.08). This algorithm is compared with four others which use constant confidence with values from 0.5 up to 0.8. The results suggest that the weighted confidence approach tries to achieve good performance in all three metrics. Indeed, it performs above the average of the four constant-confidence algorithms in all categories.

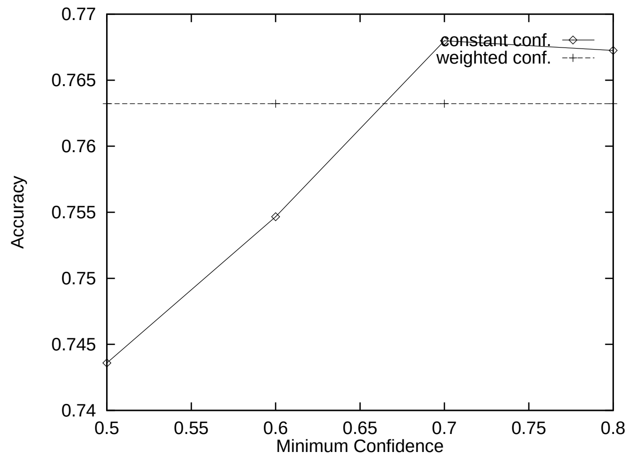
In Figure 5.9 weighted-confidence algorithms using different maximum order models are compared. The prediction models have orders ranging from 3 up to 6, while the weighted



(a)

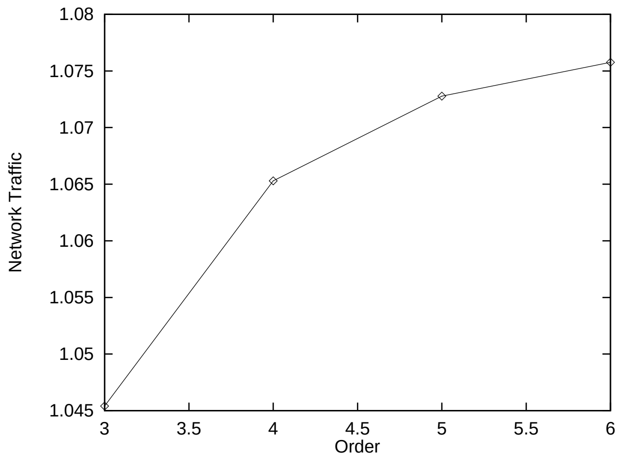


(b)

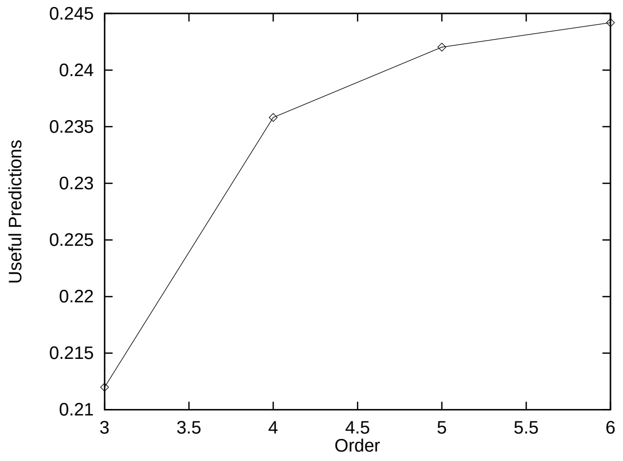


(c)

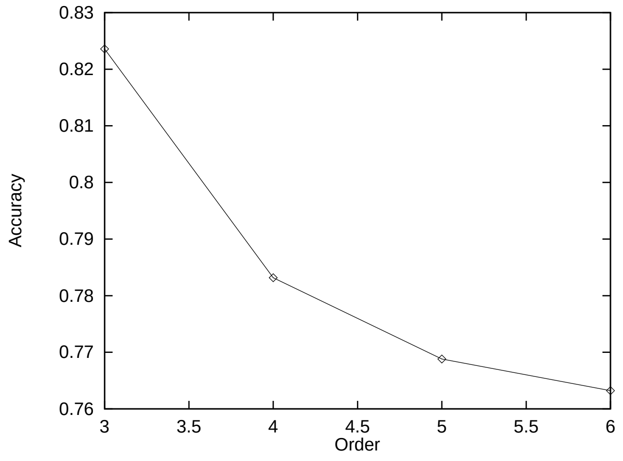
Figure 5.8: Constant and weighted confidence when combining predictions from different orders.



(a)



(b)



(c)

Figure 5.9: Varying the order of the model when combining predictions from different orders.

confidence is set to 0.8 for the context of order 2 and reduces by 0.08 for each subsequent order. The results show that the number of useful predictions increase along with the maximum order of the model, at the cost of the accuracy.

5.3 Efficiency of the Algorithm

It is evident that the performance of the algorithm on any of the specified metrics will vary depending on the parameters. Different settings may limit the algorithm to predict only the highly probable pages, or relax the constraints and result in more page requests being predicted. Two such cases are presented in Table 5.2. The first experiment predicts almost a quarter of the total number of requests issued to the Web server, with an accuracy of 80%. The second one improves the accuracy to 90%, which causes the number of predicted requests to shrink to less than one fifth.

max order	*confidence*	*prev. requests*	*network traffic*	*useful pred.*	*accuracy*
6	0.6-0.8	1	1.05	0.23	0.8
3	0.8-0.9	4	1.02	0.18	0.9

Table 5.2: This table summarizes the results obtained from simulating two particular instances of the algorithm, which use weighted confidence to combine predictions from every available order. The parameters common to both experiments are maximum predictions 5 pages, client cache size 30 pages, browsing session idle time 600 seconds, and client cache idle time 5400 seconds.

5.3.1 Potential Predictions

In order to assess the ability of the algorithm to make predictions, it is useful to know how many predictions can be done and how many of those were actually made. The potential predictions are restricted by two factors. First, the initial request of each browsing session cannot be predicted since the user may enter the Web site at an arbitrarily large number of points in its structure. Furthermore, the server does not know which client and when will start accessing the Web site. Second, the model cannot predict any sequence of requests that has not been observed before. Despite the fact that the algorithm may assign a non-zero probability for such events so as to be able to predict them, the extremely wide range of possible outcomes makes this approach impractical.

The histogram of Figure 5.10 shows the percentage of requests that can be predicted

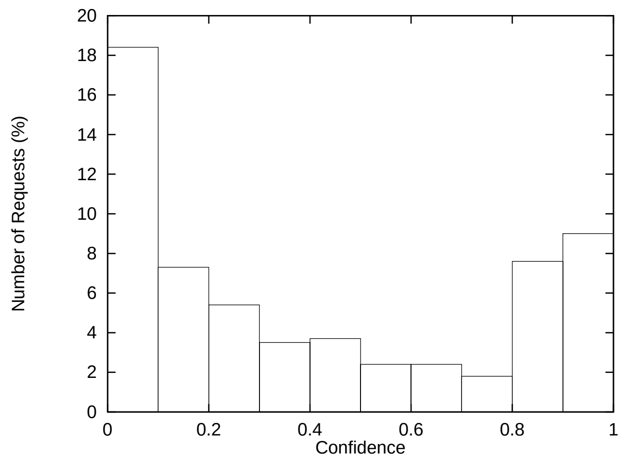


Figure 5.10: Percentage of requests that can potentially be predicted.

for the log file used in this study. These are the requests of the form $\{AB\}$, where page B is accessed after page A, and this pattern has occurred before. The number of requests is reported separately for various levels of confidence. The total number of potential predictions (i.e., the sum of all the bars in the histogram) accounts for 60% of the requests, while the remaining 40% includes first accesses to the server as well as request patterns that are encountered for the first time. As the appearance of the histogram suggests, the majority of the predictions are accompanied by low probabilities, and if selected will lead to inferior degrees of accuracy.

The experiments presented in Table 5.2 indicate that the predictions made are actually more than anticipated by the figures cited in the histogram. Indeed, taking into consideration the confidence levels used in the experiments, the predictions are about 15% more than expected. Note that even this estimate is pessimistic since the predictions in the simulations do not start right after the first access to the server. This outcome is explained by the fact that in the experiments predictions are also derived from high-order models, which usually exhibit greater confidence levels.

5.3.2 Cache Benefits

The simulations also show that prefetching can be used in a complementary fashion to the caching technique. Caches in the Web environment do not operate as efficiently as in the operating systems domain. The fast changing nature of Web documents and user interests limit the performance of caches. Prefetching has the potential to assist caches to store pages that are more likely to be requested again.

Figure 5.11 graphically depicts the considerable benefits that prefetching can offer. The graph reports the cache hits for various cache sizes, with and without the use of the prefetching technique. In the former case, the parameters of the algorithm were the same as in the first experiment of Table 5.2. When prefetching is employed, the cache experiences up to 6.5 times more cache hits than in the non-prefetching case. As Figure 5.11 shows, the relative benefits increase as the cache size gets smaller.

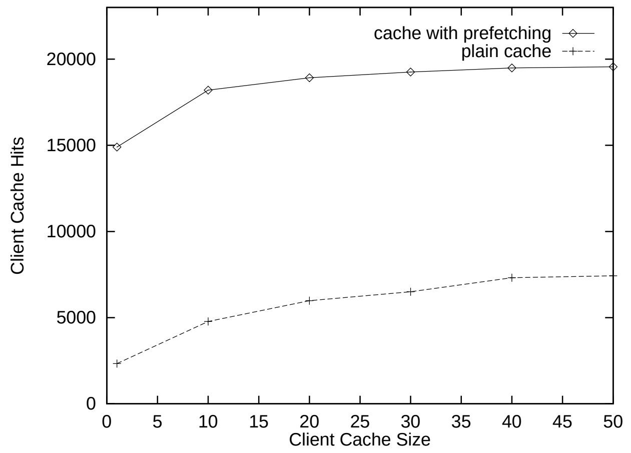


Figure 5.11: Cache hits with and without prefetching, for different cache sizes.

Chapter 6

Conclusions

6.1 Summary

This work investigated the application of a multi-order context modeling technique for prefetching in the Web domain. The prefetching method presented herein is based on the Prediction by Partial Match algorithm. The special characteristics of the Web context were recognized, and the algorithm was tailored to fit this new environment. The results indicate that the prediction of future client requests is feasible. Despite the heterogeneous nature of the users (their geographic origin, the reason they are visiting the Web site, the information they are looking for) which imposes limitations on the performance of the prefetching system, the simulation still suggests that the advantages of employing prediction are significant. Evidently, the users that are aimlessly navigating in the Web, hopping from one server to another, will not benefit. However, the proposed scheme will assist those users who are consistently following some regular access patterns when searching for some information. (Though, it is debatable whether the aforementioned characteristic would be a desired one for a society.)

The present work does not quantify the decrease in time latencies perceived by the clients. Yet, a recent study [CB98] suggests that prefetching in the Web (with performance levels similar to those presented in this work) not only reduces time latencies, but also has the potential to improve the network traffic properties. Correct predictions of future traffic enable better utilization and management of the available network resources.

So far, only caching methods have been applied in order to reduce time latencies. However, the effectiveness of caches in the Web domain is rather limited, due to the fast changing nature of the medium. Prefetching can help in this direction by causing the cache to hold documents relevant to user requests. It is the combined use of caching and prefetching techniques that yields the best results.

6.2 Future Work

The work presented in this thesis can be extended in several ways. The main directions are to investigate the notion of *transaction* in the browsing model, to explore more elaborate ways of making predictions, and to introduce flexibility in the algorithm.

User Transaction. This is the analogue to the *browsing session* used in this work. As discussed earlier, the correct definition and subsequent identification of the user transactions is crucial for the construction of the prediction model. Two problems arise here. The distinction among several users of the same machine (who appear as one entity in the Web server's access log file), and the grouping of their requests in transactions. An interesting suggestion for the latter problem is given by Chen et al. [CPY96]. The authors propose the use of *maximal forward references*, which is a way of filtering out the effect of some backward references.

Prediction Engine. The prefetcher can be extended to accommodate the need to make predictions further in the future (i.e., several steps ahead of the user requests).

It would also be interesting to examine other schemes for selecting which particular pages to prefetch. In most cases, Web pages are stored in hierarchical directories depending on their content, and users tend to follow this structure when navigating within a site. Therefore, considering the path names of Web pages may assist the prefetcher in making more accurate predictions. Moreover, a method for constraining the possible predictions is to consider only those pages which are referenced by the currently requested page. This technique is employed by the Letizia system [Lie95], which is a Web agent for proposing interesting pages to the user.

General Enhancements to the Algorithm. The common characteristic here is the adaptiveness of the system. First, the prediction model should be able to automatically respond to changes in the user access patterns. An aging mechanism could be incorporated to address this issue. Second, the algorithm should adjust to server load. When the load is high the predictions should be few, or even none, so as not to degrade the performance of the system. Last, the prefetcher should cooperate in a tighter fashion with the network layer. Information concerning network traffic and queues could be exploited for achieving a better prediction strategy, which would boost overall performance.

Bibliography

[Ama98] Amazon.com. http://www.amazon.com, March 1998.

[Apa98] Apache HTTP Server Project. http://www.apache.org, March 1998.

[AW97] Martin F. Arlitt and Carey L. Williamson. Internet Web Servers: Workload Characterization and Performance Implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, 1997.

[BAD$^+$92] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, September 1992.

[BB92] Suzanne Bunton and Gaetano Borrielo. Practical Dictionary Management for Hardware Data Compression. *Communications of the ACM*, 35(1):95–104, 1992.

[BBLS91] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Centre, 1991.

[BCW90] T. C. Bell, J. C. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, 1990.

[Bes96] Azer Bestavros. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time in Distributed Information Systems. In *International Conference on Data Engineering*, pages 180–189, New Orleans, LO, February 1996.

[BLCL$^+$94] T. Berners-Lee, R. Cailliae, A. Luotonen, H. F. Nielsen, and A. Secret. The world wide web. *Communications of the ACM*, 37(8):76–82, 1994.

[BLFF95] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol - HTTP/1.0. *RFC 1945, Internet request for comments*, 1995.

[CB98] Mark Crovella and Paul Barford. The Network Effects of Prefetching. In *IEEE Infocom*, San Francisco, CA, USA, 1998.

[CKV93] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical Prefetching via Data Compression. In *ACM SIGMOD International Conference*, pages 257–266, Washington, DC, USA, June 1993.

[Con98] Connectix Corp. http://www.connectix.com, March 1998.

[CPY96] Ming-Syan Chen, Jong-Soo Park, and Philip S. Yu. Efficient Data Mining for Path Traversal Patterns. In *International Conference on Distributed Computing Systems*, pages 385–392, May 1996.

[CW84] John G. Cleary and Ian H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[Dep98] Department of Computer Science, University of Toronto. http://www.cs.toronto.edu, March 1998.

[DFKM97] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. Rate of Change and other Metrics: a Live Study of the World Wide Web. In *USENIX Symposium on Internet Technology and Systems*, pages 147–158, Berkeley, CA, USA, December 1997.

[FGM$^+$97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. *RFC 2068, Internet request for comments*, 1997.

[Fos98] Cormac Foster. The Price of Speed. http://www.cnet.com/ Content/ Reviews/ Compare/ Accelerator, March 1998.

[GA96] James Griffioen and Randy Appleton. The Design, Implementation, and Evaluation of a Predictive Caching File System. Technical Report CS-264-96, Department of Computer Science, University of Kentucky, June 1996.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM Conference*, pages 314–329, August 1988.

[KL96] Thomas M. Kroeger and Darell D. E. Long. Predicting File System Actions from Prior Events. In *Winter USENIX Conference*, 1996.

[KLM97] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *USENIX Symposium on Internet Technologies and Systems*, pages 319–328, San Diego, CA, USA, January 1997.

[Knu73] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[KTP$^+$96] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *USENIX Association Symposium on Operating Systems Design and Implementation*, pages 19–34, October 1996.

[KW97a] Achim Kraiss and Gerhard Weikum. Vertical Data Migration in Large Near-Line Document Archives Based on Markov-Chain Predictions. In *VLDB International Conference*, pages 246–255, Athens, Greece, August 1997.

[KW97b] Balachander Krishnamurthy and Craig E. Wills. Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web. In *USENIX Symposium on Internet Technology and Systems*, pages 1–12, Berkeley, CA, USA, December 1997.

[LD97] Hui Lei and Dan Duchamp. An Analytical Approach to File Prefetching. In *USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, January 1997.

[Lie95] Henry Lieberman. Letizia: An Agent that Assists Web Browsing. In *International Joint Conference on Artificial Intelligence*, pages 924–929, August 1995.

[LRV97] Paolo Lorenzetti, Luigi Rizzo, and Lorenzo Vicisano. Replacement Policies for a Proxy Cache. http://www.iet.unipi.it/ ~luigi/ caching.ps.gz, 1997.

[Mar96] Evangelos P. Markatos. Main Memory Caching of Web Documents. In *International World Wide Web Conference*, Paris, France, May 1996.

[MDFK97] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *ACM SIGCOMM Conference*, pages 181–194, Cannes, France, September 1997.

[MDK96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *USENIX Association Symposium on Operating Systems Design and Implementation*, pages 3–17, October 1996.

[MJHS96] Bamshad Mobasher, Namit Jain, Eui-Hong Han, and Jaideep Srivastava. Web Mining: Pattern Discovery from World Wide Web Transactions. Technical Report TR96-050, Department of Computer Science, University of Minnesota, March 1996.

[MRG97] Carlos Maltzahn, Kathy J. Richardson, and Dirk Grunwald. Performance Issues of Enterprise Level Web Proxies. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 13–23, New York, NY, USA, June 1997.

[NGBS$^+$97] Henrik Frystyk Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *ACM SIGCOMM Conference*, 1997.

[OCH$^+$85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Symposium on Operating Systems*, pages 15–24, Orcas Island, WA, USA, December 1985.

[Ous90] John K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Summer USENIX Conference*, pages 247–256, 1990.

[Par98] Parsons Technology Inc. http://www.parsonstech.com, March 1998.

[Pax94] V. Paxson. Empirically-Derived Analytic Models of Wide-Area TCP Connections. *ACM/IEEE Transactions on Networking*, 2(8):316–336, 1994.

[Pea98] PeakSoft Corp. http://www.peak.com, March 1998.

[PGS93] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. *SIGOPS Operating Systems Review*, 27(2):21–34, 1993.

[Pit97] James Pitkow. In Search of Reliable Usage Data on the WWW. In *International World Wide Web Conference*, pages 451–463, Santa Clara, CA, USA, April 1997.

[PM94] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP Latency. In *International World Wide Web Conference*, Chicago, IL, USA, October 1994.

[PM96] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *ACM SIGCOMM Computer Communication Review*, 27(3):22–36, 1996.

[Pos81] J. Postel. Transmission Control Protocol. *RFC 793, Network Information Center, SRI International*, 1981.

[Rob98] RobSoft Inc. http://www.robsoft.com, March 1998.

[SF98] Myra Spiliopoulou and Lukas C. Faulstich. WUM: A Web Utilization Miner. In *International Workshop on the Web and Databases*, Valencia, Spain, March 1998.

[SSV97] Peter Scheuermann, Junho Shim, and Radek Vingralek. A Case for Delay-Conscious Caching of Web Documents. In *International World Wide Web Conference*, Santa Clara, CA, USA, April 1997.

[SSV98] Junho Shim, Peter Scheuermann, and Radek Vingralek. A Unified Algorithm for Cache Replacement and Consistency in Web Proxy Servers. In *International Workshop on the Web and Databases*, Valencia, Spain, March 1998.

[Sto88] James A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[WAS$^+$96] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *ACM SIGCOMM Conference*, pages 293–305, New York, NY, USA, August 1996.

[Web98a] Web3000 Inc. http://www.web3000.com, March 1998.

[Web98b] WebEarly Inc. http://www.webearly.com, March 1998.

[WS97] Craig E. Wills and Joel Sommers. Prefetching on the Web Through Merger of Client and Server Profiles. http://www.cs.wpi.edu/ ~cew/ papers/ webprofile.ps, 1997.