
UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

**DEPARTMENT OF COMPUTER ENGINEERING**

# Web Prefetching Techniques in Real Environments

Thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computing

Bernardo Antonio de la Ossa Pérez

Ph.D. advisors:
Dr. Ana Pont Sanjuán
Dr. José A. Gil Salinas

València, March 2011

# Acknowledgments

Firstly, I would like to acknowledge the work of Dr. Julio Sahuquillo as advisor of this Ph.D. thesis, although his name does not appear on the cover due to the limit on the number of advisors established by the university. His help, together with the contribution and knowledge of Prof. Ana Pont and Dr. José A. Gil, have been crucial for the development of this work.

# Abstract

This dissertation studies the prefetching technique applied to the World Wide Web from a realistic and practical point of view. Web prefetching is a technique to reduce users perceived latency by predicting and pre-processing next user accesses.

Until now the open research literature about web prefetching has focused on theoretical questions and has not taken into account some of the problems that arise when implementing the technique in real conditions. Furthermore, previous works have used simplified models for evaluation that do not consider how practical issues really affect the implementation of a prefetching technique. Moreover, only few works have considered performance indexes that are relevant to users when evaluating the benefits that prefetching can achieve.

In order to overcome these three limitations, we developed Delfos, a web prefetching framework that implements web prediction and prefetching in a real environment, and can be integrated into the web architecture without modifying the standard web protocols and in a compatible way with the existing software products. Delfos can also be used to evaluate and compare existing prefetching techniques and algorithms and to assist in the design of new ones because it provides detailed statistics reports. As an example, Delfos is used to propose, test and evaluate a new technique (Predict at Prefetch, P@P) able to considerably reduce the users perceived latency with no additional cost compared to the basic prefetch mechanism.

The prediction algorithms proposed in the research literature that achieve the highest precision involve a high computational cost, which is an important drawback for including them in real systems. To deal with this disadvantage, a novel low-cost web prediction algorithm (Referrer Graph, RG) is proposed in this PhD dissertation. This algorithm learns from users accesses and builds a Markov model that can distinguish between dependencies in objects of the same page and objects of different pages by using the object URI and the referrer in each request. RG includes a prune mechanism that controls the computational resource consumption while sustaining the performance.

This dissertation also includes an empirical study to investigate the maximum benefits that web users can expect from prefetching techniques in the current web. Unlike previous theoretical studies, this work considers a realistic prefetching architecture using real and representative traces. In this way, the influence of real implementation

constraints are considered and analyzed. The results obtained show that web prefetching can improve page latency up to 52% in the studied traces, which encourages researchers to focus future works on this direction.

# Resum

Aquesta tesi estudia l'aplicació a la World Wide Web (WWW) de les tècniques de prebusca des d'un punt de vista realista i pràctic. La prebusca s'aplica a la web per a reduir la latència percebuda pels usuaris ja que, bàsicament, consisteix a predir i preprocessar els següents accessos dels usuaris.

Fins ara, la literatura disponible sobre la prebusca web s'ha concentrat en qüestions teòriques i no ha considerat alguns dels problemes que apareixen en implementar la tècnica en condicions reals. D'altra banda, els treballs d'investigació existents usen per a l'avaluació models simplificats que no consideren com els aspectes pràctics afecten realment la implementació d'una tècnica de prebusca. A més, apenes uns pocs treballs han usat índexs de prestacions que siguen rellevants per als usuaris en l'avaluació dels beneficis que la prebusca pot aconseguir.

A fi de superar aquestes tres restriccions, s'ha desenvolupat Delfos, un entorn de prebusca web que implementa predicció i prebusca en un Sistema real, pot integrar-se a l'arquitectura web sense realitzar modificacions en els protocols web estàndard, i és compatible amb els programes existents. Delfos també pot usar-se per a avaluar i comparar tècniques de prebusca i algoritmes de predicció així com ajudar en el disseny d'uns altres de nous ja que proporciona informació estadística detallada dels experiments duts a terme. A manera d'exemple, Delfos s'ha usat per a proposar, provar i avaluar una nova tècnica (Predir a la Prebusca, P@P) que és capaç de reduir considerablement la latència percebuda per l'usuari sense costos addicionals respecte al mecanisme de prebusca bàsic.

Els algoritmes de predicció proposats en la literatura d'investigació que aconsegueixen la millor precisió incorren en un alt cost computacional, i açò representa un problema per a incloure'ls en Sistemes reals. Per a minorar aquest inconvenient, en aquesta tesi es proposa un nou algoritme de predicció de baix cost, (Referrer Graph, RG). Aquest algoritme aprèn dels accessos d'usuaris i construeix un model de Markov que distingeix entre dependències d'objectes de la mateixa pàgina i objectes de distintes pàgines usant la URI i la informació de referència de l'objecte indicat en la petició. RG inclou un mecanisme de poda que controla el consum de recursos computacionals mentre manté el rendiment.

Aquesta tesi també inclou un estudi empíric que investiga els màxims beneficis que els usuaris de la web poden esperar de les tècniques de prebusca en la web actual.

Contràriament a altres estudis teòrics previs, aquest treball considera una arquitectura de prebusca realista, usant traces reals i representatives. D'aquesta manera es considera i analitza la influència de les restriccions d'implementació reals. Els resultats obtinguts mostren que la prebusca web pot reduir la latència de pàgina fins en un 52% en les traces estudiades, la qual cosa incentiva la realització d'un major esforç investigador en aquesta direcció.

# Resumen

Esta tesis estudia la aplicación a la World Wide Web (WWW) de las técnicas de prebúsqueda desde un punto de vista realista y práctico. La prebúsqueda se aplica a la web para reducir la latencia percibida por los usuarios ya que, básicamente, consiste en predecir y preprocesar los siguientes accesos de los usuarios.

Hasta ahora, la literatura disponible acerca de la prebúsqueda web se ha concentrado en cuestiones teóricas y no ha considerado algunos de los problemas que aparecen al implementar la técnica en condiciones reales. Por otra parte, los trabajos de investigación existentes usan para la evaluación modelos simplificados que no considera n cómo los aspectos prácticos afectan realmente a la implementación de una técnica de prebúsqueda. Además, apenas unos pocos trabajos han usado índices de prestaciones que sean relevantes para los usuarios en la evaluación de los beneficios que la prebúsqueda puede lograr.

Con objeto de superar estas tres restricciones se ha desarrollado Delfos, un entorno de prebúsqueda web que implementa predicción y prebúsqueda en un sistema real, puede integrarse en la arquitectura web sin realizar modificaciones en los protocolos web estándar, y es compatible con los programas existentes. Delfos también puede usarse para evaluar y comparar técnicas de prebúsqueda y algoritmos de predicción así como ayudar en el diseño de otros nuevos ya que proporciona información estadística detallada de los experimentos llevados a cabo. A modo de ejemplo, Delfos se ha usado para proponer, probar y evaluar una nueva técnica (Predecir en la Prebúsqueda, P@P) que es capaz de reducir considerablemente la latencia percibida por el usuario sin costes adicionales respecto al mecanismo de prebúsqueda básico.

Los algoritmos de predicción propuestos en la literatura de investigación que alcanzan la mayor precisión incurren en un alto coste computacional, y esto representa un problema para incluirlos en sistemas reales. Para aminorar este inconveniente, en esta tesis se propone un nuevo algoritmo de predicción de bajo coste, (Referrer Graph, RG). Este algoritmo aprende de los accesos de usuarios y construye un modelo de Markov que distingue entre dependencias de objetos de la misma página y objetos de distintas páginas usando la URI y la información de referencia del objeto indicado en la petición. RG incluye un mecanismo de poda que controla el consumo de recursos computacionales mientras mantiene el rendimiento.

Esta tesis también incluye un estudio empírico que investiga los máximos beneficios que los usuarios de la web pueden esperar de las técnicas de prebúsqueda en la web actual. Contrariamente a otros estudios teóricos previos, este trabajo considera una arquitectura de prebúsqueda realista, usando trazas reales y representativas. De esta forma se considera y analiza la influencia de las restricciones de implementación reales. Los resultados obtenidos muestran que la prebúsqueda web puede reducir la latencia de página hasta en un 52% en las trazas estudiadas, lo cual incentiva la realización de un mayor esfuerzo investigador en esta dirección.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Objectives of the Thesis

## 1.1 General Objective

The main objective of this thesis is to demonstrate that web prefetching is an effective solution to reduce web latency perceived by the users, and that it can be implemented easily and efficiently in the current real environment.

## 1.2 Motivation

Web prefetching has not been widely used in the real world until now due to three main reasons. The first one is the limited user bandwidth that restricted the benefits of prefetching in the early Web. The second reason is that some early proposals required the modification of the standard web protocols in order to support web prefetching. Finally, the third reason is that the most efficient prediction algorithms proposed to predict next user accesses require a lot of computational resources in order to generate precise predictions. This is because these algorithms work by capturing previous user accesses in order to build a model of user patterns to predict future accesses.

Regarding the first reason, the ever-increasing bandwidth in the current Internet opens a new window for exploiting web prefetching, thus becoming an interesting option for improving web performance. Web browsers based on Mozilla already support the web prefetching technique.

This thesis focuses on solving the second and third mentioned drawbacks. In the first part of this dissertation, we check how web prefetching can be implemented in a real environment without modifying the standard HTTP 1.1 protocol, making it also compatible with current web browsers and servers. Then, we design some efficient web prefetching proposals that carefully consider resource consumption.

## 1.3 Partial Objectives

In order to accomplish the main objective of this dissertation, and considering the motivation previously outlined, several partial objectives have been proposed. These partial objectives are listed below and are individually addressed in the next chapters.

### 1.3.1 To build a prototype that considers the client, the server and the protocol

The first objective is to design and implement a framework to efficiently and easily perform web prefetching techniques, integrated in a real web architecture.

The prototype must be fully compatible with HTTP, making it suitable to be used with current browsers, web servers and protocols. It should also be able to serve as a framework to investigate how web prediction and prefetching techniques can work efficiently in a real environment without modifying the standard HTTP protocol.

### 1.3.2 To conduct a performance evaluation study of web prefetching techniques considering real conditions

A lot of research works were published focusing on prediction and prefetching algorithms, but few of them compare the performance of different proposals by using simulation or emulation tools. The main advantage of using these tools is their flexibility and immediacy providing results. Unfortunately, simulators may present significant result deviations since they are abstractions of the real world. As a consequence, there is a need to develop a tool in order to gather results when running prefetching algorithms in real environments.

To fill this gap, the prototype previously built can be extended to evaluate and compare the performance of web prefetching techniques under real conditions.

### 1.3.3 To study real factors that affect the web prefetching limits, and quantify their impact

The main goal is to explore the maximum benefits that web prefetching can achieve when working in the real world, i.e., the objective is to quantify the upper bounds in latency savings assuming real conditions. The experiments can be performed in the prototype previously built by implementing a *perfect* prediction algorithm which always provides accurate predictions. In this way, we can discern which performance losses come from miss-predictions and which ones come from the prefetching technique. This work should serve as a guide for future research works aimed to improve web prefetching.

### 1.3.4 To explore how web prefetching techniques can be improved

The aim is to design and test general web prefetching techniques that improve prefetch performance. Then, they will be implemented in the prototype and tested in real world conditions.

### 1.3.5 To propose a Low Cost Prediction Algorithm

Our objective is to design a new prediction algorithm that becomes a low-cost alternative to the existing ones, but capable of achieving precision values and web latency savings similar or even better than those ones of the best proposals that can be found in the open literature. The proposal will be carefully evaluated by using the prototype framework, and the results will be compared against the proposals of the literature, in the same conditions.

# Chapter 2

# Web Prediction and Prefetching

## 2.1 Introduction

Today the massive use of the web has increased the traffic in the network as well as the load that the web servers manage. Although nowadays web users have higher bandwidth connections, they still perceive high latencies when navigating the web due to overloaded elements (e.g., network, servers, switches, or intermediate hardware), long message transference times, and the Round Trip Time (RTT). Consequently, the reduction of the users perceived latency when browsing the web is still a crucial research issue.

The reduction of the web users perceived latency has been the subject of many research efforts over the past few years. The most popular techniques proposed to reduce this latency are web caching, geographical replication, and prefetching. Nowadays, caching techniques are widely implemented since they achieve important latency savings. Big companies usually implement web replication by using CDNs (Content Delivery Networks) [Rabinovich 02] to reduce their websites access time, but this solution is expensive and many small companies and organizations cannot afford it. Web prefetching techniques are orthogonal to caching and replication techniques, so that they can be applied together to achieve a better web performance. While caching and replication techniques have been widely implemented in real world, far fewer studies have investigated web prefetching in real environments.

This chapter presents a detailed description of the web prediction and prefetching techniques. The chapter is organized as follows: Section 2.2 presents the concepts of a generic web prefetching architecture, its components, how commercial web browsers implement it and other related considerations. Section 2.3 surveys the existing prediction algorithms. Section 2.4 outlines the published works related to performance

evaluation of web prefetching techniques. Finally, section 2.5 lists the current commercial products and prototypes that implement some kind of web prefetching.

## 2.2 Web Prefetching

The purpose of web prefetching is to preprocess object requests before the user explicitly demands those objects, in order to reduce the users perceived latency. Web prefetching involves two main steps. First, it is necessary to accurately predict the next user accesses. These predictions are usually made based on previous experience about users' accesses and preferences, and the corresponding hints are provided to a prefetching engine. Second, the prefetching engine decides which objects from the predicted hints are going to be prefetched.

The information that can be used by the prediction engine depends on the element of the web architecture (the client, the proxy or the server) at which the prediction engine has been located. When it is located at the client, only one user access pattern is used to perform predictions [Bestavros 95, Duchamp 99]. However, when the predictor is at the proxy server, it takes advantage of the multi-user and multi-server information gathered at this element to perform the predictions [Fan 99, Bouras 04, Teng 05]. If the engine is located at the server side, it makes predictions based on multi-user accesses to the same website [Domènech 06a, Schechter 98, Padmanabhan 96, Bestavros 96]. Finally, the predictions can be performed by several elements in collaboration [Markatos 98, Domènech 06e].

The prefetching engine can be implemented in any of the elements that receive the predictions results. The objects prefetched are stored in a cache waiting to be demanded.

This work assumes that the web server performs the predictions and provides the hints, and the web browser prefetches the objects during the browser idle time [Crovella 98]. We use this architecture because this is how most commercial products work, it does not require changes in the protocols, and it is the easiest way to implement it in practice.

We define page latency as the elapsed time from the time when the user demands a page until all the objects in the page are received. This time finishes earlier if the page download is canceled. We also define the browser idle time as the elapsed time since the last embedded object of a page is received until the next page is requested. Figure 2.1 illustrates these definitions in an example page.

The limitations on the available user's bandwidth constrained the benefits of prefetching in the past because prefetching can increase network traffic if its predictions are not accurate enough. This fact together with the difficulty in implementing these techniques without modifying the massively used protocols have left a gap between academic results and available products. But the current user's bandwidth opens again new possibilities for prefetching to improve web performance with a reasonable cost.

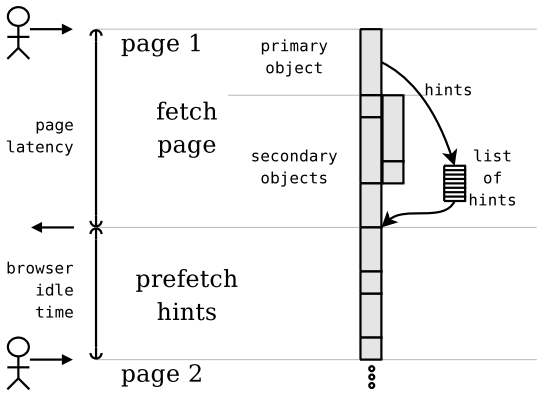



Figure 2.1: Evolution in time of a user request with prefetching

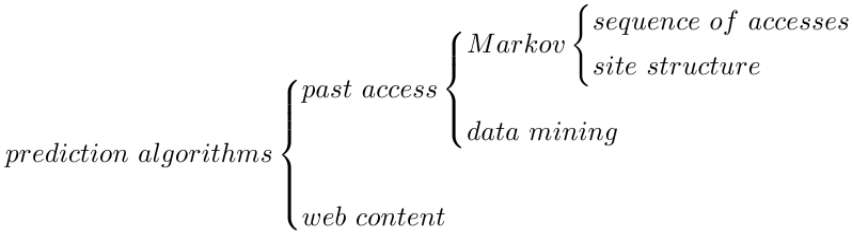



Figure 2.2: Taxonomy of prediction algorithms

Although a requested web page can be the result of a dynamic request, many of the objects that compose the page are usually static and, consequently, cacheable. Moreover, a dynamically generated object can be cached if it is properly labeled. As all the cacheable objects can be prefetched (precached), this study covers, among other technologies, dynamic web server and application programming, and browser application programming like AJAX, Java, Flash, and other Rich Internet Applications.

## 2.3 Web Prediction Algorithms

Prediction algorithms can be classified in two main groups according to the type of information used to make predictions [Domènech 06e] (see Figure 2.2).

The first group includes algorithms that predict future accesses based on the previous access patterns. Two subgroups can be distinguished: one that consists of algorithms that use Markov models [Padmanabhan 96, Palpanas 99, Domènech 06a, Zhu 02], and the other with algorithms that use data mining techniques [Yang 03, Gündüz 03, Nanopoulos 03]. Many prediction algorithms based on Markov models

can be found in the literature, and some of them provide high precision predictions but at the expense of intensive computation and memory consumption. The resource consumption of data mining based algorithms is even higher.

The second group contains the algorithms that analyze the web content to make predictions. Some authors propose to combine the analysis of the content with usage profiles [Duchamp 99], others apply neural networks to keywords extracted from HTML content [Ibrahim 00], and some others detect similarities in context words around links in the HTML content [Davison 02]. The proposals [Ibrahim 00, Davison 02] are based on the object popularity and the associated hyperlinks, but they do not consider the relationship among objects.

Regarding the prediction algorithms based on Markov models, one of the most widely used in the literature is the Dependency Graph (DG) proposed by Padmanabhan and Mogul [Padmanabhan 96]. DG builds a dependency graph that depicts the pattern of accesses to the objects. There is a node in the graph for each object that has ever been accessed. Other well-known algorithm is the Prediction by Partial Match (PPM), proposed by Palpanas and Mendelzon [Palpanas 99], which also uses a Markov model to store the context of accesses. Both algorithms achieve a high precision in the predictions, but unfortunately this fact is not directly related to high latency savings for web users because the algorithms do not consider the structure of current websites [Domènech 06a].

The Double Dependency Graph (DDG) algorithm, proposed by Domènech *et al.* [Domènech 06a], is based on DG but it considers the structure of current websites by differentiating between pages and embedded objects. DDG provides useful predictions to effectively reduce the users perceived latencywhen downloading web pages.

The prediction algorithms mentioned above try to learn the user patterns from the sequence of accesses. These algorithms consider that two objects are related if they are requested by the same user closely in time.

Other Markov algorithms learn user patterns from the site structure. Zukerman *et al.* [Zukerman 99] compare different prediction models. Some consider the order in which documents are requested, and others the structure of the server site. But the study does not present any algorithm in detail and any result about the actual page latency savings. Zhu *et al.* [Zhu 02] propose to build a Markov model from web log files and use it to make predictions. This work mainly concentrates on the compression of the transition probability matrix. It does not present any algorithm in detail, and does not study the performance of the prediction algorithm.

Using current prediction algorithms in the real world presents some problems with the resource consumption. Several works have addressed this topic in order to reduce the complexity of the prediction models and, consequently, their computational and memory costs. In this sense, Deshpande *et al.* [Deshpande 04] focus on pruning Markov models of web prediction algorithms. They present several techniques to combine Markov models of different order to reduce the state-space complexity while maintaining the prediction accuracy. They also propose three schemes for pruning states: states with low frequency of occurrence; states with low confidence on state

outgoing transition; and states with high error associated with a state (determined by using off-line verification with a trace). They conclude that the Markov models after pruning achieve similar or better accuracy than the original models.

## 2.4 Performance Evaluation

Many research efforts related to web prefetching focus on how to improve theoretical indexes of the prediction algorithm like precision and recall [Dongshan 02, Nanopoulos 03]. Nevertheless, there are few works dealing with the reduction of users perceived latency, traffic, and web server load increase that evaluate and compare their proposals [Domènech 06c].

Moreover, few research works have been addressed to compare the performance between different prediction algorithms mainly because of the difficulty in reproducing environments and workloads [Domènech 06b]. Two algorithms based on Markov models, proposed by Zukerman [Zukerman 99] and by Bestavros [Bestavros 95], are compared in [Albrecht 99]. The comparison is only performed at the algorithmic level, without considering details related to the users perceived latency. Another work [Bouras 04] compares two algorithms, one based on the idea of popular objects of Markatos [Markatos 98] and the other based on a variation of the Prediction by Partial Matching algorithm. These comparisons were made from the point of view of the prediction and its precision, and to the knowledge of the author of this dissertation, there is only a fair attempt to compare them from the user's perspective [Domènech 06d].

The first relevant study that analyzed the potential benefits of web prefetching was carried out by Kroeger *et al.* [Kroeger 97]. They analyzed the limits on latency savings that caching and prefetching reached in several scenarios. Experiments were performed with a perfect off-line prediction algorithm and using traces obtained from a proxy server in 1996. An interesting conclusion of that research work was that prefetching doubles the latency reduction achieved by caching, which is limited by the frequent update of objects in the web. However, their results are difficult to compare due to the assumptions made about the environment and the workload characteristics. In 1999, Li Fan *et al.* [Fan 99] investigated, using traces from 1996, the limits on the benefits that a perfect prediction algorithm located at the proxy server could achieve.

Bouras *et al.* [Bouras 04] stated that the results obtained in web prefetching experiments strongly depend on the architecture or model assumptions, and that the parameters that influence web prefetching are closely related to the web architecture itself.

Domènech *et al.* [Domènech 06e] studied the impact of the web architecture on the limits of latency reduction. They identified that the main constraint to obtain the upper bound in latency savings is due to the location in which a user access cannot be predicted, e.g., the first access of a session and the first time the predictor sees an object. Their results showed that latency reduction clearly depends on the location of the predictor. They stated that latency can be reduced by 36%, 54%, and 67%

when the predictor is located at the server side, client, or proxy, respectively. Latency reductions higher than 90% could be obtained if the predictor worked collaboratively at different elements of the architecture.

Finally, Balamash *et al.* [Balamash 07] proposed a mathematical model for a web prefetching architecture. Their results showed that prefetching was profitable even with the presence of a good caching system.

To sum up, the studies discussed above are quite theoretical either because they use analytical models or because prefetching is not applied under real conditions and scenarios. For instance, most of them assume that all the predicted objects will be prefetched, which is far from real implementations because the prefetching is only performed during the browser idle time, and only if the object is not already in the browser cache. Another common problem is that most studies only model a part of the web architecture, the prediction engine, and do not take into account the web client and the prefetching engine. Furthermore, in the performance evaluation they take into account the performance of the prediction engine, but not the results of prefetching observed by the final user.

## 2.5 Software Implementations

This section reviews the previous attempts to implement web prefetching techniques. These include both prototype implementations proposed in the research literature, and commercial products available for production usage. The software products are grouped in three categories: servers, proxies and clients, according to the element of the architecture where the prediction or prefetching engines are located, as summarized in Table 2.1.

Kokku *et al.* [Kokku 03] propose NPS, a system to perform non-interfering web prefetching. The system monitors the network state and adapts the parameters of the prediction and prefetching system to prevent saturation. It does not require modifications either in the web browser or in the HTTP protocol since it includes specific JavaScript code in the served pages to perform the actual prefetching. It does not provide hints using HTTP standard headers, which is possible nowadays. The learning process is done only in an initial step.

The results provided by Google search sometimes include the first page of the list as a hint embedded in the HTML code. If the web browser is capable of prefetching, it may request that page in advance.

Domènech *et al.* [Domènech 04a] propose a free available framework for prefetching. It is a hybrid implementation that combines both real and simulated parts in order to provide flexibility and accuracy. It implements state-of-the-art prediction algorithms to produce hints on the emulated web server. It also emulates web clients that prefetch the objects and provides several performance results like precision, recall and response time. This framework is very useful to test prediction and prefetching algorithms, but it is not designed for a real world usage.

Table 2.1: Software with prediction or prefetching capabilities

| Type | Name | | Description |
|---|---|---|---|
| Server | NPS | E | Non-interfering Prefetching System (2003) |
| | Google Search | C | Hints embedded on HTML search results |
| | JosepDom | E | Benchmarking framework (2004) |
| Proxy | Wcol | E | Prefetches all links (1997) |
| | Squid-prefetch | E | Prefetches all links (small Perl script) (2004-2008) |
| | AllegroSurf | C | Prefetches all links (2004-today) |
| | Paketeer SkyX Accel. | C | Prefetches links (?-2007) |
| | Robtex Viking Server | C | Prefetches links (1996-today) |
| Client | Mozilla | C | Browser with prefetching capabilities (2002-today) |
| | Google Web Accel. | C | Prefetches all links in HTML (2005-2008) |
| | FasterFox | C | Prefetches all links in HTML (2005-2006) |
| | PeakJet 2000 | C | Prefetches all or visited links (1998) |
| | NetAccelerator | C | Prefetches all links (1998-2005) |
| | Personalized Mozilla | E | Predicts and prefetches based on history (2003) |

E: experimental implementations

C: commercial or productive implementations

There are several web proxies with prediction and prefetching capabilities. Some of them (Wcol, Squid-prefetch and AllegroSurf) prefetch all the hyperlinks of a html document, thus wasting bandwidth unnecessarily. This massive and indiscriminate prefetching can be problematic, and has been criticized by system administrators, web designers and users [Dornfest 05]. No information about the prediction algorithms used in the other proxies is available, which leads to consider they use a similar method. Packeteer SkyX Accelerator was a gateway designed to accelerate connections in the local network using an undisclosed prefetching method (it was discontinued in 2007). Viking Server is a commercial product for Microsoft Windows operating systems that is supposed to include a proxy with prefetching capabilities.

There are several products that provide prefetching capability to the end-user web client, but all of them use the same method as the proxies: prefetching all hyperlinks. In this case, not only the web servers' bandwidth is wasted unnecessarily, but also the client's one. The only exception is Mozilla-based products, because they prefetch only the hints provided by the web server during idle time.

Mozilla Firefox is a web browser with web prefetching capacity. Other web browsers based on the same Mozilla Foundation technologies include this capacity, for example SeaMonkey Netscape, Camino, and Epiphany. Web prefetching was first available in Mozilla Suite 1.2 (published at the end of 2002).

Google Web Accelerator [Google 05] was a free web browser extension available for Mozilla Firefox and Microsoft Internet Explorer on Microsoft Windows operating systems between 2005 and 2008. It includes, among other features, web prefetching. It prefetches hints included in the HTML body, but also prefetches all the links in the pages that are being visited, even if no hints are provided.

FasterFox is an open and free extension for Mozilla web browsers presented in 2005 that prefetches all the hyperlinks found in the current page during the browser idle time.

PeakJet was a commercial product for the end user, available around 1998, that included several tools to improve the user access to the web. It included a web browser independent cache with prefetching capability, based either on history or on links. It therefore could prefetch links on the current web page that were visited by the user at some time in the past, or all the links on the current web page.

Another commercial product for the end user that prefetched all the links in the page that are being visited, and stored the objects in the browser cache was NetAccelerator. It was commercialized between years 1998 and 2005, and included the possibility of refreshing the cache content in order to avoid obsolete objects.

Wei Zhang *et al.* [Zhang 03] present the design and implementation of a modified Mozilla web browser with prediction capability that includes two prediction algorithms. The main one is based on history and uses the Prediction by Partial Matching algorithm (PPM) [Palpanas 99]. If this one provides few hints, another algorithm based on the page content is additionally used.

In summary, most prediction or prefetching implementations are proprietary or do not even attempt to implement a smart prediction algorithm. The remaining implementations are not ready for usage in a real environment, or do not take into account both the web server and the web client overload.

# Chapter 3

# Delfos. Architecture Prototype with Prefetching Support

## 3.1 Introduction

Despite the research efforts focused on web prediction and prefetching techniques, the field is short of studies dealing with the implementation and use of these techniques in real environments, as mentioned above. The first partial objective we proposed was to develop a prototype to demonstrate that web prefetching can be implemented to be used usage in real-world conditions.

This chapter presents Delfos, the framework developed to perform web predictions and prefetching in a real environment that tries to file the existing gap between research and praxis. Delfos is the Spanish name of Delphi, the famous oracle of Apollo perched on the sides of mount Parnassos where ancient Greeks went to know the future. Delfos is integrated in the web architecture without modifying the standard HTTP 1.1 protocol, which makes it suitable for being used with current browsers, web servers and protocols. Delfos performs web predictions at the web server side, and provides the hints in HTTP headers. Prefetches are carried out by the web clients in a compatible way with current commercial browsers.

This chapter is organized as follows: Section 3.2 outlines the framework architecture. Section 3.5 describes how the web client works. Section 3.3 presents the prediction server. Section 3.4 describes the web server. Section 3.7 shows the results of the experiments performed in a real environment. Finally, Section 3.8 presents the concluding remarks of this development.

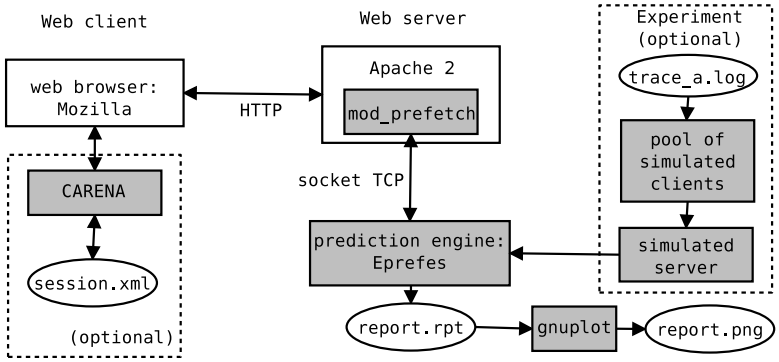



Figure 3.1: Framework architecture

## 3.2 Framework Architecture

As mentioned above, *Delfos* is a framework to perform prefetching in a real system. Because of its flexibility, it can also be used to develop, test and evaluate prefetching techniques. And these capabilities will be explained in the next chapter. The current version of *Delfos* is integrated with Apache 2 web server and Mozilla web browser, although any web server or web client is suitable for working with *Delfos*.

Fig. 3.1 depicts the framework architecture. It comprises three main parts: the prediction engine, the web server, and the web client. The prediction engine (*Eprefes*) performs predictions and provides hints to the web server. The web server (Apache 2) includes a module (*Mod-prefetch*) to query predictions and provide them to the web client. The web client includes a web browser with prefetching support (Mozilla) and a tool to capture and replay web navigation sessions, which is explained in detail in the Appendix A (*CARENA*, [Niño 05]). Below we detail how these parts work.

## 3.3 Prediction Engine: *Eprefes*

*Eprefes* is a prediction engine designed to be used in a real environment. It runs a prediction algorithm, gathers statistics and listens for TCP connections. When *Eprefes* receives a prediction request, it executes the prediction algorithm and returns the resulting hints. This process has a minor impact on the response time, being currently around 1 millisecond.

To verify the *Eprefes* accuracy, experiments were run both on it and on the simulator proposed by Domènech *et al.* [Domènech 04a], obtaining negligible deviations.

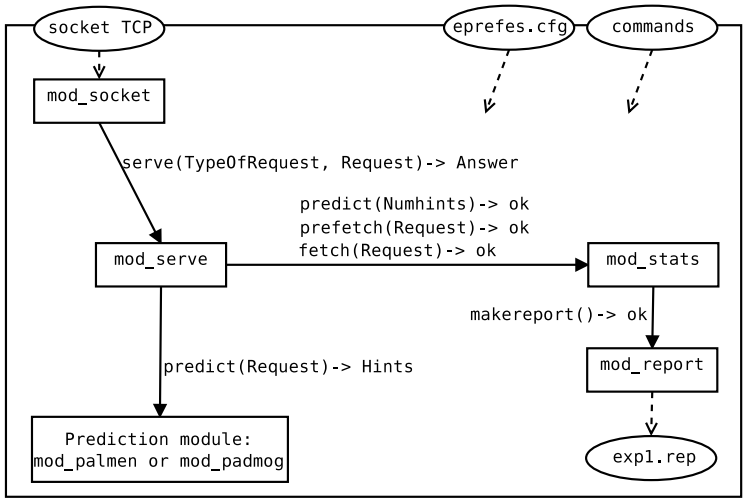



Figure 3.2: Architecture of the *Eprefes* prediction engine

### 3.3.1 Features

The main features of *Eprefes* are: it is independent of the web server; it can be controlled externally, it is modular, different parameters of the modules can be reconfigured dynamically and the code can be modified, compiled and reloaded at runtime without restarting neither the entire engine nor any module. Let us discuss them in more detail.

*Eprefes* is independent of the web server that queries it. The communication between both is by means of a TCP socket. This design provides several advantages. The prediction engine can be used with different models of web server. It is only required to write a module for the web server that connects, queries and adds the hints to the HTTP headers. The prediction engine and the web server can be implemented on different languages. The web server and *Eprefes* can be located in the same or in different machines, which sometimes is preferable due to security, stability or efficiency reasons. A single prediction engine may be capable of serving several web servers, and it is not required to install it in all of them.

All the functionalities available in *Eprefes* are distributed in different modules. Table 3.1 gives a general view of the available modules and their purpose. Fig. 3.2 shows the server architecture, the relation among the different modules, the called functions and parameters, and the results returned by them. The optional elements used for training and performance evaluation experimentation are explained in detail in the Section 4.3.

Most modules have configurable parameters, for example, the maximum number of hints that can be provided as a response to a prediction request. They can be set

Table 3.1: Modules in the *Eprefes* prediction engine

| Function | Module | Description |
|---|---|---|
| Connectivity | *mod-socket* | Listens for TCP connections |
| Serving requests | *mod-serve* | Manages requests depending on the request type |
| | *mod-trainer* | Optional. Reads log files and trains the predictor |
| Statistics | *mod-stats* | Calculates statistics and performance indexes |
| | *mod-report* | Generates reports periodically |
| Prediction | *mod-palmen* | Makes predictions using the Palpanas and Meldelzon's algorithm (PPM) |
| | *mod-padmog* | Makes predictions using the Padmanabhan and Mogul's algorithm (DG) |
| | *mod-ddg* | Makes predictions using the Domènech *et al.* algorithm (DDG) |

in the configuration file before start up, or modified at runtime by other modules, i.e., a new module that allows to modify such parameters using a web interface or shell commands, which can be very useful for adaptive policies.

*Eprefes* is entirely written in Erlang/OTP [Armstrong 07, Armstrong 93], which allows code swapping among other features. Runtime code swapping allows to add new functionalities, improve performance, or fix bugs on the source code and reload the newly compiled modules into memory without restarting the server or missing the internal data.

### 3.3.2 Connectivity

The module *mod-socket* provides connectivity by using a TCP connection and binary format messages. When started, this module opens a socket to listen for TCP connections in the configured port number. Once a connection is established, it creates a process that waits for requests. Each request will be parsed and submitted to the *mod-serve* module. The response is conveniently packaged and sent back throughout the TCP connection. The messages received include the client IP address, timestamp, and object URI, MIME type and file size. The format of the accepted message is:

```
{Action, {ClientIP, Timestamp, URI, MIME, Size, ResponseCode}}.
```

where Action can be `predict`, `prefetch` or `fetch`. The response message is simply a list of hints. The format of the final message sent is:

```
Options"Hint1"Hint2"...HintI"...HintN"end"
%
```

where Options can be `Require-validation` or empty and HintI is a string ready to be included in the HTTP response headers.

An alternative module *mod-xmlrpc*was also developed. This allowed communications using XML-RPC instead of TCP connections. But it was not used since it consumes more CPU time than *mod-socket*.

### 3.3.3 Serving requests

The module *mod-serve* manages each received request depending on the message type, which can be a prediction, a prefetching or a fetching request. If the message is a prediction request, it is redirected to the prediction module that will answer with none, one or several hints. Finally, the hints are sent back to the calling module. If the calling module is *mod-socket*, the hints are returned to the web server, which returns the hints to the web client for prefetching them as described in the following sections of this chapter. Besides, the hints are also notified to the statistics module for evaluation purposes.

### 3.3.4 Prediction algorithms

In a first step and for performance evaluation purposes, we have implemented several prediction algorithms widely referred to in the literature, although any prediction algorithm using web server information can be implemented in *Eprefes*.

The algorithms are: Prediction by Partial Matching (PPM) proposed by Palpanas and Mendelzon [Palpanas 99], Dependency Graph (DG) proposed by Padmanabhan and Mogul [Padmanabhan 96], and Double Dependency Graph (DDG) proposed by Domènech *et al.* [Domènech 06a].

These algorithms learn dynamically with each prediction request, so a special training phase is not required, and this information will be updated with subsequent changes in the web objects, web structure or users' patterns. The prediction algorithms include parameters to limit the growth of the data structures.

#### 3.3.4.1 *mod-palmen*

The module *mod-palmen* implements the prediction algorithm proposed by Palpanas and Mendelzon [Palpanas 99], which is based on Prediction by Partial Matching (PPM). This algorithm uses Markov models to store the context of accesses, and performs predictions by comparing the current context to each Markov model.

This algorithm builds a tree as long as it receives requests, and keeps a list of contexts for each navigation session. To provide hints to a client it takes into account both the tree and the mentioned client's navigation session.

A prediction algorithm of order M keeps a maximum of M+1 elements in the context lists and the maximum tree depth is also M+1. The context list of order M for a given navigation session contains a maximum of M+1 node identifiers. The node identifier in position I in the context list refers to a node of depth I on the tree, and this node is called order I context for this navigation session. The context of order 0 for all navigation sessions is the root node of the tree.

When a request is received, a new child node is added to all the nodes in the context list, or the occurrence is updated if the child node already exists. This node has two attributes: the URI of the requested object and occurrence. Therefore,

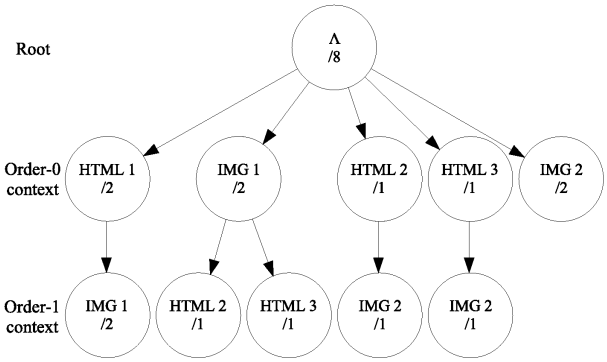



Figure 3.3: Example of a graph generated by the PPM algorithm

for each depth of the tree, one and only one node is added or updated. Those node identifiers recently added or updated are the new contexts for that navigation session.

After a period of inactivity, the navigation session is declared obsolete and the list of context is deleted.

When the list of contexts is updated, it points to the new recently visited nodes. The parent nodes correspond to the ones visited immediately before. This way, the branch that currently has node I in the context list is the concatenation of the last I accesses carried out in this navigation session. If a given context node has children nodes that were added by other navigation sessions in the past, those children are potential new accesses and are candidates to be reported as hints. The probability of a hint is calculated as the division of the potential hint probability and the parent node occurrence. It is possible to give more probability to hints from higher contexts since they come from longer branches where more steps match and hence the probability of its future appearance is higher.

As an example, Figure 3.3 illustrates the graph built by PPM given a hypothetical sequence of requests. The sequence consists of two user sessions: the first requests the objects HTML1, IMG1, HTML2, IMG2; and the second requests HTML1, IMG1, HTML3, IMG2. Note that IMG2 is an object linked both by HTML2 and by HTML3. Each node represents a context, where the root node is in the first row, the order-0 context is in the second, and the order-1 context is in the third. The label of each node also includes the counter of times a context has appeared, so one can obtain the confidence of a transition by dividing the counter of a node by the counter of its parent, i.e., the node in the previous context. The arcs indicate the possible transitions. For instance, the label of the IMG2 in order-0 context is 2 because IMG2 appeared twice in the training; once after HTML2 and another after HTML3; IMG2 has two nodes in the order-1 context, i.e., one per each HTML on which it depends.

The configurable parameters for this module are the maximum order of the tree (*maxo*), the minimum order in which hints will be reported (*mino*), the probability

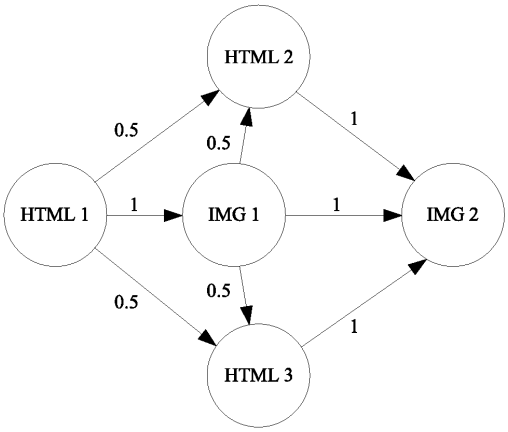



Figure 3.4: Example of a graph generated by the DG algorithm

threshold required to report a hint (*threshold*), the threshold reduction in subsequent tree orders (*thred*), the maximum number of hints that can be reported in a request (*mnh*) and the duration of a navigation session (*sessiondur*).

#### 3.3.4.2 *mod-padmog*

This module implements the prediction algorithm Dependency Graph (DG) described by Padmanabhan and Mogul [Padmanabhan 96]. It is based on a Markov model and considers that two objects are more related when they are requested more frequently one after the other in a window containing the last accesses of that client.

The algorithm builds a dependency graph that represents the access patterns of the objects. This graph keeps a node for each single object that has ever been accessed. There is an arc from node A to node B, if and only if, at any time node B was accessed after node A in a timeframe not longer than W accesses. W is the lookahead window size. The weight of an arc is the relation between the occurrence of the arc that goes from A to B and the occurrence of A.

Each navigation session is associated with a window of the last W accesses. When a new request arrives, a node is included in the graph with the corresponding URI. In addition, arcs linking that node with the other nodes in the window of last accesses are updated.

Given an access to the object A, the URIs of all the nodes that receive an arc from that node will be reported as hints, being the weight of each arc considered as the probability that this node will be requested.

As an example of this, Figure 3.4 illustrates the graph built by DG when configured a lookahead window of 2 accesses, and given the same hypothetical sequence of requests previously mentioned for the PPM algorithm (section 3.3.4.1). A node in

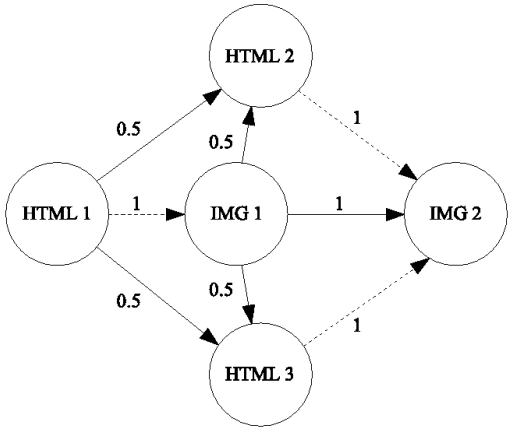



Figure 3.5: Example of a graph generated by the DDG algorithm

the graph represents a web object, and the weight of an arc indicates the confidence level of the transition from the arc's predecessor node to the arc's successor node.

The configurable parameters in the algorithm are: the size of the last accesses window (*sizew*), the threshold applied to hints' probability to be included on the prediction response (*threshold*), and the duration of the navigation session (*sessiondur*).

#### 3.3.4.3 *mod-ddg*

The module *mod-ddg* implements the Double Dependency Graph (DDG) prediction algorithm proposed by Domènech *et al.* [Domènech 06a].

The DDG prediction algorithm is based on a graph that keeps track of the dependences among the objects accessed by the user. It distinguishes two classes of dependences: to an object of the same page and to an object of other page. Like DG, the graph has a node for every object that has ever been accessed. There is an arc from node A to B, if and only if, at some point in the time a client accessed to B within W accesses to A, where W is the lookahead window size. The arc is a primary arc if A and B are objects of different pages, that is, either B is an HTML object or the user accessed one HTML object between A and B. If there are no HTML accesses between A and B, the arc is secondary.

The predictions are obtained by firstly applying a cutoff threshold to the weight of the primary arcs that leave from the node of the last user's access. In order to predict the embedded objects of the following page, a secondary threshold is applied to the secondary arcs that leave from the nodes of the objects predicted in the first step.

The DDG algorithm has the same order of complexity as DG, since it builds a similar graph but distinguishing two types of arcs.

As an example of this, Figure 3.5 depicts the graph built by DDG when configured a lookahead window of 2 accesses, and given the hypothetical sequence of requests mentioned previously on page 18. In the figure, primary arcs are displayed with solid lines, and secondary arcs with dashed lines.

## 3.4 Web Server: *Mod-prefetch* for Apache 2

*Mod-prefetch* is a module developed for the Apache 2 web server that permits this web server to act as a predicting server. This module requests hints to the prediction engine and submits them in the HTTP response headers to the web browser. See section 3.3.2 for more details.

First, when the user clicks on a URI, the web browser requests the object to the web server. When the web server receives the request, *Mod-prefetch* establishes a TCP socket connection to the prediction engine and sends a message to it depending on the HTTP request: if it is a standard GET request, *Mod-prefetch* sends a *predict* message request. Then the prediction engine performs a prediction based on the URI of the requested object. The result of the prediction is a list of hints, which are added to the HTTP response as HTTP response headers, as described in HTTP/1.1 [Fielding 97] (e.g.: `Link: /news/190309; rel:prefetch`). The HTTP response is sent to the web browser.

This dissertation assumes that the prefetching engine, located at the browser, prefetches, during the browser idle time, the hints received as mentioned previously, and this is how Mozilla-based browsers work [Fisher 03b]. Once prefetched, the web browser stores the objects in its local cache. In this way, if the user demands any of these objects later, they will be served without any network latency. Only those objects that can be stored at the web client can be predicted and prefetched, and it is important to use prefetching in well-designed websites where GET requests only trigger idempotent actions.

## 3.5 Web Client: Web Prefetching in Mozilla

Mozilla Firefox is a web browser with web prefetching capabilities. Web prefetching was available for the first time in Mozilla Suite 1.2 (published at the end of 2002). Other web browsers based on the same Mozilla Foundation technologies include this capability, e. g., SeaMonkey Netscape, Camino, and Epiphany.

We use Mozilla Firefox in our work since it already implements all the required features regarding prefetching, that is: it is widely used by both casual and expert users, it is published with a free and open source license and its source code is freely available.

Mozilla is able to prefetch hints if they are included in the response HTTP headers or embedded on the HTML file [Fisher 03a]. This prefetching mechanism was

first proposed by Padmanabhan and Mogul [Padmanabhan 96], and standardized in HTTP/1.1 RFC 2068 [Fielding 97].

The hints can be provided in three different ways:

- in a response HTTP header:

```
Link: <ch3.html>; rel=prefetch
```

- in a 'meta' tag on the HTML header:

```
<meta HTTP-EQUIV="Link"
CONTENT="<ch3.html>; rel=prefetch">
```

- in a 'link' tag on the HTML body:

```
<link rel="prefetch" href="ch3.html">
```

The implementation of web prefetching in Mozilla features some interesting aspects that are outlined below. Only the provided URIs using the HTTP protocol are prefetched, without embedded objects. URIs that contain parameters (the query part of the URI) will not be prefetched. Prefetching will only occur when the web browser is idle. Web requests sent by Mozilla when prefetching include an additional HTTP request header in prefetching requests: `X-moz:prefetch`. Mozilla does not require those prefetching requests to be responded, so web servers can filter them, for example, in case of overload conditions. Hints are only prefetched when the object that includes the hints is demanded by the user. If the user clicks on a link while the browser is prefetching, the prefetch process is interrupted to satisfy the user's real request. If there is any prefetching queue, it is discarded. The object partially downloaded will be kept on cache and completed if the user demands it. Later, when the browser is idle again, new hints can be prefetched.

A web page consists of a main object (the one demanded by the user) referred to as primary and many embedded objects called secondary objects. When the user demands a web page identified by its URI, the web browser firstly requests the primary object of the page. Once this primary object is received, the browser processes it to get the secondary objects from the network or from the local cache. Although the requested web page can be the result of a dynamic request, many of the objects that compose the page are usually static and, consequently, cacheable. Moreover, a dynamically generated object can be cached if it is properly labeled. As all the cacheable objects can be prefetched (precached), this study covers, among other technologies, dynamic web server and application programming, and browser application programming like AJAX, Java, Flash, and other Rich Internet Applications. Two HTTP 1.1 persistent connections with no pipelining are assumed, since that is the maximum number of connections that the standard recommends [Fielding 97]. So, the secondary objects are requested using two independent parallel network threads.

When all the secondary objects are available in the browser, the download ends, since the whole page has arrived to the client. Our experimental environment, based on the mentioned architecture, assumes that secondary objects are retrieved once the primary object of the page is completely received. This assumption overestimates the page latency in all cases, but it is made in all the experiments both with and without prefetching, so its comparative impact on the page lattency saving is negligible.

The hints received by the web browser in an object response are added to the *hints queue* of the page that contains that object. When the browser is idle (i.e., it is not downloading any object), it prefetches the objects referenced in the hint queue. This structure is handled as a FIFO queue, thus, it is important that the prediction engine provides the hints sorted according to the associated probability. In this way, if only a subset of the provided hints is going to be prefetched, these hints will be the most *useful* ones. Remark that if a hint is already in the browser cache, it will not be prefetched.

The prefetching process is canceled and its hints queue is flushed if the user demands another page while the browser is prefetching. In such a case, the prefetch process is aborted to satisfy the user's current demand. When a prefetch cancellation occurs, those objects that have been partially predownloaded (i.e., a part of them has already been received and stored in cache) are allowed to reside in the cache. Those parts can be reused if the objects are demanded later.

## 3.6 Interrelation Between the Components

As mentioned above, the web prefetching architecture consists of in two main components: a predictor engine that we locate in the web server, and a prefetching engine that we locate in the web client.

Fig. 3.6 shows the communication between the web browser, the web server with the module *Mod-prefetch* and the prediction engine. In this example, the user first demands the object A, and consequently the web client sends an HTTP request to the web server. Then, the server requests a prediction to the prediction engine given the access to A. The prediction engine predicts that the user will demand objects B and H in the near future. The web server returns the requested object A with two HTTP response headers indicating the predicted hints. The web browser prefetches the object B in its idle time, and stores it in the browser cache. If the user later demands the object B, the web browser will show it to the user with zero service time because the object is in the browser cache.

## 3.7 Experiments in real environment

We ran two experiments to verify that *Delfos* is ready for real usage. The first one analyzed how the number of objects in the browser cache increases due to the prefetch actions performed, and the second one dealt with page latency saving achieved due to

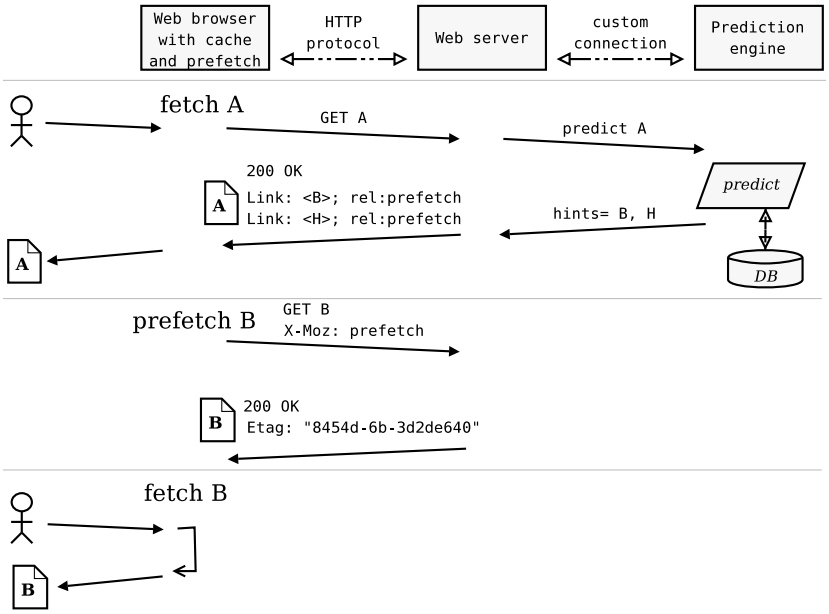



Figure 3.6: Communication between the web browser, the web server and the prediction engine when using web prediction and prefetching

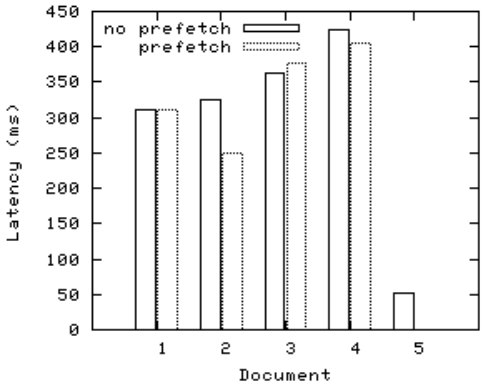



Figure 3.7: Latency of documents in a navigation session

prefetch. To accomplish this, the Mozilla web browser navigated a web server where we previously inserted the prediction module, and this module requested predictions to the prediction engine. The documents were simple HTML files with one or two embedded image files. The prediction engine had conveniently been trained before. The PPM prediction algorithm was used in this test.

A specifically developed tool (*CARENA*, see Appendix A of this dissertation) was launched on the web browser in order to capture the navigation session, including object headers and accurate document latencies as perceived by the user. With this information it is possible to replay exactly the same navigation session several times.

For the first experiment, a navigation session consisting of 14 documents was captured. If prefetching is not used, 60 objects are requested, while if prefetching is enabled, that number rises to 67. The number of hits in the browser cache when using prefetching increases from 39 to 45. That means that 6 of the 7 (i.e., $67 - 60$) prefetched objects were later required by the user, which results in a precision of 85.7%. In summary, this small and limited experiment illustrates how prefetching is able to reduce the users perceived latency in a transparent way.

Results of the second experiment are presented in Fig. 3.7. In this case, the navigation session consisted of five document requests. It was repeated twice, the first time without prefetching and the second time with prefetching enabled. The client and the server were in different networks so the network latency was not negligible.

The prediction engine provides twenty hints, but only five of them are prefetched since the other ones were already in cache or being requested. The prefetchs are requested during idle time, so they do not increase the document latency as perceived by the user. Only two of the five prefetched objects are later requested by the user. A prefetch from the first document was a hit on the second document, which explains the 70 ms latency saving. A prefetch from the fourth document was a hit on the fifth document, which explains the 50 ms latency saving.

## 3.8 Conclusions

This chapter presented *Delfos*, a framework that provides web prefetching capabilities in real environments. To the knowledge of the author, it is the first implementation for real usage that features smart prediction algorithms and provides hints by using the method described on HTTP 1.1.

The prediction engine is an independent program that connects to the web server to provide hints, and a module for Apache 2 is available for this purpose. Mozilla web browser is used since it already includes the required support for prefetching. An important novelty of the proposed framework is that it does not require any modification in the standard HTTP 1.1 protocol.

Partial results of the work presented in this chapter were published in [de la Ossa 07a, de la Ossa 07c, de la Ossa 06].

# Chapter 4

# Delfos: Evaluation Environment

## 4.1 Introduction

Delfos is not only a prototype of web prefetching in a real scenario, as described in the previous chapter; but also an evaluation environment, as is described in this chapter. In this way, Delfos can be used as a flexible framework to evaluate and compare existing prefetching techniques and algorithms, and to assist in the design of new ones because it provides detailed statistics reports. Those statistics reports also permit to evaluate the performance of either the prediction engine, the prefetching engine or both, thus contributing to the design of new and more efficient algorithms and structures.

This chapter is organized as follows. Section 4.2 describes CARENA, a Mozilla extension to capture and replay navigation sessions. Section 4.3 describes the additional modules developed to support the new usage. Section 4.4 explains the evaluation methodology used in the remaining experiments of this dissertation. Section 4.5 shows an example of evaluation, including performance indexes and system statistics. Finally, Section 4.6 summarizes the conclusions.

## 4.2 *CARENA*

*CARENA* is a Mozilla extension to capture and replay user navigation sessions. *CARENA* captures information about the user session, which can be used later to replay or mimic the gathered user navigation. *CARENA* emulates the original user think times as these times are very important to obtain precise and reliable performance results. *CARENA* is a multiplatform, open source, lightweight, standards

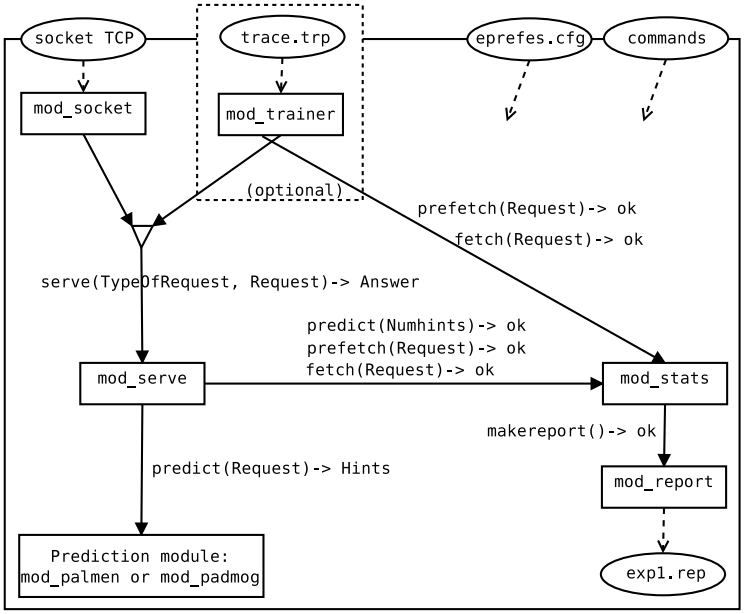



Figure 4.1: *Eprefes* architecture for experimentation

based, easily installable and usable application, programmed in JavaScript and XUL. We use it to test the correct behaviour of *Delfos*.

Details about the implementation and use of CARENA can be consulted in Appendix A.

## 4.3 Modules for Evaluation

This section describes the modules implemented in *Delfos* to perform automated experiments in the prediction engine and to obtain statistical results. These modules were already mentioned in Table 3.1, and are included in the *Eprefes* extended architecture, as shown in Figure 4.1.

### 4.3.1 Client Pool with *mod-trainer*

The *mod-trainer* module emulates web clients behaviour by reading a trace file and by generating load to the prediction engine, thus making it suitable for training the prediction algorithm or for performing controlled experiments.

The trace file uses a customized Common Log Format or Combined Log Format: a set of time-ordered lines, being one line for each HTTP request received by the web

server. The information of each line includes the web client IP address, the object URI, the timestamp when the HTTP response is sent, and the size of the requested object. Trace files are slightly customized before being read by the module. Additionally, trace files are filtered to select the appropriate HTTP method (i.e., *GET*) and the HTTP response code (i.e., *200 OK*, *304 Not Modified* and *206 Partial Content*).

The module reads the trace file sequentially and sends object fetch requests to the prediction module. When hints are received, the module sends prefetch requests in the browser idle time as indicated in the trace file. If the legitimate user, later in the trace, requests an object that was virtually prefetched, the module sends a fictitious fetch request.

This module is useful to gather statistics when using web server log files as input for the prediction engine instead of real web clients with prefetching capability.

### 4.3.2 Statistics gathering with *mod-stats* and *mod-report*

The *mod-stats* module maintains variables and calculates performance indexes that can be used for comparison purposes, e.g., to evaluate the prediction accuracy and usefulness or the resources consumed by the prediction algorithm. These data are calculated and written to disk periodically without stopping the process, so all statistics are available immediately.

Some statistics available are: the received requests, fetched objects, hints sent to the web server, objects that were prefetched, prefetchs that were later fetched (prefetch hits), hints that were later proved right (good predictions). All these variables are measured both in number and byte size.

This module also calculates different performance indexes, which are described in the next section.

For all the performance indexes, the mean value and the confidence interval are calculated. Performance indexes are measured using two methods. The standard one, called *EXP*, measures the values from the beginning of the measurement session. The method called *INT* calculates the indexes using only information of the last measurement interval. Proceeding in this way, the evolution of the performance indexes is shown without the interference of very old values.

All performance indexes are obtained with a confidence interval of 95%. Nevertheless, for the sake of clarity, only average values are shown in the figures, as the interval lengths are always lower than 15% the average value for latency related indexes, and lower than 5% for recall related indexes. In order to calculate average values with confidence intervals, each experiment is splitted in shorter successive runs. The run length for each single experiment is 120K-user requests and each run consists of 20 intervals equally sized. On the other hand, for each single experiment a preliminary warming up phase of 70 intervals (420,000 object requests) that use the initial part of the trace was carried out before collecting statistics. This represents 7.43% and 12.31% of the total requests of Trace A and Trace B, respectively. The remaining part of the trace was used to continue the experiment and obtain the results.

Finally, the *mod-report* module generates periodic statistic reports provided by the statistics module and writes them to data files for later usage by specific tools such as *Gnuplot*.

## 4.4 Evaluation

This section describes the evaluation used to carry out the experiments in this chapter and the following ones. We are going to use the cost-benefit evaluation methodology described by Domènech *et al.* in [Domènech 06d].

### 4.4.1 Performance indexes

The most useful performance indexes for evaluating web prefetching techniques were identified and described in [Domènech 04b]. According to that study, in this dissertation we use the indexes described below.

Most research works found in the literature use the precision and recall as the main and only performance indexes. They can be measured per object or per byte.

The precision measures the ratio of objects that were predicted, prefetched and then finally requested by the user (prefetch hits) versus the total number of objects that were predicted and prefetched. The precision per byte, or byte precision, is calculated using the objects size.

$$Pc = \frac{\text{Prefetch hits}}{\text{Prefetchs}} \tag{4.1}$$

$$Pc_B = \frac{\text{Size of Prefetch hits}}{\text{Size of Prefetchs}} \tag{4.2}$$

The recall measures the ratio of user requested objects that were previously predicted and prefetched.

$$Rc = \frac{\text{Prefetch hits}}{\text{User requests}} \tag{4.3}$$

$$Rc_B = \frac{\text{Size of Prefetch hits}}{\text{Size of User requests}} \tag{4.4}$$

The object latency is obtained from the service time reported by the web server, or it is zero if the object is already in the browser cache. Thus, the object latency saving is the ratio of the latency perceived using prefetching to the latency without prefetching.

$$\nabla OL = \frac{\text{Average Object Latency with Prefetch}}{\text{Average Object Latency no Prefetch}} \tag{4.5}$$

The page latency is obtained by performing an experiment without prefetching, and these values are used as a baseline for comparison purposes in the experiments.

The page latency saving ($\nabla PL$) is calculated as:

$$\nabla PL = \frac{\text{Average Page Latency with Prefetch}}{\text{Average Page Latency no Prefetch}} \tag{4.6}$$

Similarly, the page latency saving percentage ($\nabla PL(\%)$) is calculated as:

$$\nabla PL(\%) = (1 - \frac{\text{Average Page Latency with Prefetch}}{\text{Average Page Latency no Prefetch}}) * 100 \tag{4.7}$$

This work uses the page latency saving as the main performance index for measuring prediction and prefetching effectiveness because our aim is to study the maximum benefit perceived by web users.

The traffic increase quantifies, in bytes, the extra traffic incurred by the prefetched objects that are never requested by the user. We do not take into account the network overhead introduced by the transmission of hints on HTTP headers, as its size can be assumed negligible when compared to the objects size. We show the traffic increase as a ratio of the traffic generated with and without prefetching enabled.

$$\Delta Tr_B = \frac{\text{Objects not used}_B + \text{Network overhead}_B + \text{User requests}_B}{\text{User requests}_B} \tag{4.8}$$

The object traffic increase quantifies the percentage in which the number of objects that a client gets increases when using prefetching, compared to not using prefetching.

$$\Delta Tr_{ob} = \frac{\text{Objects not used} + \text{User requests}}{\text{User requests}} \tag{4.9}$$

### 4.4.2 Workload Description

The prediction engine could be fed by a real web server that receives real requests from real users. However, in order to compare the performance with different configurations, a reproducible workload must be used.

For this purpose, the experiments were performed using two web traces (A and B) from different websites. The initial trace files were logged by Apache 2 web servers in a custom format that included, among other request information, the referrer, the user agent, and the object latency. But not all the captured HTTP requests were or could be used by real prediction algorithms and prefetching engines, so they were filtered by request type and response code: the request protocol is HTTP (neither HTTPS nor any other); the request method is GET; the request URL does not contain any query string; and the response code is 200, 206 or 304. The requests that meet this filtering criteria are considered cacheable for the purpose of this work. Table 4.1 summarizes the main characteristics of these traces.

Table 4.1: Trace characteristics

| **Characteristic** | **Trace A** | **Trace B** |
|---|---|---|
| Starting date | Sept, 27th 2007 | Mar, 21st 2005 |
| Ending date | Jun, 18th 2008 | Nov, 22nd 2006 |
| Unique IP addresses | 131,668 | 48,283 |
| Browsing sessions | 317,268 | 271,736 |
| Page requests | 992,037 | 1,159,191 |
| Avg. page latency (seconds) | $0.877 \pm 0.117$ | $2.075 \pm 0.378$ |
| Unique objects | 6,790 | 1,330 |
| Object requests | 5,654,371 | 3,411,307 |
| Requests of objects smaller than 10 kB | 77% | 70% |
| Bytes transferred (MB) | 28,135 | 42,910 |

Trace A was obtained from a dynamic website that acts as the home page of an open source project, which mainly contains news posts, documentation, and forums. Most of the content is generated dynamically by PHP scripts that query a database. However, the dynamic URIs are persistent, since they are written using an Apache 2 feature that does not use URI query strings. The site is visited by worldwide users using a wide variety of web browsers.

Trace B is from the website of the School of Computer Science at the Universidad Politécnica de Valencia. This site mainly contains news posts, and information addressed at students, staff, and visitors. Unlike the previous website, the content of this site is not dynamically generated and its visitor community is much more constrained.

A web session is a group of page requests made by the same web browser (identified by its IP address). The trace file does not indicate when a session finishes, so we assume that a browser idle time longer than 15 minutes represents the end of the web session.

Since it is not possible to know the browser idle time of the last page in a session, in the experiments we assume it to be 30 seconds long.

The experiments do not include a preliminary training phase. Instead, the prediction algorithm constantly learns the user's patterns during the experiments. This guarantees that the knowledge of the prediction algorithm about user patterns is updated at the time that the patterns change [Domènech 05]. The length of the experiments is long enough so the experiments do not end in a transitional phase.

## 4.5 Performance Evaluation Using Delfos

The purpose of the experiments presented in this section is to show how *Delfos* can implement prefetching techniques and how it permits to evaluate the performance obtained.

To allow fair comparisons, the configuration of *Delfos* was the same in the different experiments. Common configuration options were: maximum of 100 hints allowed in a HTTP response, interval length of 100000 user requests and subinterval length of 5000. Regarding *mod-palmen* (PPM) specific options: threshold 0.2, maximum order 1, minimum order 1 (see section 3.3.4 for references). And *mod-padmog* (DG) specific options: lookahead window size 1. A previous work [Domènech 06d] demonstrates that those values provide relatively good cost-benefit ratio.

### 4.5.1 Experiments

In this section we show how *Delfos* can be used for performance evaluation of prefetching techniques using trace-driven experiments. The PPM prediction algorithm was used in the first experiment. It was configured to produce reasonably good results. Our prediction engine can be fed by a real web server that receives real requests from real users. However, in order to compare the performance of prediction algorithms with different configurations, a reproducible workload must be used.

In the remaining experiments the prediction engine receives requests from a special trainer program that reads preprocessed web server logs. The module *mod-trainer* (described on Section 4.3.1) is enabled to generate prefetchs based on the predictions and hits based on the real user requests logged. These results were obtained using the particular behaviour of *mod-trainer*, therefore they are an upper bound of the results expected in real world conditions. An experiment with five million user requests takes around ten hours to complete in a standard PC (Intel Pentium 4 3.4 GHz, 1 GB of RAM).

The length of the experiments is measured in processed user requests. Trace B is used to perform those experiments. In order to allow the observation of all the learning process, no previous training phase is carried out prior to the experiment.

Fig. 4.2 shows the evolution of the precision and the recall, both of them measured per object and per byte. Decreasing the prediction algorithm threshold increases the prediction (and hence the prefetching) aggressiveness, which also increases the cost measured in bandwidth usage, and also the benefit on latency savings. Since no training phase was used, the confidence intervals are considerably large at the beginning, but decrease slowly and consistently over the experiment. As this figure shows, the possibility of seeing not only average values but also confidence intervals helps to detect transitional phases.

Fig. 4.3 shows the evolution of recall and recall per byte through time. In addition to the standard accumulated indexes shown before (labeled *EXP*), this figure includes the *INT* indexes that only consider the values of the last interval to calculate the

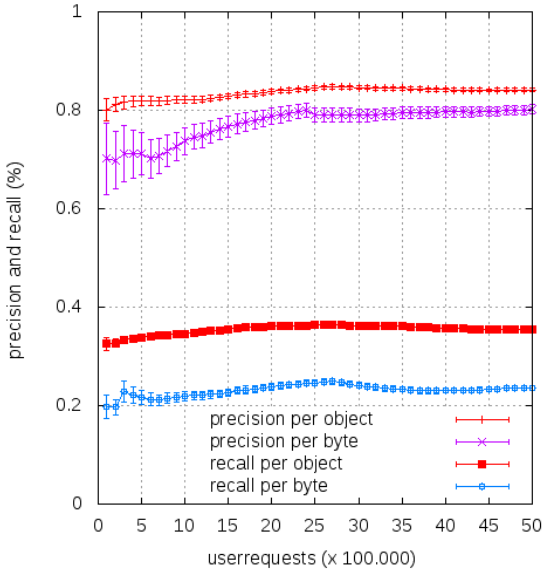



Figure 4.2: PPM: precision and precision per byte, recall and recall per byte

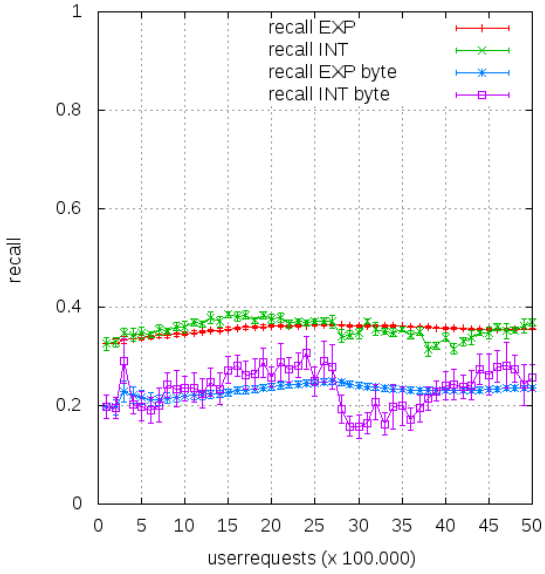



Figure 4.3: PPM: recall and recall per byte, both of them *EXP* and *INT*

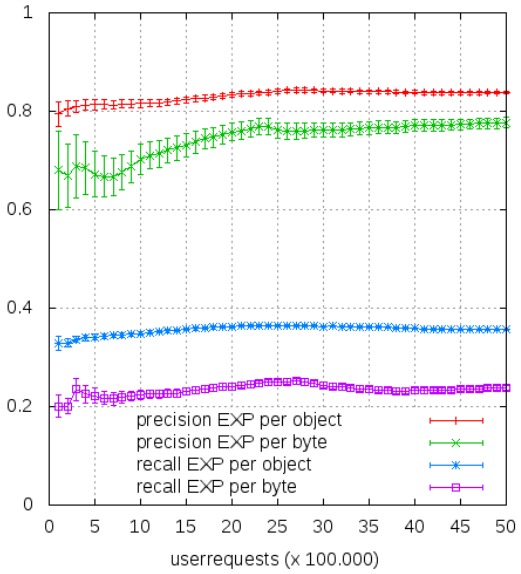



Figure 4.4: DG: precision and precision per byte, recall and recall per byte

indexes. Each interval includes 100,000 user requests. As expected, the indexes that do not consider old values (*INT*) are more variable than the indexes that consider all the values from the beginning of the experiment (*EXP*). This is clearly observed around 2.7 million user requests after the start, when the trace used in our experiments produces an important reduction on recall indexes. Those unexpected variances in indexes are common and reasonable when using real traces instead of synthetically generated ones.

Another experiment was run using the DG prediction algorithm. Fig. 4.4 shows the precision and recall indexes obtained in this experiment. Since the same environment with similar characteristics was used to run this and the previous experiment, the results can be compared side by side to detect differences in the performance indexes results due to the prediction algorithm. For example, both algorithms achieve almost identical precision indexes. With respect to recall indexes, PPM achieves almost identical mean values but slightly smaller confidence intervals.

Other performance indexes measure the latency saving and bandwidth consumption. Fig. 4.5 depicts the object latency saving as a mean value and the confidence interval. On the other hand, Fig. 4.6 depicts the object traffic increase. Comparing the results of these figures, the PPM prediction algorithm configured in this experiment required 20% of object traffic increase to provide 10% of latency saving per object.

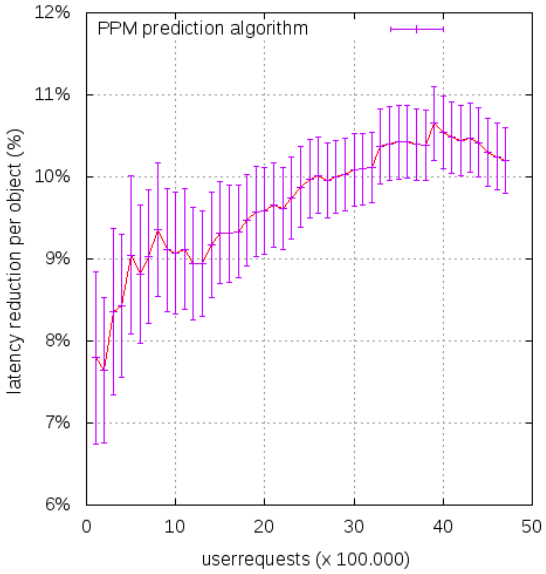



Figure 4.5: PPM: Object latency saving

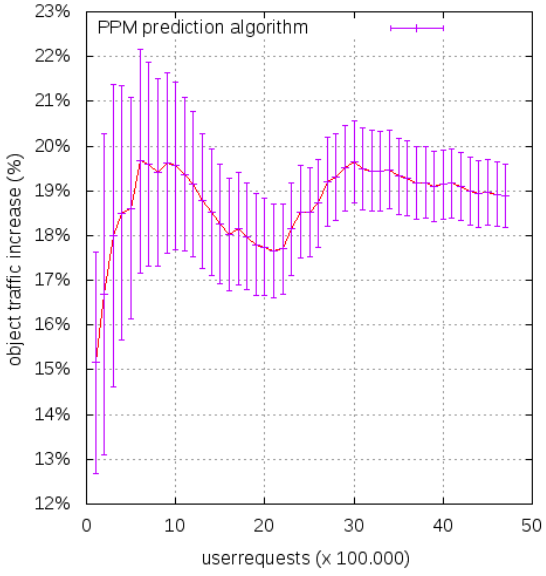



Figure 4.6: PPM: Object traffic increase

### 4.5.2 System statistics

In addition to the performance indexes, *Delfos* allows the modules that implement prediction algorithms to report statistics that may be interesting in each case, for example, those related to data structures: number of registers in a database table, nodes and arcs in a graph, total memory consumption, etc.

Some statistics are equally defined for all prediction algorithms and hence can be used to compare how the algorithms operate. For example, Fig. 4.7(a) shows the total memory consumption of data structures in the experiment using PPM, and Fig. 4.7(b) shows the same but using DG. Memory consumption is important in real world implementations, since an algorithm providing great precision and recall may not be suitable for real world conditions if it has high memory or computation requirements.

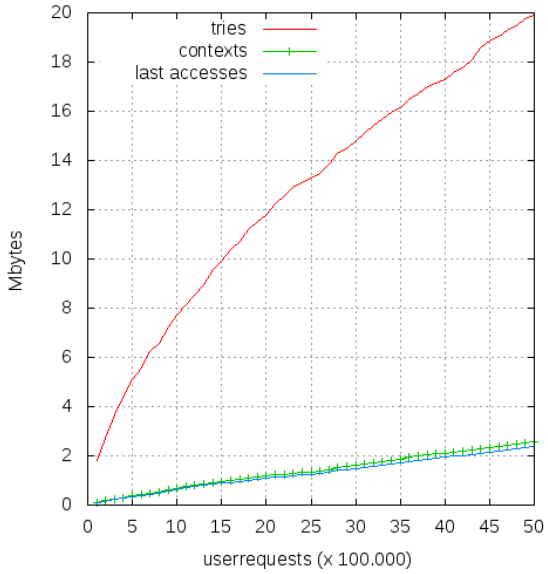
Fig. 4.9 shows how the service time required by the prediction algorithms increases as the experiment progresses. This figure clearly shows the learning process of the prediction algorithms and how the increase in their data structures has a negative effect on the resulting service time. Please note that the values obtained depend on the hardware used to run the experiments and the particular implementations of the prediction algorithms.

Other general system statistics are illustrated in Fig. 4.8. Database operations (Fig. 4.8(a)) provide an approximate number of operations performed in the database. An approximation to the CPU consumption is depicted in Fig. 4.8(b), since each reduction (term related to functional programming) involves a function call.

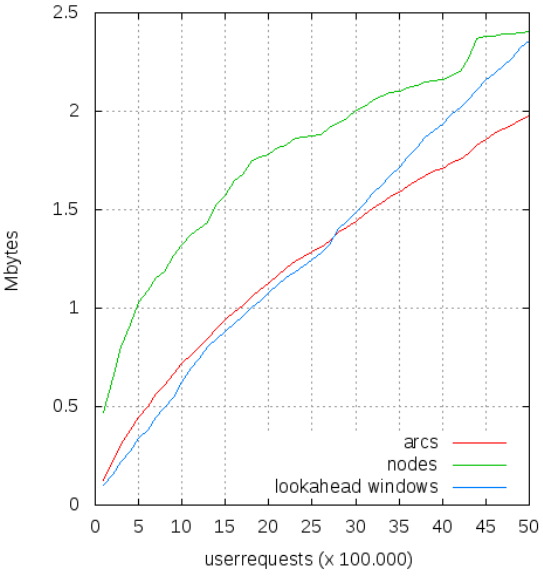
Other statistics are specific to the used algorithm, but even if they cannot be used to compare different algorithms, they are interesting to observe how different configuration and workloads affect the algorithm performance. Example of statistics on a tree-based data structure as used by *mod-palmen* (PPM) are: the mean number of children (Fig. 4.10(a)), and the number of nodes of order 0 and 1 (Fig. 4.10(b)).

Fig. 4.11 shows examples of statistics on a graph-based data structure, in this case the one used by *mod-padmog* (DG): total number of nodes and arcs, mean nodes occurrence, mean arcs occurrence, and mean arcs probability.

*Delfos* can be used to discover new insights into prefetching thanks to the detailed statistics. As an example, let us briefly observe the relation between performance indexes and resource consumption (memory and CPU). The figures show that data structures are still growing when the experiments end. Instead, performance indexes like precision and recall were mostly invariant during the last part of the experiments, when using both *mod-palmen* and *mod-padmog*. This means that the prediction algorithm did not improve performance indexes after an initial learning phase. Allowing unlimited learning and size of data structures did not improve precision or recall, but data structures grew, making the algorithm slower.

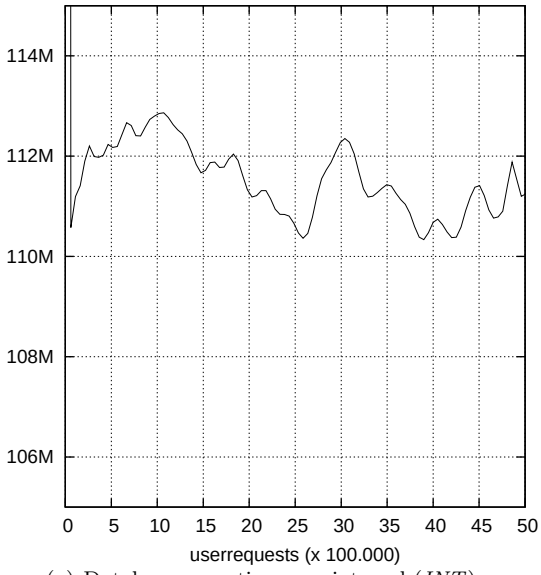



(a) *mod-palmen* (PPM)

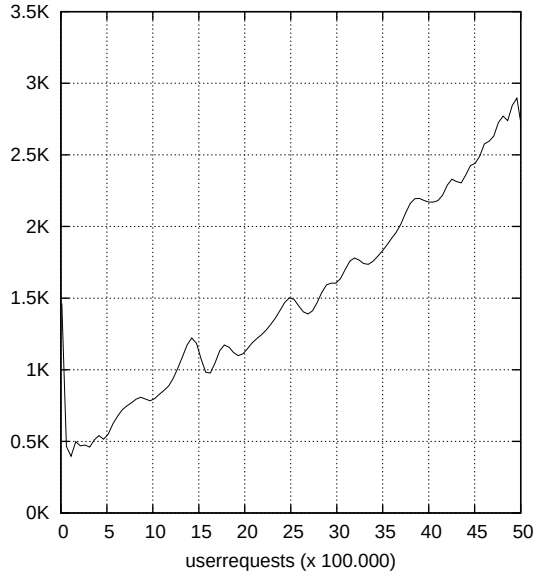



(b) *mod-padmog* (DG)

Figure 4.7: Memory consumed by data structures




(a) Database operations per interval ($INT$)




(b) Reductions per interval ($INT$)

Figure 4.8: System statistics

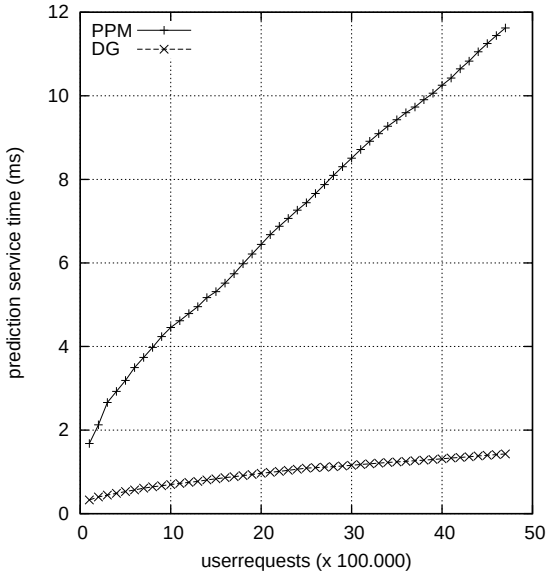



Figure 4.9: Prediction service time (in milliseconds)

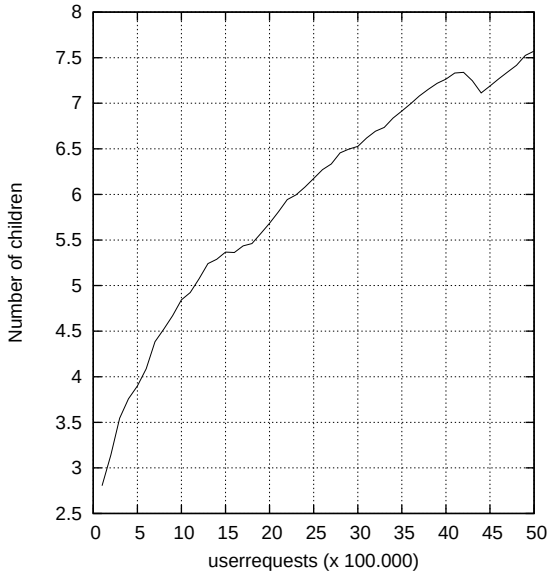
## 4.6 Conclusions

This chapter described the enhancements added to *Delfos* to make it not only a web prefetching framework suitable for real environments, but also a flexible tool that can be used either for research purposes or performance evaluation analysis.

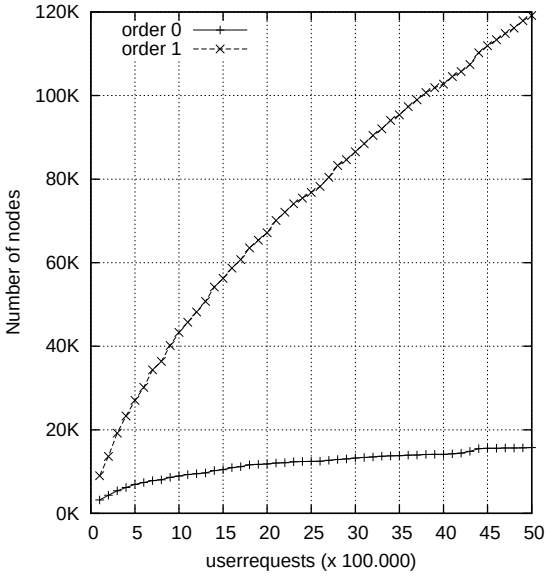
To this end, *Delfos* provides detailed statistic reports and allows easy implementation and replacement of prediction algorithms, since they are isolated on independent modules in the prediction engine. Statistics include both performance indexes like precision and recall (both per byte and per object) and resource utilization.

The possibility of seeing the confidence interval in a graph permits us to clearly see if the measured index converges to a value at the end of the experiment, and to detect which prediction algorithms converge faster. The possibility of seeing not only the index value accumulated during the experiment, but also the instantaneous index value in a short period of time, permits us to see the general behaviour of a prediction algorithm, and the sporadic changes localized in time. Finally, having resource consumption indexes allows to evaluate and compare the algorithms benefits as well as the cost.

Some results of the work presented in this chapter were published in [de la Ossa 07a, de la Ossa 07c, de la Ossa 06, Niño 05, de la Ossa 04].

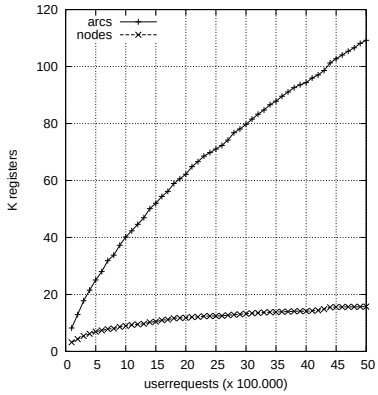



(a) Mean number of children

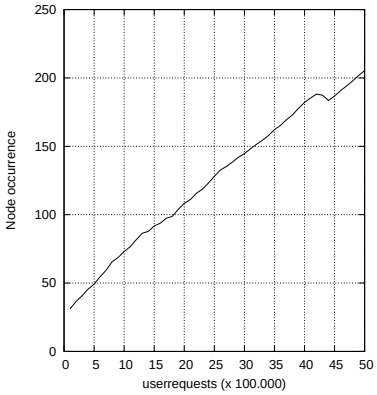



(b) Number of nodes of order 0, 1

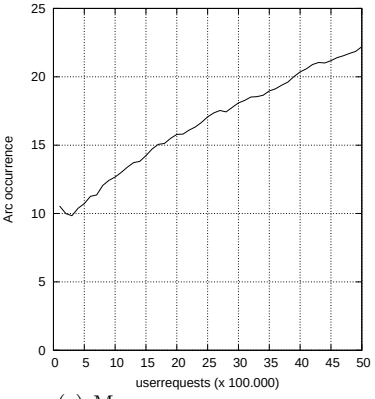
Figure 4.10: Statistics provided by *mod-palmen* (PPM)



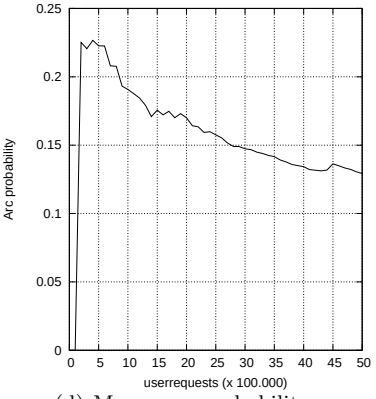

(a) Total number of nodes and arcs




(b) Mean nodes occurrence




(c) Mean arcs occurrence




(d) Mean arcs probability

Figure 4.11: Statistics provided by *mod-padmog* (DG)

# Chapter 5

# Predict at Prefetch: a Technique to Improve Prefetching

## 5.1 Introduction

This chapter proposes Predict at Prefetch (P@P), a technique for the prediction engine to improve web prefetching performance. A conventional prediction technique can be extended to include the P@P proposal in real world conditions without changes in the web architecture or HTTP protocol. To show how this proposal can improve prefetching performance, an extensive performance evaluation study has been carried out and the results show that P@P can considerably reduce the users perceived latency with no additional cost over the basic prefetch mechanism.

The Predict at Prefetch (P@P) technique allows the prediction algorithm located at the web server to provide hints not only in standard object requests, but also in prefetching requests. That is, this technique allows the prediction engine to provide more hints to the client. In this sense, this proposal is orthogonal to the prediction algorithm, that is, it and can be used with any prediction algorithm without any modification.

The technique Predict at Prefetch has been implemented on *Delfos* and tested in real world conditions. An important feature of the technique is that it does not require changes in the web architecture, the HTTP standard protocol or the web browser.

The remainder of this chapter is organized as follows. Section 5.2 describes Predict at Prefetch. Then, Section 5.3 details the experiments and results. Finally, Section 5.4 presents some concluding remarks.

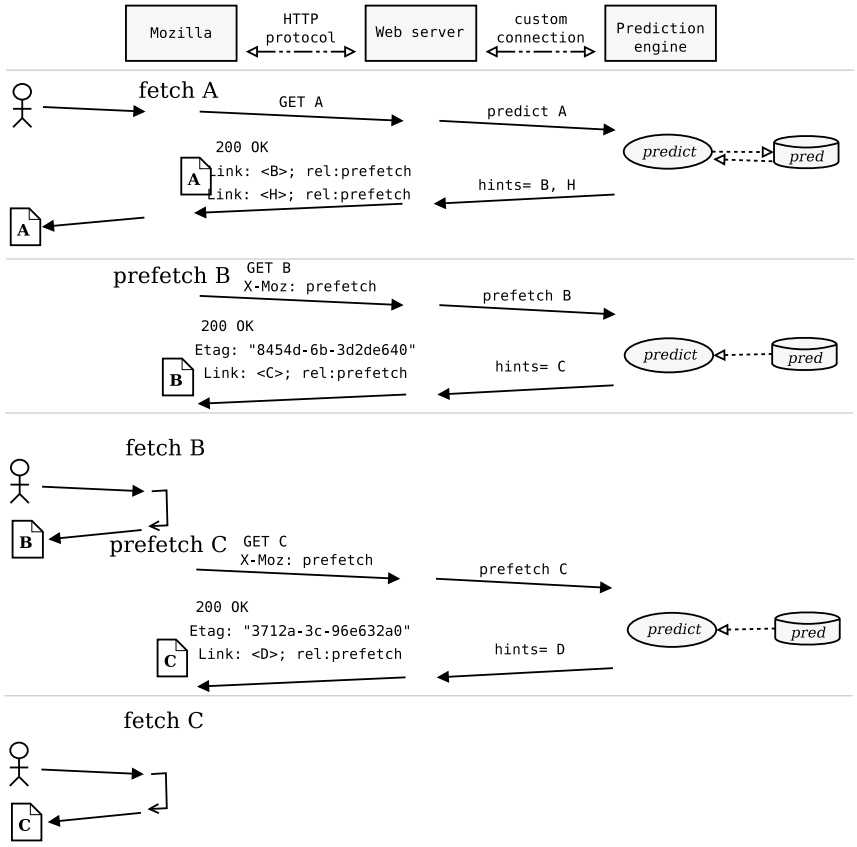



Figure 5.1: Communication with Predict at Prefetch enabled

## 5.2 Predict at Prefetch (P@P)

The proposed technique to improve prefetching performance, Predict at Prefetch (P@P), is a simple and effective technique that allows web clients to receive more hints without negatively affecting the precision of the prediction algorithm.

Predictions are provided not only for objects requested by demand, but also for prefetched objects. Any web request received by the web server, either a fetch or a prefetch request, triggers a prediction and the resulting hints are included in the corresponding response.

Notice that when the P@P technique is used, more hints are reported to the web browsers than when using the basic prediction. Therefore, more objects are expected to be prefetched. As a consequence, the traffic will increase, but if the prediction algorithm is accurate enough, the users perceived latency will be reduced. By properly

configuring the aggressiveness of the prediction algorithm, both for fetch and prefetch requests, it is possible to reduce the latency in a higher ratio than the traffic increase.

Fig. 5.1 shows an example of communication among the web browser, the web server and the prediction engine when working web prediction, prefetching and Predict at Prefetchtogether. First, the user demands the object A. After this request, the prediction engine predicts that the user will demand objects B and H in the near future. The web browser, while idle, prefetches object B. The response to that prefetch request includes another hint, this time for object C. However, the web browser does not prefetch that hint, since it was provided in a prefetch request. If the user later demands object B, as the browser already has it on its cache, it will be provided to the user with zero service time. Now the prefetched object B is considered as an object demanded by the user, so the hints included in its response can be prefetched: object C is finally prefetched.

The hints provided to the browser in a prefetch request are prefetched only if the prefetched object is finally requested by the user. This ensures that those hints are as accurate as the hints provided together with objects requested by demand.

The predictions performed during a prefetch request should not update the information gathered by the prediction algorithm about the user's behaviour and navigation patterns. The reason is that prefetched objects are not requested by the user, but by the web browser based on a prediction that might or might not be successful. If the prediction algorithm wrongly assumed that a prefetch request is equivalent to a user request, it would produce an unrealistic learning of user's navigation patterns.

Notice that no modification on the web browser is required, but the web server that provides hints must be updated to conveniently handle the prefetch requests. Any existing prediction algorithm can be used with P@P without any modification.

## 5.3 Experimental results

Trace-driven experiments were performed to show the impact of the proposed technique on the users perceived latency and what traffic increase is required to accomplish it. To this end, several experiments were run both enabling and disabling our proposal, and varying the threshold of the prediction algorithms, because it affects the prefetching aggressiveness.

Two prediction algorithms were used in the experiments: the DG algorithm proposed by Padmanabhan and Mogul [Padmanabhan 96], which has been widely referenced in the literature; and the DDG algorithm proposed by Domènech *et al.* [Domènech 06a], which is an improvement over DG, as it provides better performance with similar cost. The evaluation methodology used to carry out the experiments was described in detail in the previous chapter, as the characteristics of the trace B that is used as workload.

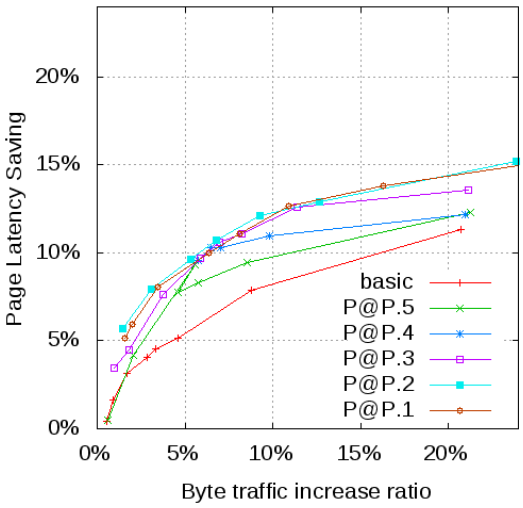



Figure 5.2: Page latency saving versus byte traffic increase with the DDG prediction algorithm

### 5.3.1 Cost-Benefit

Fig. 5.2 shows the page latency saving versus the byte traffic increase results for two different prediction methods. The basic method, which performs predictions only for user requests, as described in Section 3.6, and the P@P method which performs predictions also in prefetch requests as described in Section 5.2. In both cases the prediction algorithm used is DDG. The curves shown in the figures have been obtained by varying the algorithm threshold (aggressiveness) from 0.1 to 0.8.

As it is known and also showed by our results, there are some configurations that allow the basic prefetching technique to reduce the page latency in a higher percentage than the traffic increase. So, prefetch is an interesting technique to reduce the users perceived latency if there is available bandwidth and the prediction algorithm is properly configured for the target workload.

When using the P@P method, the threshold for those additional predictions can be set independently of the standard prediction requests. In the experiments, P@P is evaluated with different thresholds ranging from 0.5 to 0.1 (P@P.5 down to P@P.1). The objective of this study is not to find the optimal configuration of the P@P technique, because it strongly depends on the environment conditions (bandwidth, server load, traffic,...), but to demonstrate that we can always find a P@P configuration that outperforms the results of the basic prefetching technique.

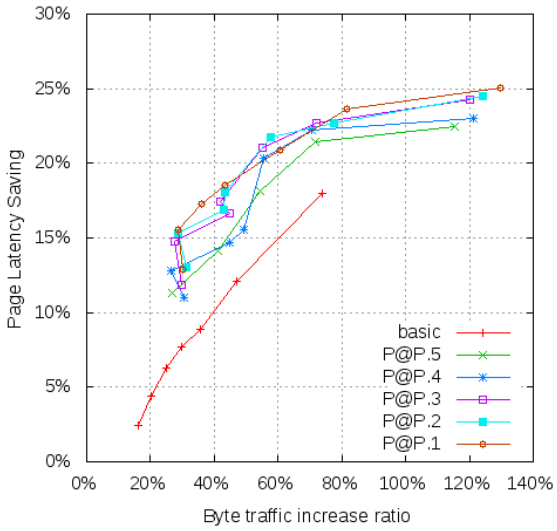



Figure 5.3: Page latency saving versus byte traffic increase with the DG prediction algorithm

As observed, the prediction algorithm aggressiveness has a two-side effect on the results, as it achieves a reduction on the users perceived latency but at the expense of a higher byte traffic increase. The P@P configurations for a given byte traffic increase can achieve higher page latency savings than the basic prefetching technique. That means that for the same or similar latency saving obtained using the basic prefetch mechanism, there is always a P@P configuration that requires less byte traffic and obtains a similar benefit. P@P.1 and P@P.2 are particularly interesting, since they provide the best cost-benefit ratio. Moreover, in some situations, P@P.2 provides a latency saving up to 12% while increasing the byte traffic by about 10%.

Fig. 5.3 depicts the latency saving versus byte traffic increase obtained when using DG. Notice that this algorithm is much more aggressive than DDG, as it generates by about five times more byte traffic. As observed, the results show that the use of the Predict at Prefetch technique provides a similar benefit on both algorithms. That is, the maximum distance between the best P@P curve and the basic prefetch, measured in absolute values, is similar when using the DDG (see Fig. 5.2) and the DG algorithms (see Fig. 5.3). Nevertheless, the DG algorithm is less suitable for working with bandwidth restrictions because, for a reasonable latency saving, it generates 20% more traffic than DDG.

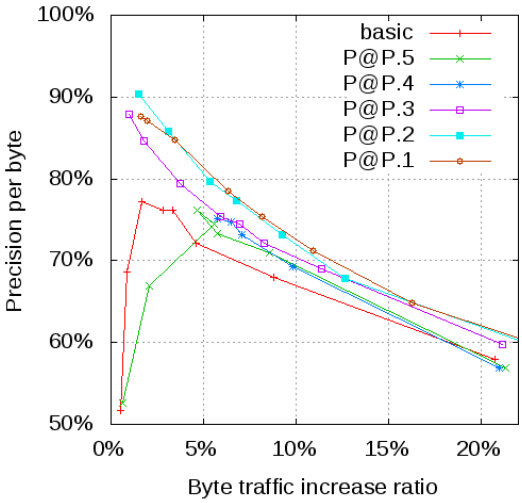



Figure 5.4: Precision per byte versus byte traffic increase with the DDG prediction algorithm

### 5.3.2 Prediction related performance indexes

In this section we show the impact of our technique on the prediction-related performance indexes, i.e., from the prediction algorithm point of view.

The experiments performed using both prediction algorithms show similar behaviour. Nevertheless, as showed in the previous section, the DDG algorithm is more efficient than DG because it requires less traffic increase to achieve the same latency saving, so we only present the results for the DDG algorithm.

Fig. 5.4 shows how the prediction method affects the precision per byte of the prefetched objects versus the byte traffic increase. The precision is reduced as the prediction algorithm becomes more aggressive, which is the usual behavior of the prediction algorithms. In general, the precision obtained with Predict at Prefetch is better than the obtained with basic prefetching because the prediction algorithm, which is quite accurate, has more opportunities to provide hints and alleviate the precision reduction caused by sporadic wrong predictions. This fact becomes more noticeable with the most aggressive configurations. In these cases, the recall per byte also increases, as shown in Fig. 5.5. This plot has similar shape to the one in Fig. 5.2 because the represented indexes are directly proportional (recall and latency saving).

The P@P technique generally gets better cost-benefit ratio than the basic prefetching, being the configuration called P@P.1 the one providing the best value.

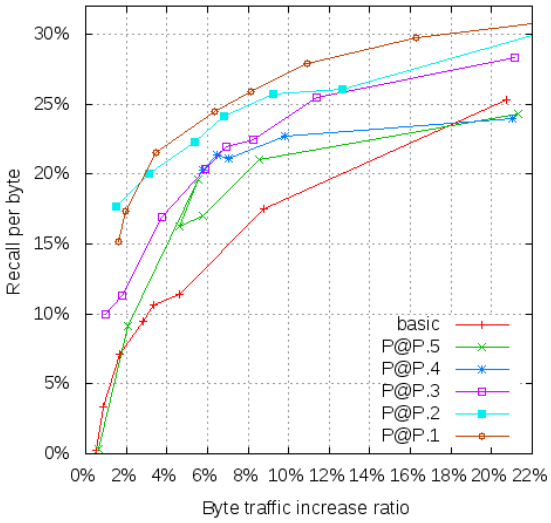



Figure 5.5: Recall per byte versus byte traffic increase with the DDG prediction algorithm

The maximum latency saving and recall achievable by our proposal can be observed in Fig. 5.2 and 5.5, respectively. Neither the latency saving nor the recall can be improved beyond a certain threshold (about 35% and 20%) when allowing a reasonable traffic increase (about 22%).

### 5.3.3 Algorithm Storage Usage

Fig. 5.6 gives an overview of the amount of information stored by the prediction algorithm with different configurations. Each individual object requested to the web server is represented by a node on the prediction algorithm data structure, and each time a client requests an object, its node occurrence is increased. When the prediction algorithm is more aggressive, it provides more hints, and consequently the clients prefetch more objects. That means less objects requested to the web server by the client on behalf of a direct user demand. The prediction algorithm learns the user patterns from user requests, not from prefetch requests, since these are speculative. When using more aggressive configurations the algorithm gathers less information, and this fact becomes a problem if the prediction algorithm receives so few requests that it is unable to appropriately learn user patterns. Experimental results obtained for a wide range of threshold values show that an excessive aggressiveness largely reduces

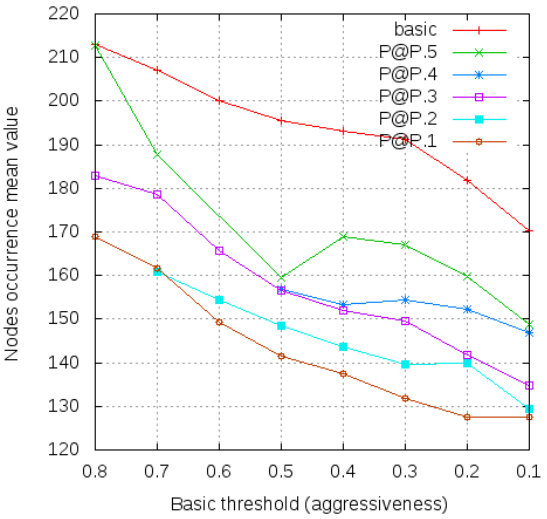



Figure 5.6: Mean nodes occurrence versus algorithm threshold (aggressiveness) with the DDG prediction algorithm

the precision and does not improve the recall and users perceived latency, thus it is not necessary to implement aggressive policies to improve prefetching performance.

Fig. 5.7 depicts the amount of database operations performed by the prediction algorithm during the experiment. As it is directly proportional to the algorithm complexity, it gives us an idea of the CPU consumption required by the different configurations. Obviously, P@P requires more database operations, since the prediction algorithm makes more predictions. However, each prediction requires a similar processor time and similar database operations, whether the proposed P@P technique is used or not.

## 5.4 Conclusions

Predict at Prefetch is a technique that pursues to improve web prefetching in real environments, permitting the prediction algorithm located at the web server to provide hints not only in normal fetch requests, but also in prefetch requests.

We discussed, in detail, what characteristics are required in the web browser and the prediction engine in order to perform Predict at Prefetch in a safely way. Mozilla is a well-known web browser that satisfies all the requirements, thus it can be used without any modification. Regarding the web server and the prediction engine, we

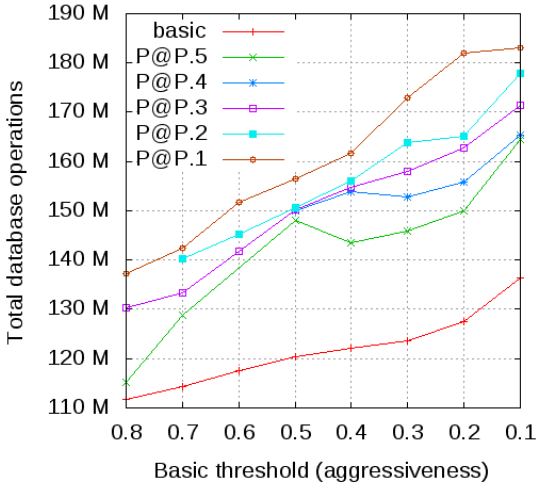



Figure 5.7: Total database operations versus algorithm threshold (aggressiveness) with the DDG prediction algorithm

proposed and implemented Predict at Prefetch on *Delfos*, and tested it with Mozilla in real world usage.

The additional aggressiveness allowed by the prediction engine when using the proposed technique reduces the users perceived page latency at the expense of increasing the traffic. The effectiveness of Predict at Prefetch with different prediction algorithms and thresholds has been checked. The results of the experiments show that a properly configured prediction engine provides a good cost-benefit ratio. A latency saving up to 12% was achieved while increasing the byte traffic by about 10%.

It was observed that using a very aggressive configuration the algorithm gathers less information, the prediction algorithm receives less requests, and it is unable to appropriately learn user patterns.

A summary of the results of this chapter was presented in [de la Ossa 07b, de la Ossa 07d].

# Chapter 6

# Theoretical limits of Web Prefetching in a real environment

## 6.1 Introduction

This chapter presents an empirical study to investigate the maximum benefits that web users can expect from prefetching techniques in the current web. Unlike previous theoretical studies, this work considers a realistic prefetching architecture using real and representative traces. In this way, the influence of real implementation constraints are considered and analyzed. The results obtained show that web prefetching can improve page latency up to 52% in the studied traces.

Many published research works focus on prediction and prefetching algorithms to improve web performance, as summarized in chapter 2.3. However, few studies [Kroeger 97, Fan 99, Domènech 06e, Balamash 07] focus on the maximum performance achievable through web prefetching and the main constraints to reach it. Some of these works study the upper bounds in performance of web prefetching from a theoretical point of view but, to the best of our knowledge, none of them has empirically analyzed how real restrictions affect the benefits of prefetching.

The main goal of this chapter is to explore the maximum benefits that web prefetching can achieve when working in the real world. To this end, a simulation framework based on a realistic prototype of web prefetching architecture has been used. We present and study the potential on performance of a *perfect* prediction algorithm which always provides accurate predictions. In this way, we can discern which performance losses come from miss-predictions and which ones come from the prefetching technique. Latency saving achieved by such algorithm is explored by varying the maximum number of provided hints.

The remainder of this chapter is organized as follows. Section 6.2 discusses the characteristics of the perfect prediction algorithm. Section 6.3 analyzes some key conditions in web prefetching and how they affect performance improvements. Section 6.4 analyzes other proposed techniques for improving web prefetching performance. Section 6.5 provides comparative experimental results with real web prediction algorithms. Finally, Section 6.6 presents some concluding remarks.

## 6.2 The Perfect Prediction Algorithm

This section discusses the *perfect* prediction algorithm used in this work to study upper bounds in latency savings.

We define a perfect prediction algorithm as a predictor having four main properties: i) it never provides wrong hints, ii) it provides at least as many hints as they can be prefetched during the browser idle time, iii) it only provides hints for those objects that have been demanded before at least once by any user, and iv) it has no adverse impact on the server functionality.

Property i) means that it provides a precision rate of 100%, and implies that this algorithm does not inject additional traffic with respect to non-prefetching techniques, since the objects prefetched according to the provided hints will always be later demanded by the user.

According to property i), the more provided hints the more latency savings. However, this number cannot be unbounded since browsers have limited slots of time to prefetch (browser idle time). This time ranges from a few seconds to several minutes in the traces. In other words, there is no way to guarantee that there will always be enough time to prefetch all the provided hints. That is why property ii) is defined. Notice that a provided hint means that the user will request that object, but this will not necessarily be prefetched since its request may result in a cache hit.

Property iii) is introduced because any real predictor requires to be informed of the existence of an object before that object can be predicted as a hint.

Finally, property iv) is defined because the algorithm consumes computational resources and this fact could affect the web server response time. Obviously, if the algorithm has to provide responses in real time, it must be fast enough to avoid delaying the normal web server functionality. There are different ways to avoid or alleviate them, for instance, locating the prediction engine in a computer different to the web server, or implementing the prediction algorithm in a custom hardware.

In summary, we define the perfect algorithm as the best algorithm that can be designed by exploring the trace file at simulation time in advance. The next sections analyze how limitations in the current real web prefetch architecture impact on the latency savings.

## 6.3 Conditions to Prefetch

This section analyzes critical issues (or factors) which can reduce the benefits on latency savings that prefetching could provide.

The evaluation methodology used to carry out these experiments is described previously in Section 4.4.

### 6.3.1 Maximum Number of Hints per Prediction

This section is aimed at determining the optimal number of hints to be sent to the prefetching engine in order to find the trade-off between bandwidth consumption and number of prefetched objects. The perfect prediction algorithm reports three main types of hints for a page: the primary object of the next page to be requested by the user, the secondary objects of that next page, and the next pages.

When the prediction algorithm makes a prediction, it may provide one, several or no hints at all. The resulting hints are sent from the web server to the web browser in HTTP response headers. Those headers consume extra bandwidth, fact that can be considered negligible when the number of hints is low. As a simple estimation, let us assume that each HTTP header hint is 32 bytes long, and a prediction always provides 10 hints; the traffic overhead caused by the hint header transference is about 1.08% in trace A and about 0.83% in trace B. If the response includes a high number of hints, for example a thousand hints, the web browser usually prefetches only a small subset of them. The reason is that a hint is neither prefetched when there is not enough time nor when the corresponding object is already in the browser cache.

Figure 6.1 illustrates how the number of provided hints affects the page latency saving for traces A and B. Experiments were run by varying the maximum number of hints (1, 5, 10, and 30) that the prediction algorithm can provide in each prediction. Each point in the figure is labeled with the average number of hints provided per response in the different experiments. Notice that this value widely differs from the number of allowed hints (with the only exception of one hint). As expected, the higher the number of allowed hints the higher the relative difference. For instance, there is no difference when allowing just one hint; however, when allowing 30 hints, the perfect prediction algorithm provides, on average, 9.66 hints (about one third of the allowed hints) in trace A. An interesting observation is that by allowing only one hint, the users perceived page latency is reduced by 34%, and when providing 10 perfect hints per prediction, the page latency is reduced by half. This means that all hints have not the same impact on latency savings, but the first ones have much stronger impact.

Regarding trace B, page latency savings are lower than in trace A although the number of provided hints is slightly higher. As the prediction algorithm is perfect, we can affirm that trace A is more predictable than trace B. However, even in this trace, important latency savings ranging from 30% to 39% (depending on the number of allowed hints) are achieved. Providing more than 30 hints does not save additional

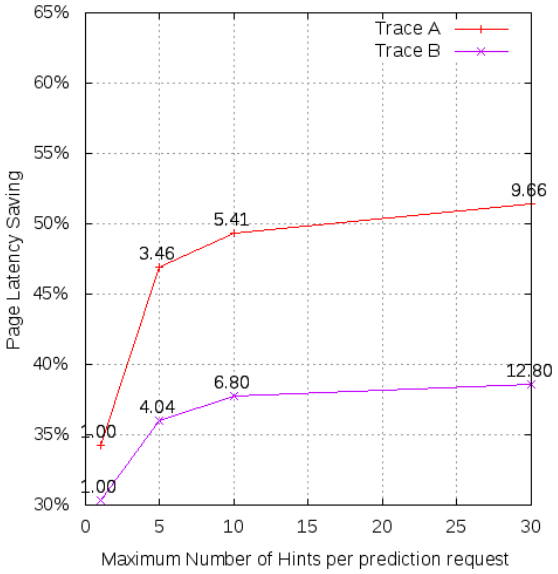



Figure 6.1: Page Latency Saving with different number of maximum hints permitted per response

page latency, which means that the raw amount of hints cannot be the only concern of a prediction algorithm.

Table 6.1 extends the results presented in Figure 6.1 to analyze, in depth, the effect that constraining the number of hints per response has on performance.

Experiments assume that the prefetching of hints ends when a new page is requested by the user. When this happens, three situations related to the provided hints can occur: i) there are hints that are still in the hint queue, waiting to be prefetched (labeled as *PCRH, Prefetch Canceled with Remaining Hints* in Table 6.1); ii) all the hints have been prefetched, and the browser is waiting for a new click (labeled as *PERIT, Prefetch Ends with Remaining Idle Time* in the aforementioned table); iii) it is the last page of the session so no more objects will be prefetched. The latter case is not shown in the table because its value always remains constant (i.e., 32.43% of the total pages in trace A, and 24.68% in trace B). Notice that even when the perfect prediction algorithm is allowed to provide as many hints as possible, barely 2.61% of the page requests have remaining hints. This fact means that the browser idle time in the trace is long enough to prefetch the hints provided by the perfect prediction algorithm.

The obtained recall is not higher than 30% in trace A, even when allowing the perfect prediction algorithm to provide all the possible hints. The reason is that many pages in the website used in the experiments share most of their secondary objects.

Table 6.1: The effect of the maximum number of hints per response

| Trace | Max hints | Provided hints | PCRH (%) | PERIT (%) | Recall (%) | $\nabla PL(\%)$ |
|---|---|---|---|---|---|---|
| A | 01 | 1.00 | 0.02 | 67.55 | 6.80 | 34.24 |
| | 05 | 3.46 | 1.69 | 65.88 | 13.66 | 46.92 |
| | 10 | 5.41 | 2.18 | 65.39 | 17.70 | 49.32 |
| | 30 | 9.66 | 2.57 | 65.00 | 25.90 | 51.41 |
| | $\infty$ | 12.20 | 2.61 | 64.96 | 25.50 | 51.77 |
| B | 01 | 1.00 | 0.17 | 75.15 | 14.58 | 30.33 |
| | 05 | 4.04 | 3.47 | 71.85 | 35.21 | 35.99 |
| | 10 | 6.80 | 5.06 | 70.26 | 44.40 | 37.76 |
| | 30 | 12.80 | 5.29 | 70.02 | 46.26 | 38.60 |
| | $\infty$ | 19.37 | 5.14 | 70.17 | 46.27 | 38.56 |

PCRH: Prefetchs that are canceled with remaining hints
PERIT: Prefetchs that end with remaining idle time

That is, the first page requested in a session requires many secondary objects to be fetched, and the subsequent pages reuse most of these secondary objects. When those secondary objects are required again, they are considered cache hits, not prefetch hits, and the recall does not increase. Since the objects of the first page in a session cannot be predicted, the prediction algorithm, even when it is perfect, can consider as hints only the objects of the second page visited as well as of the following ones. As these pages have few different secondary objects, the amount of prefetch hits is very low compared to the total amount of objects requested by the web browser. Consequently, the recall in trace A is rather low. Despite the low recall, the page latency saving is important in all cases, being higher than 51% when the number of hints allowed is over 30. This saving is much greater than the recall because the recall increases due to the secondary objects requested in the first page of a session. However, those objects represent a small amount of the users perceived page latency. As a consequence, the initial secondary objects prevent the increase of recall, but do not impede the increase of latency saving.

Results for trace B show a higher recall, which almost doubles the recall obtained by using trace A. This happens because the algorithm provides, on average, more hints per prediction than in trace A. Nevertheless, this fact does not result in a higher page latency saving. This situation is the opposite of the observed in trace A: there are many hints that are prefetched and later required by users, so they are prefetch hits that increase the recall but barely reduce the overall page latency.

### 6.3.2 Browser Idle Time

The browser idle time required to run the experiments is taken from the collected traces. This time widely differs from one request to another across the trace. Thus, depending on its value, the web browser can have enough time to prefetch all the hints either provided or not.

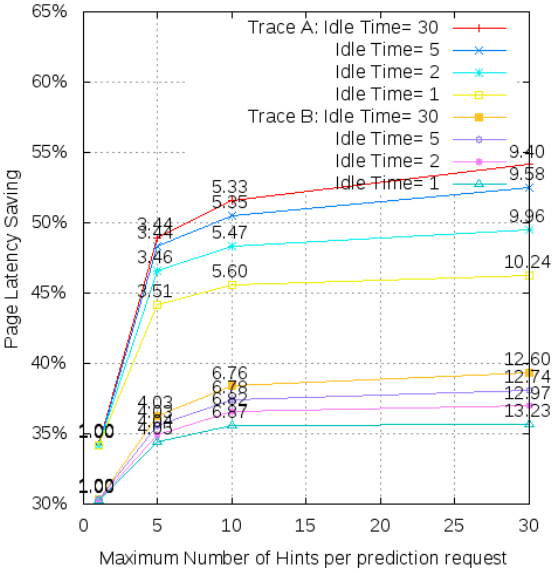



Figure 6.2: Page Latency Saving with different number of maximum hints permitted per response and idle time

This time depends on the user behavior when navigating through the website, that is, if the user navigates too fast then the web browser may not have enough time to prefetch all the hints. This happens even if the prediction algorithm provides accurate hints that permit to reach the upper bound in latency savings. If the web browser does not have enough time to prefetch those hints, the latency saved in practice will not reach the upper bound. For this reason, it is important not only to provide good hints, but also to provide them in the order in which the user will request them. Then, the web browser will prefetch them according to this order. In this way, if the browser is not able to prefetch all the provided hints, at least it will prefetch the most useful ones.

Figure 6.2 shows how the idle time affects the page latency saving in trace A and trace B. For this purpose, browser idle times are fixed to 1, 2, 5, and 30 seconds in this experiment. Results show that values longer than 30 seconds (not shown) do not further benefit latency savings. This makes sense because although the algorithm is able to provide up to 30 hints per prediction, it only generates around 10 and 13 hints on average in traces A and B, respectively. When the perfect prediction algorithm is allowed to provide only one hint, an idle time of one second is enough to prefetch the hint. This can be observed in the figure, since all curves draw the same point for abscissa 1, regardless of the assumed idle time. With the only exception of this point, the longer the idle time the higher the latency savings.

Table 6.2: The effect of the browser idle time

| Trace | Idle time (s) | Provided Hints | PCRH (%) | PERIT (%) | $\nabla PL(\%)$ |
|---|---|---|---|---|---|
| A | 1 | 13.50 | 8.84 | 58.73 | 46.04 |
| | 2 | 12.96 | 4.20 | 63.37 | 49.43 |
| | 3 | 12.58 | 2.49 | 65.08 | 51.02 |
| | 4 | 12.32 | 1.65 | 65.92 | 52.12 |
| | 5 | 12.15 | 1.18 | 66.39 | 52.65 |
| | 10 | 11.77 | 0.43 | 67.15 | 53.98 |
| | 15 | 11.66 | 0.26 | 67.31 | 54.27 |
| | 20 | 11.62 | 0.20 | 67.37 | 54.40 |
| | 30 | 11.59 | 0.14 | 67.43 | 54.57 |
| B | 1 | 20.54 | 9.16 | 66.15 | 35.61 |
| | 2 | 20.07 | 5.15 | 70.16 | 36.86 |
| | 3 | 19.81 | 3.64 | 71.67 | 37.45 |
| | 4 | 19.64 | 2.80 | 72.51 | 37.85 |
| | 5 | 19.52 | 2.28 | 73.03 | 38.06 |
| | 10 | 19.26 | 1.14 | 74.17 | 38.60 |
| | 15 | 19.17 | 0.74 | 74.57 | 38.82 |
| | 20 | 19.12 | 0.53 | 74.78 | 39.03 |
| | 30 | 19.09 | 0.34 | 74.98 | 39.35 |

PCRH: Prefetchs that are canceled with remaining hints
PERIT: Prefetchs that end with remaining idle time

Table 6.2 shows the results without constraining the number of hints per response. It can be observed that increasing the idle time by 1 second reduces the amount of pages with remaining hints by half. This amount drops to 0% when the idle time is 30 seconds. However, when the number of hints per response is limited (e.g., to 5), the number of pages having remaining hints widely decreases by just providing an idle time 1 second longer. For an idle time of 3 seconds, the number of pages having remaining hints is about 1%.

### 6.3.3 Type of Hints

A prediction algorithm provides hints that can be primary objects or secondary (embedded) objects of web documents. It is unclear if it is appropriate to restrict the type of hints that can be provided by the prediction algorithm, as discussed below. By default, we allow the prediction algorithm to provide hints of any type, that is, both primary and secondary.

To explore if the hints type has an impact on the prefetching benefits, experiments are run by providing only primary hints, only secondary, or both types. The object latency savings and page latency savings are measured, and Table 6.3 shows the results. As one can observe, the page latency is substantially reduced by just providing

Table 6.3: The effect of the type of hints permitted

| Trace | Hint type | Max hints | Provided Hints | $\nabla OL(\%)$ | $\nabla PL(\%)$ |
|---|---|---|---|---|---|
| A | both | 01 | 1.00 | 31.55 | 34.24 |
| | | 05 | 3.46 | 43.77 | 46.92 |
| | | 10 | 5.41 | 46.42 | 49.32 |
| | | 30 | 9.66 | 49.27 | 51.41 |
| | primary | 01 | 1.00 | 31.60 | 34.30 |
| | | 05 | 2.90 | 43.34 | 46.88 |
| | | 10 | 3.94 | 44.85 | 48.49 |
| | | 30 | 5.19 | 45.04 | 48.72 |
| | secondary | 01 | 1.00 | 1.58 | 1.46 |
| | | 05 | 3.10 | 2.69 | 2.24 |
| | | 10 | 4.95 | 3.73 | 2.67 |
| | | 30 | 9.41 | 5.32 | 3.62 |
| B | both | 01 | 1.00 | 27.46 | 30.33 |
| | | 05 | 4.04 | 35.96 | 35.99 |
| | | 10 | 6.80 | 38.56 | 37.76 |
| | | 30 | 12.80 | 39.58 | 38.60 |
| | primary | 01 | 1.00 | 27.28 | 30.13 |
| | | 05 | 3.28 | 30.40 | 33.26 |
| | | 10 | 4.95 | 30.51 | 33.37 |
| | | 30 | 7.91 | 30.55 | 33.39 |
| | secondary | 01 | 1.00 | 2.96 | 1.66 |
| | | 05 | 4.27 | 9.09 | 5.09 |
| | | 10 | 7.02 | 10.06 | 5.81 |
| | | 30 | 12.22 | 10.12 | 5.89 |

as hint the primary object of the next page. This reduction is about 49% in trace A and 33% in trace B. Nevertheless, prefetch hits of secondary objects only provide about 4% and 6% of page latency savings in traces A and B.

There are three main reasons why primary objects provide much more page latency savings. The first reason is that the service time of primary objects is much longer than the one of secondary objects. Nowadays, primary objects in most websites are dynamic files generated with PHP or other real-time programming languages, which often involve queries to a database. This is the case of the website in which the studied traces were collected. Secondary objects are usually plain binary files like images, or text files that only need to be read from disk, such as JavaScript code or Cascade Style Sheets. In addition, the size of HTML files in most current websites is far larger than that of secondary objects, as discussed in [Changa 08]. The second reason is that primary objects are requested by web browsers using a single connection, while secondary objects are requested simultaneously using two parallel connections. The third reason is that the browser requests the secondary objects of the page only after the primary object has been received and parsed. This makes the primary object even more relevant in the page latency.

## 6.4 Enhancing Web Prefetching

From previous sections one can conclude that the more provided hints the higher latency savings. This section analyzes the impact on latency savings of two techniques aimed at acting as a catalyst to enable the predictor to provide more hints than the baseline predictor explained above. The former technique, namely Predict On Secondary (POS), allows the perfect predictor to provide hints both in primary and in secondary objects, and is described in more detail below. In the latter technique, Predict at Prefetch (P@P) described in chapter 5, the predictor performs predictions not only when the user explicitly requests objects, but also when the browser prefetches hints.

### 6.4.1 Predict on Secondary

In general, a web document consists of a primary object and many secondary objects (images, background, style sheet, etc). In the analysis presented above, we assumed that no prediction is performed when the browser requests a secondary object. Below we discuss the reason of this assumption.

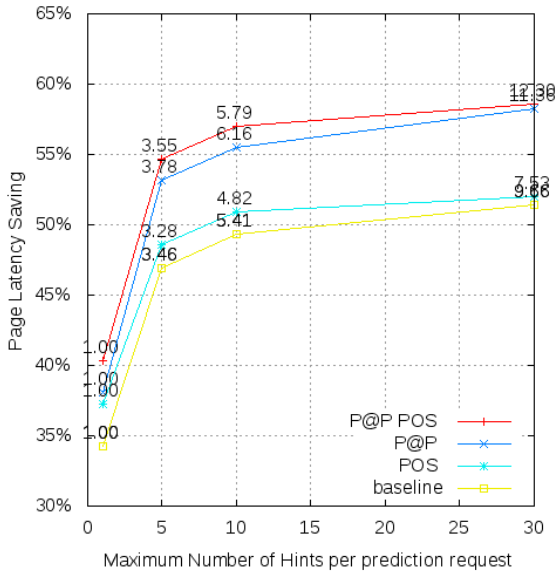
In most web prefetching research works it is assumed that a prediction is performed for each object requested to the web server. This means that the web browser receives hints in any of the requested objects. However, this does not reproduce the real user navigation pattern for two main reasons: i) the user visits web documents and not individual objects; and ii) the user visits web documents, including those that are in the cache and, consequently, are not requested to the server. In this context, Mozilla web browsers do not prefetch hints provided in secondary objects.

In a typical web prefetching architecture only the web browser is aware of which requests refer to primary objects and which ones to secondary objects. Nevertheless, predictions are made at the server side. We extend this model so that the prediction algorithm firstly receives all the object requests, and then considers if an object is primary or secondary. The perfect prediction algorithm, by definition, knows in advance which objects requested by a browser are primary and which ones are secondary. In this way, we also study the effect of making predictions in secondary objects. The hints provided with secondary objects are prefetched during the browser idle time while the object is being displayed. We refer to this technique as Predict On Secondary objects (POS).

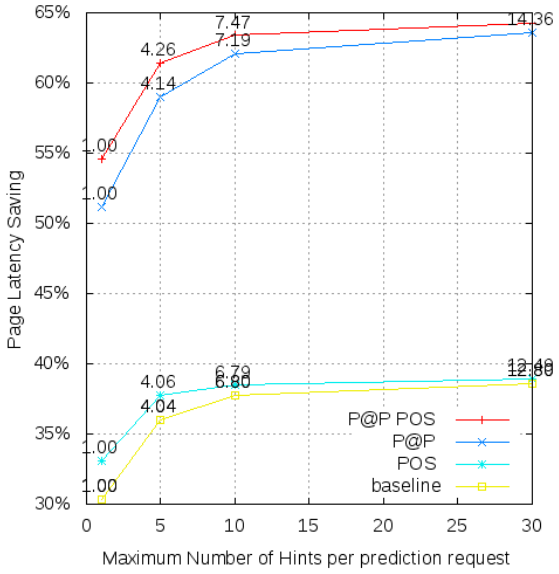
### 6.4.2 POS and P@P Experimental Results

This section evaluates to what extent the techniques described above can help to reduce users perceived latency.

As both techniques are orthogonal to each other, we measured the page latency savings that both techniques provide when working in an isolated way and when working together. Figures 6.3(a) and 6.3(b) show the results for the traces A and B,




(a) Trace A




(b) Trace B

Figure 6.3: Page Latency Saving with different number of maximum hints permitted per response, POS and P@P

respectively. For comparison purposes, the figures show the latency savings obtained with a baseline web prefetch system (that is, without P@P or POS techniques).

P@P always achieves about an additional 6% of page latency savings in trace A, regardless of whether POS is enabled or not. P@P allows the prediction in prefetch requests by providing the hints jointly with the prefetched objects. This technique exploits the browser idle time better and permits to increase the number of prefetch requests. In this way, when using a perfect prediction algorithm, the browser always has a pool of hints ready to be prefetched. For this reason, when using this technique, the perfect prediction algorithm permits to prefetch almost all the pages except the first one in the session. The benefit of P@P grows in trace B up to 12%.

As expected, POS provides scarce page latency savings (about 1%) when it works either combined with the baseline or in conjunction with the P@P technique. POS obtains more benefits when the number of hints that can be predicted is restricted (*MaxHints*). The reason is that POS allows the prediction algorithm to make more predictions when the number of hints per prediction is severely limited, or when few primary objects are requested in comparison to secondary objects. Thus, providing more predictions makes it possible to increase the total amount of hints provided.

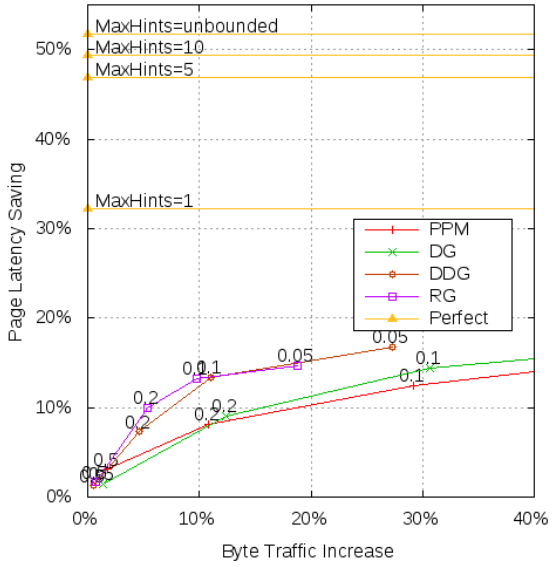
Giving hints in responses of secondary objects consumes additional computational resources to perform those additional predictions. These hints consume bandwidth when they are sent to the browser, but they are not more valuable than the hints already provided when predicting for the primary object in the document. The results suggest that the prediction algorithm should not provide hints for secondary objects, because these hints will not significantly improve the latency saving.

## 6.5 Performance Comparison With Real Prediction Algorithms

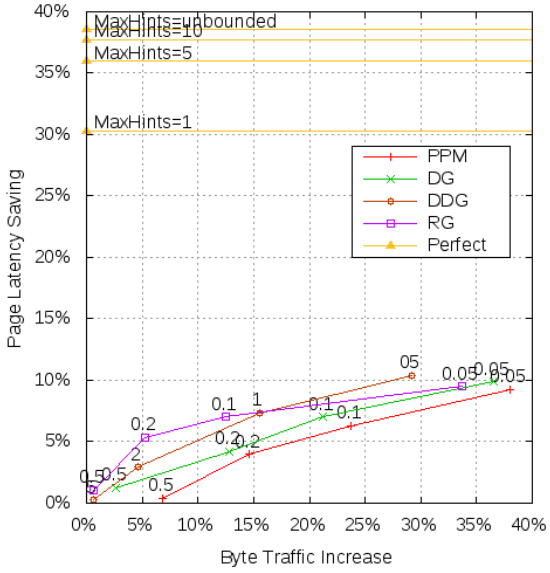
This section compares the page latency savings of several real prediction algorithms that are known to provide the best cost-benefit ratio to the perfect prediction algorithm. The main objective of this comparison is to show that the benefits achieved by real proposals are still far from the performance capabilities of web prefetching. These algorithms are the Prediction by Partial Match (PPM) [Palpanas 99], the Dependency Graph (DG) [Padmanabhan 96], the Double Dependency Graph (DDG) [Domènech 06a], and the Referrer Graph (RG) [de la Ossa 10a].

The algorithms parameters were individually tuned according to their intrinsic characteristics in order to obtain the best cost-benefit ratio with the lowest resource consumption in each individual case.

Figures 6.4(a) and 6.4(b) illustrate the cost and benefit obtained by the studied prediction algorithms in Trace A and B, respectively. In the real algorithms, the number represented in each line indicates the configured threshold, which defines the algorithm aggressiveness. The maximum number of hints allowed per prediction is 10, which is enough to obtain the best results [de la Ossa 10a]. The perfect prediction




(a) Trace A




(b) Trace B

Figure 6.4: Page Latency Saving obtained by perfect and real prediction algorithms with different aggressiveness

algorithm is represented by horizontal lines. The number in each line indicates the maximum number of hints allowed per prediction. Four different values have been studied (1, 5, 10 and unbounded).

The page latency saving quantifies the benefit, while the byte traffic increase measures the incurred cost. Experiments were run for different values of the prediction algorithms threshold parameter, because this is the most appropriate parameter to modify the aggressiveness of the algorithm prediction. Only those hints with a higher probability than the given threshold are returned to the web client.

DDG and RG obtain better cost-benefit ratio than PPM and DG. This difference is specially noticeable in Trace A, but it can also be appreciated in trace B. RG is preferable when the byte traffic is restricted, while DDG is preferable when higher byte traffic is acceptable to obtain higher page latency savings.

As depicted in both figures, all those real prediction algorithms are far from the maximum page latency savings that can be achieved by the perfect prediction algorithm in these traces. In Trace A, the best real prediction algorithms (RG and DDG) can obtain a page latency saving of 14% with a byte traffic increase of 10% and provide up to ten hints per prediction, while the perfect prediction algorithm obtains 32% latency saving by just providing one perfect hint per prediction. If this algorithm is allowed to provide up to ten hints per prediction, then it obtains almost 50% of page latency saving. In trace B, the real algorithms can only obtain up to 12% page latency saving, and the perfect prediction algorithm is almost four times better. When comparing the results of both traces, it can be observed that trace A allows us to obtain higher latency savings not only for the real algorithms used in the experiments, but also for a perfect prediction algorithm. That is, trace A is more suitable for web prefetching than trace B.

## 6.6 Conclusions

This chapter has focused on the maximum performance that web prefetching can achieve, and has analyzed the main constraints to reach it in a real scenario. To this end, we defined a *perfect* prediction algorithm. Experimental results, obtained using two recent real traces, show that the baseline perfect prediction algorithm could provide latency savings ranging from 39% to 52%, depending on the input trace. The results also show that the latency savings obtained by the current realistic prediction algorithms are far from those potentially reachable. Consequently, more research efforts are needed to reduce this gap.

Regarding how experimental conditions impact on performance, we can conclude: i) providing just five or ten perfect hints makes it possible to obtain the maximum page latency savings; ii) an idle time of about 10 seconds is enough to prefetch all the provided hints; and iii) most page latency savings are obtained by providing primary objects as hints, in contrast to secondary objects.

Two techniques were studied with the aim of improving the performance over the baseline perfect prediction algorithm. Predict on Secondary (POS) proposes to

provide hints both in primary object requests and in secondary object requests. This technique only allowed us to obtain additional page latency savings of 1% in both traces. Predict at Prefetch (P@P) provides hints both in primary object requests and in prefetch requests. Results show that this technique obtains noticeable additional savings in page latency: about 6% in the first trace and 12% in the second trace.

Therefore, we claim that a lot of research efforts should be addressed to improve current web prefetching techniques. Moreover, we feel that these results should encourage the industry to efficiently handle web prefetching in commercial products.

Partial results of the work presented in this chapter were published in [de la Ossa 09a, de la Ossa 09b] and submitted to a SCI Journal [de la Ossa 10b] in February 2010.

# Chapter 7

# Referrer Graph: a low-cost prediction algorithm

## 7.1 Introduction

This chapter presents the Referrer Graph (RG) web prediction algorithm and a pruning method for the associated graph as a low-cost solution to predict next web users accesses. RG is aimed at being used in a real web system with prefetching capabilities without degrading its performance. The algorithm learns from user accesses and builds a Markov model. These kinds of algorithms use the sequence of the user accesses to make predictions. Unlike previous Markov model based proposals, the RG algorithm differentiates dependences in objects of the same page from objects of different pages by using the object URI and the referrer in each request.

A prune mechanism is devised in order to further reduce the increase in resource usage. In this context, Section 7.4 describes and evaluates how the proposed mechanism noticeably reduces computational resources while sustaining the performance. This mechanism, referred to as pruning algorithm, removes from the graph, both periodically and continuously, those nodes, arcs, and occurrences that lose their value over time. In this way, RG can run continuously during unbounded periods of time.

The remaining of this chapter is organized as follows. Section 7.2 describes the RG prediction algorithm. Section 7.3 shows and analyzes the experimental results. Section 7.4 presents and evaluates a graph pruning mechanism. Finally, Section 7.5 presents some concluding remarks.

## 7.2 Referrer Graph

### 7.2.1 General Description

The Referrer Graph (RG) prediction algorithm builds a graph based on client requests. Each requested web object is represented by a node in the graph. For each web request that reports its referrer, an arc is created from the referred node (predecessor) to the requested nodes (successor). The resulting graph is used to make predictions that produce hints which are returned to the web client.

The idea of a prediction algorithm that considers the site structure accessed by users is motivated by the fact that users commonly navigate following hyperlinks on web pages. This model considers that two objects are related to each other if there is a hyperlink between them. Hence, the model considers structural information about the website instead of the sequence of the requested objects. In general, the algorithms that use the sequence of requests to build the graph need a window of last accesses for each client session in order to establish arcs among the previous accesses and the current one. RG does not need this window because this algorithm keeps the relationship between objects based on their direct reference, which is known thanks to the referrer field in the request [1]. Each HTTP object request includes information about the requested object and its referrer, so in order to establish an arc it is not necessary to keep track of previous accesses of the same client session.

Many prediction algorithms use the sequence of user accesses and identify each navigation session by the IP address of the client. This is a problem when different browsing sessions use the same IP address, which happens when several clients use the same proxy server, when they connect from the same local network masked by a single IP address, or when the same user has several parallel browsing sessions. Those events cause an erroneous learning of user patterns in the algorithms. On the contrary, RG handles each web request independently of the context of the whole navigation session. Therefore, RG does not need to keep track of each user navigation session, and it is not negatively affected by any of these circumstances.

Conceptually, the RG algorithm is similar to DDG, but instead of using the sequence of accesses to build the associated graph, RG extracts information from individual accesses to reconstruct the website structure and the access patterns, and it builds a graph of object dependences by using the popularity of the objects and the relationships among them. Below, we detail how RG works.

### 7.2.2 Theoretical Example

This example shows the different learning in RG and a prediction algorithm based on the sequence of accesses.

[1] Note that in the original HTTP specification document, the header field was misspelled as "referer", instead of the correct English word "referrer".

Given a user sequence of web object requests:

$$\text{User requests} \quad = \quad \text{A} \Rightarrow \text{B} \Rightarrow \text{C} \tag{7.1}$$

And considering that the prediction algorithm predicts the hints B and X for A, the client performs the following actions:

$$\text{Client actions} \quad = \quad \text{fetch A} \Rightarrow \text{prefetch B} \Rightarrow \text{cache hit B} \Rightarrow \text{fetch C} \tag{7.2}$$

The prediction engine receives only two requests:

$$\text{Predictor receives} \quad = \quad (\text{A with referrer } \phi) \Rightarrow (\text{C with referrer B}) \tag{7.3}$$

A prediction algorithm that learns from the sequence of accesses, like DDG, generates two arcs in its graph:

$$\text{Sequence learning} \quad = \quad (\phi \rightarrow \text{A}) \Rightarrow (\text{A} \rightarrow \text{C}) \tag{7.4}$$

It generates and arc from A to C because that is the perceived sequence of accesses. On the contrary, RG generates the arc considering the individual requests, not their sequence, so it learns:

$$\text{Referrer learning} \quad = \quad (\phi \rightarrow \text{A}) \Rightarrow (\text{B} \rightarrow \text{C}) \tag{7.5}$$

In this case, RG builds the arc correctly even when some intermediate requests are missing.

### 7.2.3 Data Structures

RG builds a first-order Markov model, since only the previous access is used to define the context of the current request. The graph is directed, loopless, and cyclic.

A Markov model is a directed cyclic graph where the next node only depends on the current state. The node represents the context of the user, that is, the recent accesses. Nodes are connected by arcs, which represent the transition between contexts. The arc weight represents the transition confidence of moving from the predecessor node to the successor node. The *order* of a Markov model indicates how many past accesses are used to define the context in a node.

In the proposed model, a node represents a web object, and it is identified by the object URI. A directed arc from a predecessor to a successor node represents the relationship between both nodes.

To illustrate several working aspects of the RG algorithm, let us assume a Calendar website, as shown in Figure 7.1. The main page (*Calendars*) is linked to some other secondary pages that focus on different types of calendars: there is a different page for each year calendar (*C2007*, *C2008* and *C2009*); there is page with the rules used to build the Gregorian calendar (*Gregorian*), another page with the rules for the Mayan

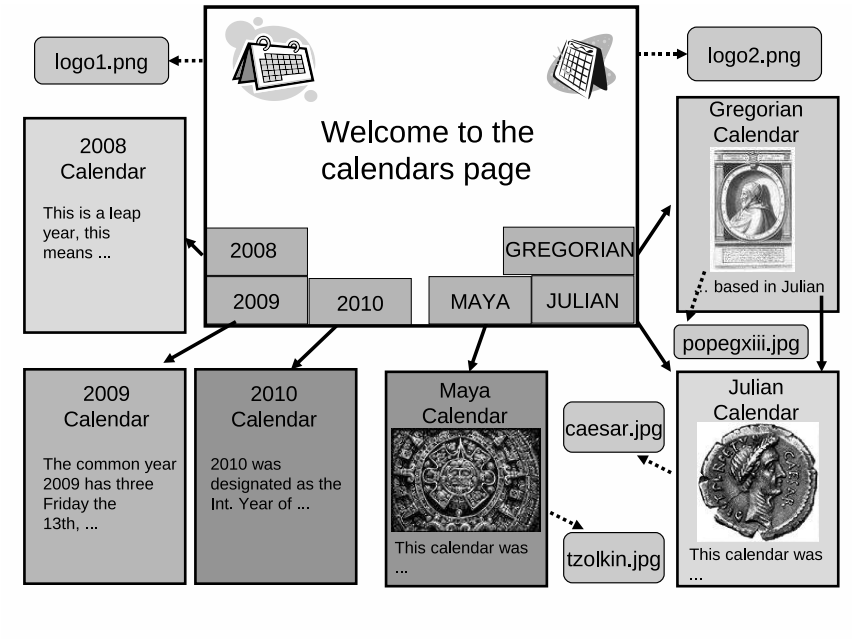



Figure 7.1: Example website

calendar (*Mayan*), and a third one with the rules for the Julian calendar (*Julian*). Images and logos are embedded objects of their own pages.

Figure 7.2 shows an example graph built by RG for a simple navigation performed in the Calendar website. In the graph there is a primary arc that has node *Calendars* as a predecessor, and node *Julian* as a successor. This means that a user requested the object represented by node *Julian*, and that request indicated as referrer the object represented by node *Calendars*.

Initially, the graph is empty. Then it is built and updated through a learning process. We define the occurrence of a node as the number of requests to the represented object, and the occurrence of an arc as the number of requests to the successor node which provided the predecessor node as a referrer. For each web request, if the graph already contains a node representing that object, the node occurrence is increased. Otherwise, the node is created with occurrence set to one. In addition, an arc is created or, if it already exists, its occurrence is increased. To avoid the uncontrolled growth of the graph, it can be periodically pruned by removing those nodes and arcs that become less representative.

Nodes and arcs can be classified in two main types: primary and secondary. Primary nodes represent objects that are requested explicitly by users, while secondary nodes represent embedded objects that are requested by the web client, not by the

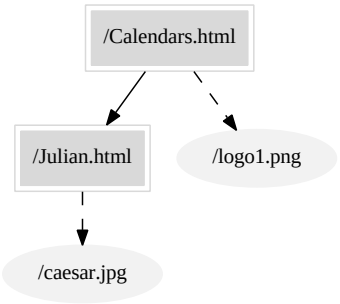



Figure 7.2: Simple graph with two primary and two secondary nodes

user. A secondary node is not predecessor of any arc and its object's MIME type corresponds to a typical embedded object (image, video, script, style sheet, etc). A secondary node is promoted to primary when an arc having that node as predecessor is established. Thus, it is not possible that an arc has a secondary node as predecessor. On the contrary, primary nodes can never be promoted back to secondary.

An arc inherits the type of its successor node. When a secondary node is promoted to primary, the arcs that have this node as successor are also promoted to primary.

A web page visited by a user consists of a main object and several embedded objects. The graph represents this page as a primary node, several secondary nodes, a primary arc from the referrer node to the primary node, and secondary arcs from the primary node to the secondary nodes.

In the proposed data structure, the graph stores for each node: the object URI, node type, node occurrence, list of referrers, list of primary arcs, and list of secondary arcs. The data stored for an arc is the destination URI, arc occurrence, and arc transition confidence.

A secondary node keeps a list of referrer nodes because this allows to quickly find them if the node is promoted to primary.

### 7.2.4 Learning Process

The learning part of RG is summarized in Figure 7.3 and consists of three main steps: updating the requested node, updating the arc from the referrer node to the requested node, and promoting the referrer node, if required.

When an object is requested, its corresponding node is found, and the occurrence and the list of referrers of the node are updated. If the node has not been created yet, it is created with a type according to the object MIME type, and inserted in the graph. If this information is not available, or the MIME type does not allow to discern the node type, then it is built as a secondary node.

If the node is secondary, it adds the referrer URI to a list. This is useful in case of node promotion, as it helps to quickly find the nodes that link to this node and the arcs that have this node as successor.

If the request provides the requested object as referrer, or if the referrer is a node that does not exist in the graph (this includes the case of external referrers), then the learning process ends. This ensures that arcs have a different node as predecessor and successor.

If the referrer already has a node in the graph and there is an arc from the referrer to the requested object, the arc occurrence is increased. In addition, all the arcs that have the referrer as the predecessor node are updated: the arc transition confidence is calculated as the arc occurrence divided by the predecessor node occurrence. If the predecessor is a secondary node, it is promoted to primary and all the arcs with that node as successor are promoted to primary, too.

### 7.2.5 Prediction Process

Figure 7.4 shows the main steps of the algorithm that makes predictions and provides hints. When a user requests an object, the corresponding node is looked for in the graph. If the node does not exist or it is secondary, no prediction is made and no hints are provided. On the contrary, if the node exists and is primary, all its primary arcs are analyzed. Those arcs that have a transition confidence greater than a given threshold are included in a list of primary arcs, ordered by transition confidence. Then the successor nodes of such arcs are looked for and their secondary arcs are analyzed. Those secondary arcs with a transition confidence (multiplied by the previous primary arc transition confidence) greater than a secondary threshold are included in a list of secondary result arcs, also ordered by arc transition confidence.

The lists of primary and secondary arcs are then concatenated. The URIs associated to these arcs are the hints that will be provided as predictions. The definitive list of hints can be cut to provide only the first N hints.

### 7.2.6 Working Example

This section presents an example of a set of clients sessions browsing the Calendar website, the graph built by the learning process, and the hints provided by the prediction algorithm.

Table 7.1 lists the web requests performed during the client session, which are processed by the learning process of RG. Figure 7.5 shows the corresponding graph. The primary and secondary nodes are depicted with solid and dashed lines, respectively. Nodes are labeled with the URI and their occurrence, while arcs are labeled with the arc occurrence and transition confidence. Table 7.2 shows the hints that RG would provide using the graph previously shown if a client performed different requests. The main threshold and the secondary threshold are not enforced in this example.

Figure 7.3: Algorithm for learning from user access and building the RG graph

**Input:**
  $rg$: RG graph
  $uri$: URI requested
  $referrer$: Referrer provided in the request
  $mime$: MIME type of the requested object
**Output:**
  $rg$: RG graph with improved knowledge
{ **UPDATING NODE: Updating the requested node** $b$ }
$b \leftarrow$ find node with $uri \in rg$ or build new node, with type based on $mime$
$b$ occurrence $\leftarrow b\ occurrence + 1$
**if** $b$ type = secondary **then**
  Add to $b$ list of referrers: $referrer$
**end if**
Store $b$ in $rg$
{ **UPDATING ARC: Updating arc** $\vec{ab}$ }
**if** $uri = referrer$ or $referrer \notin rg$ **then**
  **return** $rg$
**end if**
$a \leftarrow$ find node with $referrer \in rg$
$\vec{ab} \leftarrow$ find arc with $uri \in a$ or build new arc
$\vec{ab}$ occurrence $\leftarrow \vec{ab}$ occurrence +1
Store $\vec{ab}$ in $a$
**for all** arcs $\vec{at} \in a$ in $rg$ **do**
  $\vec{at}$ transition confidence $\leftarrow \vec{at}$ occurrence / $a$ occurrence
**end for**
{ **PROMOTION: Promoting Referrer node** $a$ }
**if** $a$ type = secondary **then**
  $a$ type $\leftarrow$ primary
  **for all** uris $urix \in a$ list of referrers in $rg$ **do**
    $x \leftarrow$ find node with $urix \in rg$
    $\vec{xa} \leftarrow$ find secondary arc with $referrer \in x$
    $x$ secondary arcs $\leftarrow$ remove $\vec{xa}$ from $x$ secondary arcs
    $x$ primary arcs $\leftarrow x$ primary arcs $\bigcup \vec{xa}$
    Store $x$ in $rg$
  **end for**
**end if**
Store $a$ in $rg$
**return** $rg$

Table 7.1: Web requests in an example of a client session

| URI requested | URI of referrer |
|---|---|
| /Calendars.html | - |
| /logo1.png | /Calendars.html |
| /logo2.png | /Calendars.html |
| /Gregorian.html | /Calendars.html |
| /popegxiii.jpg | /Gregorian.html |
| /Julian.html | /Gregorian.html |
| /caesar.jpg | /Julian.html |
| /Calendars.html | - |
| /logo1.png | /Calendars.html |
| /Maya.html | /Calendars.html |
| /tzolkin.jpg | /Maya.html |
| /Calendars.html | www.search.com |
| /logo1.png | /Calendars.html |
| /Gregorian.html | /Calendars.html |
| /popegxiii.jpg | /Gregorian.html |
| /Julian.html | /Gregorian.html |
| /caesar.jpg | /Julian.html |
| /Calendars.html | /Julian.html |

Table 7.2: Hints provided depending on the requested URI

| URI | Hints URI and probability |
|---|---|
| /Calendars.html | /Gregorian.html 50%, /Maya.html 25%, /popegxiii.jpg 50%, /tzolkin.png 25% |
| /Gregorian.html | /Julian.html 100%, /caesar.jpg 100% |
| /Julian.html | /Calendars.html 50%, /logo1.png 37.5%, /logo2.png 12.5% |
| /Maya.html | - |

Figure 7.4: Algorithm for giving hints based on RG graph

```
1: Input:
2:   rg: RG graph
3:   u: last user access
4:   th: primary threshold
5:   thsec: secondary threshold
6: Output:
7:   h: Set of hints
8: for all output primary arcs a ∈ u in rg do
9:   if a transition confidence ≥ th then
10:    hptemp ← hptemp ⋃ {a}
11:    for all output secondary arcs e ∈ a in rg do
12:      if e transition confidence ·a transition confidence ≥ thsec then
13:        hstemp ← hstemp ⋃ {e}
14:      end if
15:    end for
16:  end if
17: end for
18: hpsorted ← sort hptemp by higher probability
19: hsuniq ← delete duplicates in hstemp
20: hssorted ← sort hsuniq by higher probability
21: h ← hpsorted ⋃ hssorted
22: return h
```

## 7.3 Experimental Results

This section compares the cost-benefit and the resource consumption of the RG prediction algorithm against other three well-known algorithms. These algorithms are the Prediction by Partial Match (PPM) [Palpanas 99], the Dependency Graph (DG) [Padmanabhan 96], and the Double Dependency Graph (DDG) [Domènech 06a].

The algorithms parameters were individually tuned according to their intrinsic characteristics in order to obtain the best cost-benefit ratio with the less resource consumption. We found that, in both traces, the best configuration was the use of a lookahead window size of 2 in both DG and DDG, and a first-order Markov model in PPM.

### 7.3.1 Page Latency Saving and Byte Traffic Increase

Figure 7.6 illustrates the cost and benefit obtained by the studied prediction algorithms. The page latency saving quantifies the benefit, while the byte traffic increase measures the incurred cost. Experiments were run for different values of the prediction algorithms threshold parameter, because this is the most appropriate parameter to modify the aggressiveness of the algorithm prediction. Only those hints with a

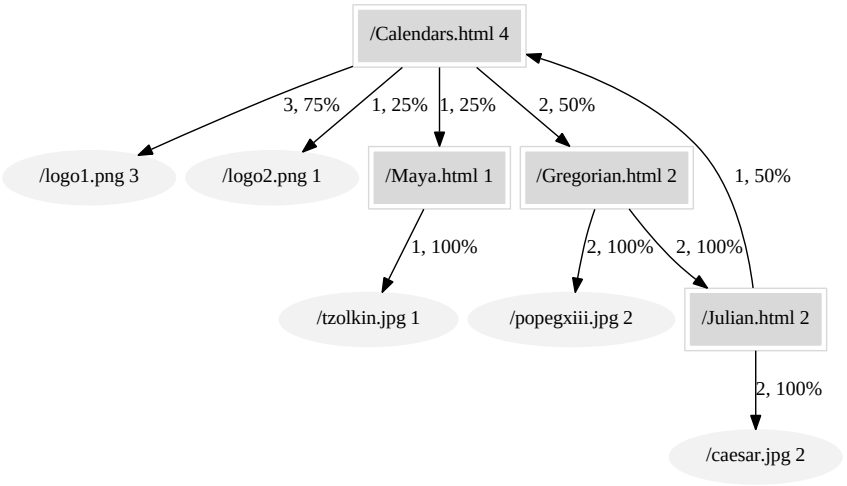



Figure 7.5: Graph generated by RG after some simple navigation sessions

higher probability than the threshold are returned to the web client. The numbers inside the figure show the threshold values used in the experiment run.

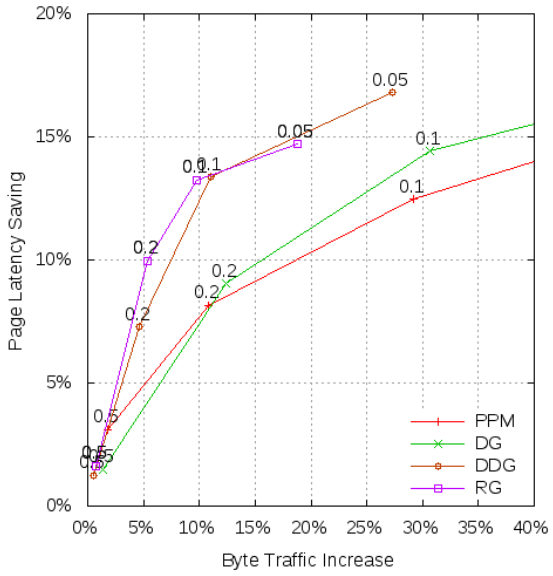
DDG and RG obtained better cost-benefit ratio than PPM and DG. This difference is specially noticeable in trace A (see Fig. 7.6(a)), but it can also be seen in trace B (see Fig. 7.6(b)). This difference arises because DDG and RG discern among primary and secondary objects in the graph; thus, they can provide more useful hints. As observed, RG is the preferable algorithm under low-cost constrains, while DDG is the best when no constrains are imposed.

In [de la Ossa 09a] we studied the maximum page latency savings that prefetching can provide under real conditions. Results showed that latency savings can be as high as 52% and 39% for trace A and trace B, respectively. These results were achieved by providing just five or ten perfect hints.

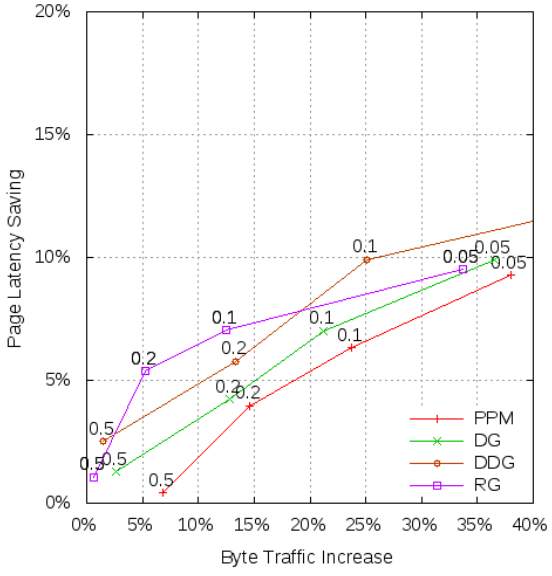
We performed experiments using the methodology described in [de la Ossa 09a] to obtain the maximum page latency saving that could be achieved using web prefetching in those traces. The results showed that the maximum page latency savings that web prefetching can obtain are 52% for trace A, and 39% for trace B.

### 7.3.2 Resource Consumption

To increase the prediction accuracy, the prediction algorithms store a lot of information about user's navigation and, consequently, have to perform a deeper information analysis handling a high number of variables to make predictions. As a consequence, prediction algorithms become more and more complex. In other words, the algorithms require more computational and memory resources both to learn from the user be-




(a) Trace A




(b) Trace B

Figure 7.6: Page latency saving versus byte traffic increase

haviour and to make predictions. Therefore, the research must concentrate not only on the precision of the prediction algorithms, but also on their resource consumption. A prediction algorithm whose resource consumption increases exponentially is not appropriate for real usage. When two prediction algorithms provide similar precision, it is preferable the one with lower resource consumption.

We quantified the resource consumption of the previously mentioned algorithms. To quantify the memory consumption we measured the number of arcs in the graph that is built by each prediction algorithm. The number of nodes in the graphs built by the algorithms DG, DDG and RG is identical because all the algorithms receive the same number of object requests, and all of them create a node for each requested object. As an approximation to the computational consumption, we measured the service time required by the algorithms to make predictions.

The algorithms DG, DDG, and RG create the same nodes in their graphs, although they are connected in a different way. However, PPM builds a data structure that is completely different. Consequently, its data structure is not directly comparable to the other algorithms, as each requested object is represented with one or more nodes in the PPM tree.

Figure 7.7 shows the number of arcs in the graph of each algorithm during the experiment measured in number of user requests.

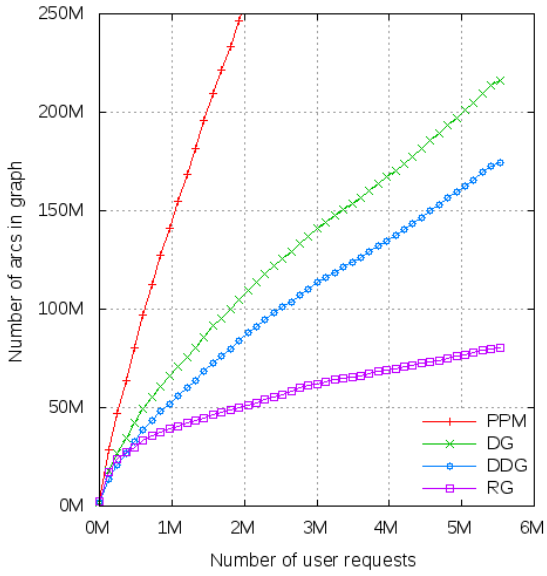
The data structure created by any algorithm strongly depends on the website navigation tree. This is the cause of the big difference in cost between trace A and trace B.

The algorithm that always created fewer arcs in its graph was RG. The reason is that RG creates an arc between two nodes when a web request of the second node references the first one. This is the main difference between RG and the other algorithms, where an arc is created between two nodes when they are requested sequentially during a short lapse of time.

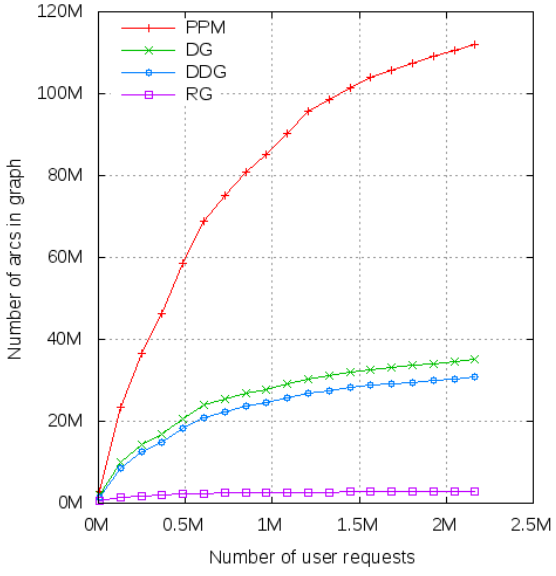
Figure 7.8 depicts the prediction service time during the experiment, that is, the time consumed by the implementation of the prediction algorithm to make a prediction. This time consists of two main components: the learning phase time and the prediction phase time. A server equipped with a 64-bit Intel Xeon Processor 3.2GHz, 2M Cache, 4 cores, with 4 GB of RAM was used to perform the experiments.

As one can observe, the prediction service time (Fig. 7.8) is directly related to the number of arcs in the graph (Fig. 7.7). The reason is that those algorithms consume most of their prediction time looping over the arcs in the graph and performing actions with each arc.

RG was the implemented algorithm that required less time to make predictions, followed by DDG, DG and PPM. However, in all cases the service time increased as the experiment progressed. The reason is that the traces included requests for several months. During the months when the trace was captured, new pages were added to the website. Therefore, the prediction algorithms were continuously adding more nodes to the graph, as well as arcs between old and new nodes. As a consequence, the prediction service time increased. To avoid the continuous increase of this time,

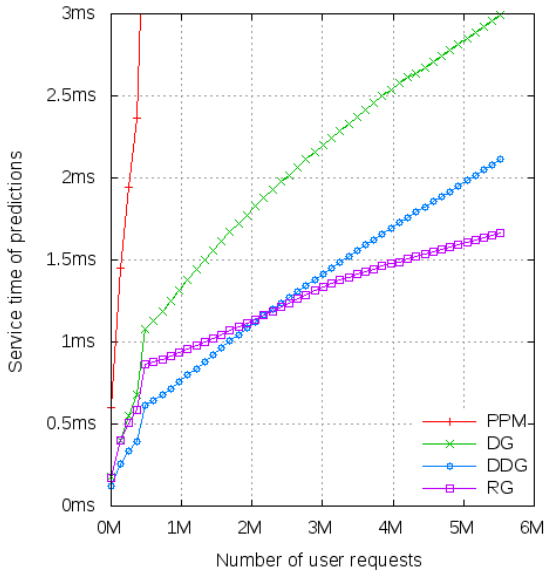



(a) Trace A

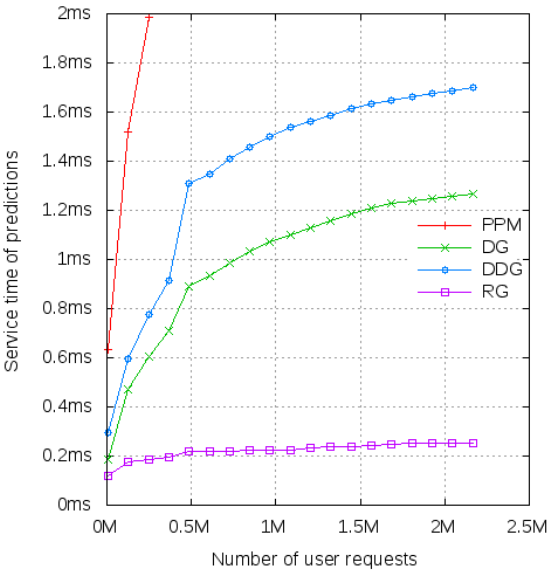



(b) Trace B

Figure 7.7: Number of arcs in graph when threshold is 0.1




(a) Trace A




(b) Trace B

Figure 7.8: Prediction service time when threshold is 0.1

we propose to prune the graph, that is, to reduce the graph complexity, as described below.

## 7.4 Graph Pruning

This section presents a method for pruning the graph built by RG. First, the main questions related to the growing of the graph and data structures associated with the prediction algorithm are discussed. Then, the proposed pruning algorithm is presented, and the design choices are explained in detail. An example of RG pruning illustrates how the algorithm works in practice. Finally, the benefits of RG pruning against the baseline RG are evaluated.

### 7.4.1 General Issues

The graph associated with a prediction algorithm is dynamically built over time and updated with the corresponding information when a user access is performed. As a result, the graph grows and continually needs more computational resources. The problem arises when some information stored in the graph becomes stale or useless. This fact can happen due to two main reasons: i) the changes on web navigation patterns, and ii) the removal of some website pages. The first case appears because users' interests vary over time. This may happen depending, among others, on the kind of contents the site provides (e.g., sports news, financial news, program documentation,...). The second case occurs because, when some pages are removed from a website, some knowledge in the graph becomes useless since users will no longer visit such pages. Furthermore, keeping obsolete and useless information in the graph not only wastes memory and computational resources in the prediction engine, but also has a negative impact on the performance because the precision can noticeably decrease. That is, if such information was used by the prediction algorithm, it could provide useless hints. The process of pruning a graph consists of walking on the graph to check which parts have become obsolete, and therefore suitable for being removed.

The design of a pruning algorithm must address two major concerns: resource consumption, and prediction accuracy. The first one means that pruning must not increase the resource consumption of the original prediction algorithm. That is, the sum of resource consumption due to the prediction and the pruning algorithms must not exceed the consumption when pruning is not performed. This entails that the pruning algorithm itself must be a low-consumption process and removing information results in a simpler graph that consumes less computational resources. The second concern means that pruning must not have adverse impact on the effectiveness of the web prediction. In other words, pruning the graph implies the removal of information, so if useful information was pruned, prediction accuracy would be adversely affected.

### 7.4.2 Proposed Pruning Algorithm

The graph built by RG has two elements that grow over time: nodes that represent pages, and arcs that represent transitions between pages. Node and arc occurrences are counters that increase continuously, as they represent the number of times a page or a transition have been observed. According to the implementation of the data structures, the action of pruning a node also prunes the arcs having that node as predecessor. On the other hand, occurrences are internally represented with integer values, so an integer overflow can raise arithmetic exceptions because an arc or a node is being highly accessed over time. In this sense, an effective pruning could help to avoid such exceptions.

The design of a pruning algorithm is determined by two main decisions: i) what to prune, and ii) when to prune.

The first decision consists in selecting those elements that must be pruned from the graph. This process must cover the pruning of the three main elements of the graph: nodes, arcs, and occurrence values. Notice that pruning nodes and arcs means to remove them from the graph, while pruning occurrences refers to decreasing their value. In order to perform a fair pruning when decreasing the occurrence values, the overall consistency must be kept, that is, these values must be kept proportional to each other. To this end, the occurrences of the arcs are all reduced at the same time, then the occurrence of the predecessor nodes is reduced in the same ratio.

The second decision refers to the points in time at which pruning is performed. This can be done either continuously, periodically, or both. In a continuous pruning, when an element of the graph is accessed, only that element and the directly related elements are checked for pruning. This process is performed dynamically and synchronously with the learning process, so computational consumption extends over time. On the other hand, in a periodic pruning all the graph contents are checked for pruning at fixed intervals. In the latter case, the computational consumption concentrates on the lapse of time when pruning is triggered.

Figure 7.9 summarizes the pruning process devised for the RG algorithm. Continuous pruning is used for arcs, and periodical pruning for nodes. Notice that the pruning and learning algorithms (Figure 7.3) must be interleaved as follows: first, updating the requested node and the arc from its referrer; second, pruning arcs and reducing occurrences; then, promoting the referrer node; and finally pruning nodes. Consequently, the order in the complete algorithm includes the following steps:

1. UPDATING NODE (Figure 7.3)
2. UPDATING ARC (Figure 7.3)
3. PRUNING ARCS AND REDUCING OCCURRENCES (Figure 7.9)
4. PROMOTION (Figure 7.3)
5. PRUNING NODES (Figure 7.9)

The criterion used for pruning arcs and occurrences, as highlighted in Figure 7.9 (PRUNING ARCS AND REDUCING OCCURRENCES), is the following. When a request indicates as its referrer a node that already exists in the graph, and the node occurrence exceeds a threshold (*node_occ_th*), its outgoing arcs are checked for pruning. An arc is pruned when its transition confidence is lower than the primary *threshold*; otherwise the arc occurrence value is reduced by a factor (*occ_reduction_factor*). As mentioned in Section 7.2.4, the arc transition confidence is calculated as the ratio of its occurrence value to the occurrence value of its predecessor node. Proceeding in this way, the occurrence value estimates the arc popularity. Once all the arcs leaving a node have been updated, the occurrence of the node is also reduced by the mentioned factor to maintain this popularity.

Node pruning is highlighted in Figure 7.9 (PRUNING NODES). A global *access counter* is used to determine the point in time at which the pruning of all nodes in the graph must start. This counter increases its value each time a web request occurs. When this counter surpasses a given value (*pruning_th*), the node pruning process starts. Nodes are pruned when they i) do not reach a minimum popularity or ii) have not been accessed for a long time. A node is considered popular enough to be maintained in the graph if its node occurrence value is higher than a minimal configured value (*node_occ_th*) in the pruning algorithm.

To determine the elapsed time since a node was accessed, a new variable is associated with each node (*last access* field). When a node is accessed, this field gets the value of the global *access counter* mentioned above (which also increases its value). Thus, if the *last access* value of a node is lower than a configured access threshold (*access_th*) set in the pruning algorithm, then the node is pruned. Otherwise the node *last access* field is set to zero. When the node pruning process finishes, the global *access counter* is reset, thus becoming ready to start a new learning process. When a node is pruned, all the arcs leaving that node or having this node as their destination are also removed.

Finally, notice that node pruning requires to walk on the entire graph because the nodes most likely to be pruned are the less requested ones.

### 7.4.3 Example of RG Pruning

To illustrate the pruning method, this section presents an example of a graph built by RG and the graph that results from the pruning of arcs and nodes. The graph built corresponds to the navigations performed by a set of clients to the Calendar website proposed in Figure 7.1.

In this scenario, navigation sessions have also been assumed to build the RG graph shown in Figure 7.10(a). As known, nodes represent the visited pages, and arcs show the transitions among them. Each node shows the value of its occurrence and its *last access*, and each arc shows its occurrence and its transition confidence.

The corresponding navigations were as follows: many clients accessed the website by first visiting its main page, which is, consequently, very popular (see occurrence of

Figure 7.9: Algorithm for pruning the RG graph

```
1: Input:
2:   rg: RG graph
3:   referrer: Referrer provided in the request
4:   th: Primary threshold
5:   accesscounter: Global counter of accesses
6: Output:
7:   rg: RG graph with improved knowledge
   { PRUNING ARCS AND REDUCING OCCURRENCES }
8: a ← find node with referrer ∈ rg
9: if a occurrence ≥ node_occ_th then
10:    for all arcs ā⃗r ∈ a in rg do
11:       if ā⃗r transition confidence < th then
12:          prune arc ā⃗r
13:       else
14:          ā⃗r occ ← ā⃗r occ * occ_reduction_factor
15:       end if
16:    end for
17:    a occ ← a occ * occ_reduction_factor
18: end if
19: Store a in rg
   { PRUNING NODES }
20: if accesscounter ≥ pruning_th then
21:    for all nodes n ∈ rg do
22:       if n occ < node_occ_th or n last access < access_th then
23:          prune node n
24:       else
25:          n last access ← 0
26:       end if
27:    end for
28:    accesscounter ← 0
29: end if
30: return rg
```

*Calendars*). Few visitors were interested in the 2008 calendar or in the Mayan one. In fact, the last accesses to those pages were performed a long time ago (compare the global *access counter* with the corresponding *last access* value). The page showing the *Gregorian* calendar is quite popular but few users requested it before accessing the main page, possibly because one of the visitors shared the page URL with his or her friends, so the new visitors requested the *Gregorian* page directly, without requesting *Calendars* before. This can be observed in the graph by comparing the high value of the *Gregorian* node occurrence to the low occurrence value of the arc that links the *Calendars* node to the *Gregorian* one. The numerical values of these fields show possible consistent values with the users behavior described above.

The pruning algorithm parameters for this demonstration are set as follows: *threshold* = 15%, *node_occ_th* = 50, *occ_reduction_factor* = 0.1, *pruning_th* = 1000, *access_th* = 500. In this example, let us suppose that the RG algorithm receives a new prediction request for *Calendars*. Then, a prediction is performed, the graph is updated to reflect the new access, and the pruning mechanism is triggered.

The pruning process starts by checking the arcs that leave the node Calendar. Two of the arcs are pruned because their transition confidence is lower than the primary threshold: the arc to the *Gregorian* node, and the arc to the *Mayan* one. Then, the remaining arcs and nodes update their occurrences using the reduction factor mentioned above. Then, when the global *access counter* reaches the pruning threshold (*pruning_th*), the process starts and all nodes in the graph are checked for pruning. Besides its popularity, the node *C2008* is removed because its last access was performed a long time ago. The *Mayan* node is also deleted but, unlike the previous one, it is deleted because it has a very low occurrence (which means it has not been popular since the last node pruning process). Of course, the arcs that arrive at or leave the pruned nodes are also removed from the graph. The graph resulting from the pruning is shown in Figure 7.10(b).

In summary, the initial graph of five nodes is reduced to only three nodes, and only one arc remains from the four initial arcs. Those elements of the graph were removed because they were not popular or had not been accessed for a long time.

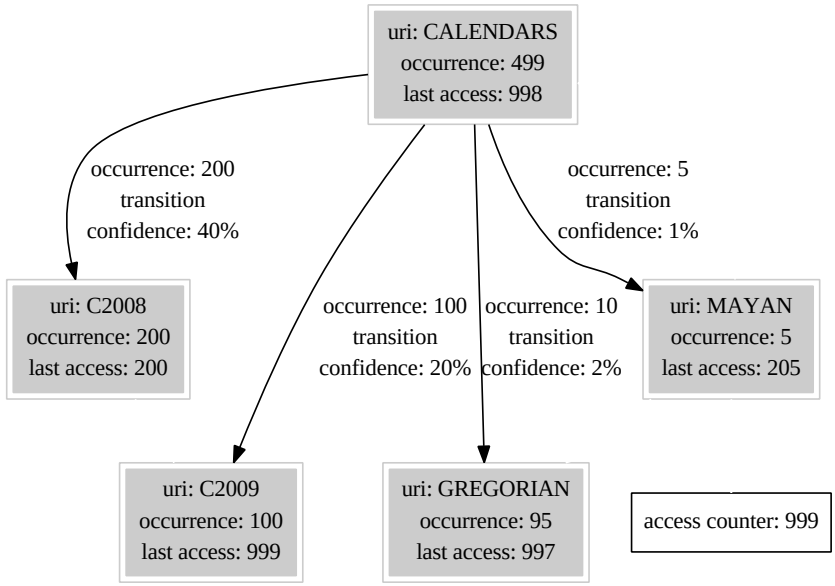
### 7.4.4 Experimental Results of Pruning

This section evaluates the benefits of pruning. Graph complexity, prediction time, and page latency savings have been measured and compared to the original algorithm. To carry out the experiments, the *node_occ_th* and the *access_th* were set to 10 and 250,000 accesses, respectively. For the sake of clarity, the results of each trace are presented in a separate plot across the performed experiments because both traces present quite different characteristics.

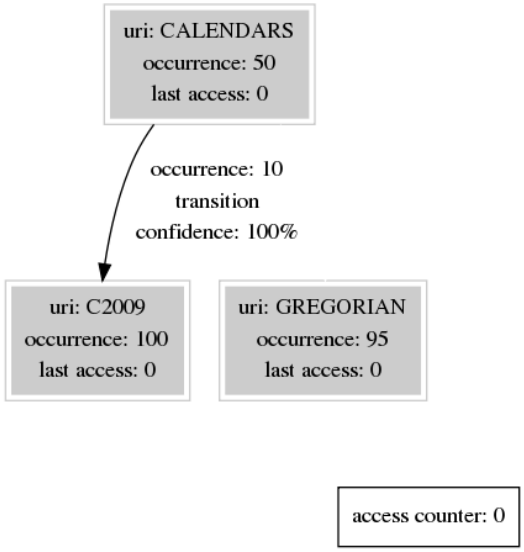
Figure 7.11 shows, for both traces, the increase in the number of nodes in the graph as the experiment progresses. Regardless of the trace, when using the original RG algorithm (upper curve), the number of nodes increases noticeably during the experiment. As expected, when applying pruning, the graph complexity is reduced each time pruning is performed.

Figure 7.12 illustrates how the number of arcs evolves in the graph when applying pruning and in the original algorithm. When pruning arcs, these are continually removed during the experiment, whereas when pruning nodes, these are removed at fixed intervals. The reduction in the number of arcs is caused by the periodic node pruning, because the removal also affects the number of arcs, that is, when a node is removed from the graph all its outgoing arcs are also removed.

Figure 7.13 shows the average service time taken by the RG prediction algorithm to perform predictions. In addition to the learning and predicting phases, this time includes the pruning phase. In spite of this fact, compared to the original algorithm,

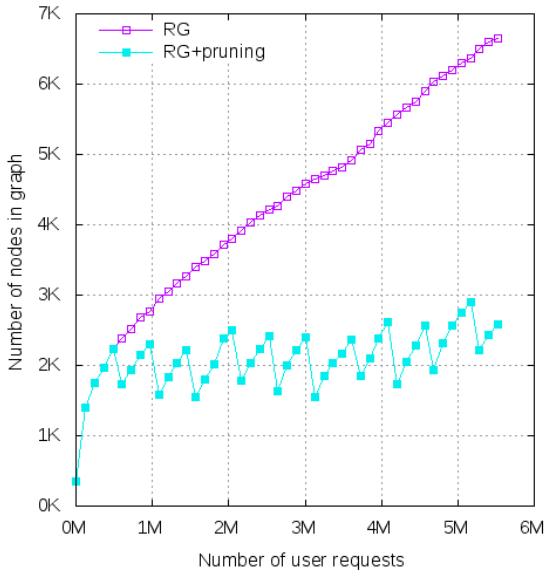



(a) Before pruning

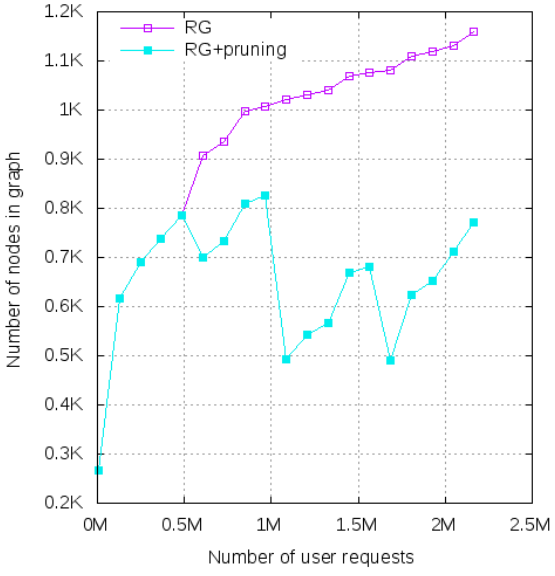



(b) Graph after arc and node pruning

Figure 7.10: Graph learnt and graph after pruning

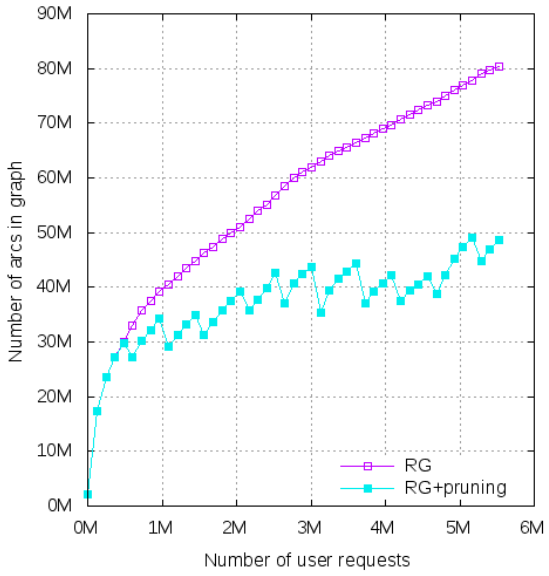



(a) Trace A

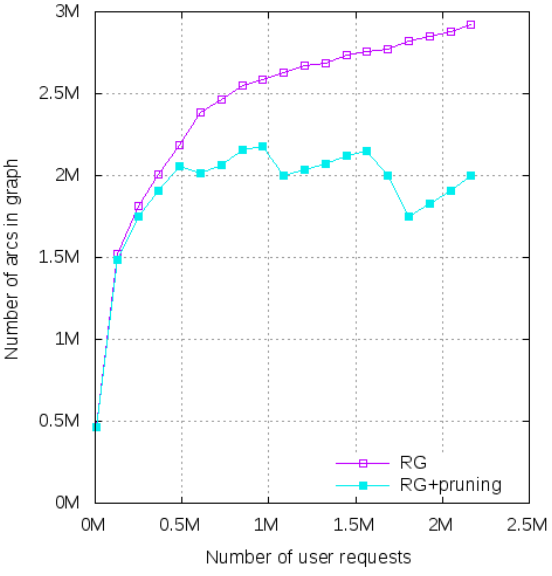



(b) Trace B

Figure 7.11: Number of nodes in graph when threshold is 0.1

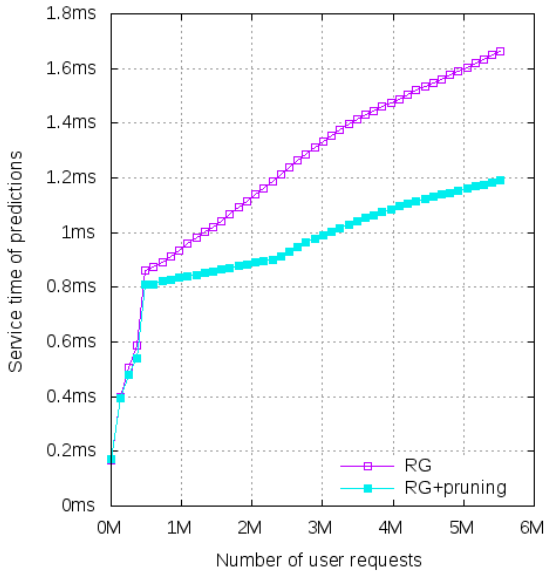



(a) Trace A

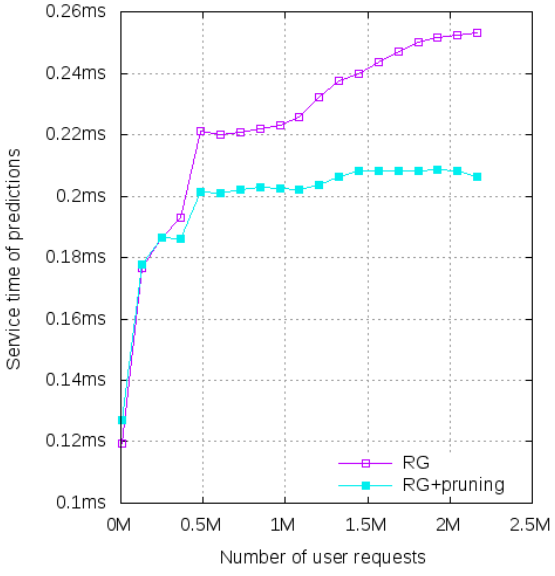



(b) Trace B

Figure 7.12: Number of arcs in graph when threshold is 0.1

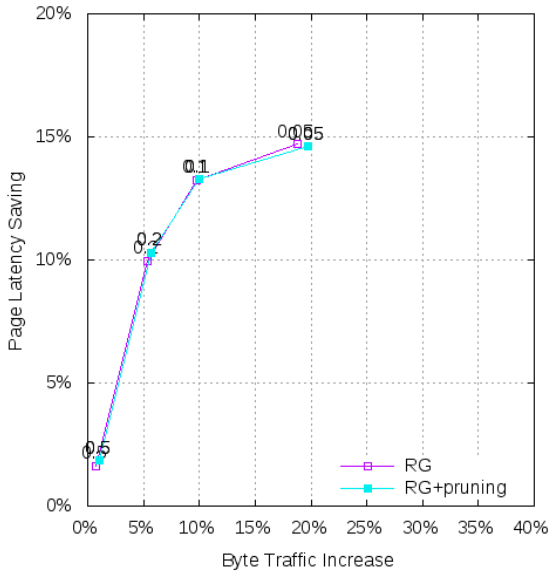



(a) Trace A

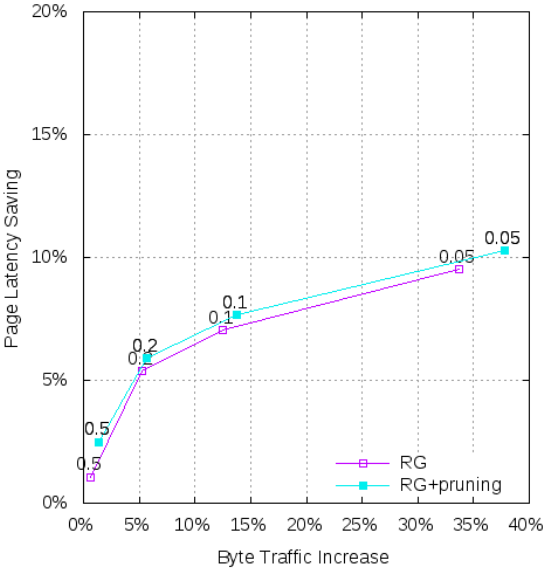



(b) Trace B

Figure 7.13: Prediction latency when threshold is 0.1




(a) Trace A




(b) Trace B

Figure 7.14: Page latency saving versus byte traffic increase

the inclusion of the pruning mechanism significantly reduces the service time, which drops by about 30% in trace A and 23% in trace B (quantified at the end of the experiment).

These results mean that graph pruning does not have any adverse effect on the service time; on the contrary, it contributes to noticeably reduce it, thanks to the reduction of the graph complexity.

Finally, Figure 7.14 shows the page latency saving and byte traffic increase obtained by the RG algorithm with and without pruning. In trace A, pruning the graph results in almost no change in the performance achieved. It is observed that in trace B, RG obtains slightly higher benefit when pruning, but at the expense of increasing the cost. This is because pruning the graph deletes nodes and arcs with low transition confidence, and this fact increases the transition confidence of the remaining arcs. As a consequence, more hints are provided in the predictions, more hints are prefetched, and some of them finally result in prefetching hits.

In summary, the proposed method for pruning the RG graph allows us to reduce computational (i.e., less service time) and storage resources (i.e., graph complexity) without decreasing the original performance.

## 7.5 Conclusions

This chapter has presented the Referrer Graph prediction algorithm and an associated prune mechanism, which is aimed to be a precise and simple solution for web prediction while consuming few computational resources. RG learns from user accesses and builds a Markov model, differentiating dependences on objects of the same page from objects of different pages. However, instead of using the sequence of user accesses as other algorithms do, RG uses the object URI and referrer associated to each request. This permits the design of a very simple algorithm because it does not need to keep track of previous user accesses (the user browsing session). It also means that RG establishes arcs to represent proven relations, instead of establishing arcs between objects for the circumstantial reason that they were requested sequentially by the same user. Consequently, RG establishes fewer arcs than other proposals, so the Markov model is smaller. This allows a faster management of the model when learning or predicting. Furthermore, when several browsing sessions use the same IP address, RG is not negatively affected, unlike the algorithms that use the sequence of accesses, which perform an erroneous learning of user patterns.

The experimental results show that RG, compared to the best prediction algorithms proposed in the open literature, obtains similar or even better page latency savings when the traffic increase is a strong constraint, but requiring less computational and memory resources, thanks to its data structure and algorithmic simplicity.

In addition to the learning and prediction processes, a simple mechanism for pruning nodes, arcs, and occurrences in the graph has been devised for RG. This allows RG to learn from new user accesses over time without increasing the resource consumption. Results show that the proposed pruning mechanism significantly reduces

the graph complexity and consequently the service time to perform a prediction, and has no adverse impact on the latency savings achieved, even improves them.

Some results of the work presented in this chapter were presented in [de la Ossa 10a] and submitted to a SCI Journal [de la Ossa 11] in March 2011.

# Chapter 8

# Conclusions and Future Work

## 8.1 Conclusions

This dissertation has demonstrated that web prefetching can work in a real scenario without requiring changes in the standard protocols. To this end, Delfos has been designed, developed, and tested as a practical implementation that interacts with existing software. We have described how the different elements of the web architecture interact in order to support web prefetching.

Delfos was then extended to support trace-driven experimentation, thus allowing the comparison of the results obtained by different prediction algorithms and prefetching techniques when using a specific load. We described the evaluation methodology, and the performance indexes that are measured by Delfos. These indexes include: i) those related to the prediction accuracy like precision and recall, ii) those related to cost-benefit like latency saving and traffic increase, and iii) those related to resource consumption like computation and memory consumption. Detailed experimental results were presented to show the usefulness of Delfos as an evaluation framework.

We proposed Predict at Prefetch, a technique to improve web prefetching performance by allowing the prediction engine to provide more hints to the client. Predict at Prefetch was implemented and evaluated experimentally using Delfos.

We considered the key factors that could limit web prefetching in a real environment. Then, using Delfos, we performed an empirical study to evaluate to what extent these factors could affect web prefetching, for which reasons, and which values are preferable.

The Referrer Graph prediction algorithm was proposed as a precise and simple solution for web prediction that consumes few computational resources. It outperforms other prediction algorithms. This algorithm uses the referrer information included in each web request, instead of using the sequence of user accesses. In this sense, RG is invulnerable to sequence alterations produced by elements like proxies, shared IP addresses in a LAN, etc.

We demonstrated that web prefetching can be implemented in the current web architecture without modifying the standard protocols, reduces the users perceived latency with a reasonable network traffic increase, and can work efficiently without requiring noticeable computational or memory consumption.

## 8.2 Summary of Contributions

The main contributions of this work can be summarized in:

- Development of Delfos, a framework that implements web prediction and web prefetching in a real scenario, interacting with existing web clients and servers.
- Development of an improvement of the framework for testing web prediction and prefetching techniques.
- Proposal of the Predict at Prefetch technique that helps to improve the performance of web prefetching by allowing the performance of predictions in more cases.
- Study on the theoretical limits affecting web prefetching when it is implemented in a real environment.
- Design of Referrer Graph, a new prediction algorithm that achieves similar results to more complex algorithms, but is simpler and requires less resource consumption.

## 8.3 Future Work and Open Research

Despite the mentioned contributions of this work, and as it occurs in a great number of PhD dissertations in the IT field, there are several open issues that can be planned as future work, for instance:

- To design a technique that allows the prefetch engine to notify the prediction algorithm about prefetch hits. This prefetch hit notification technique would permit the design of prediction algorithms that learn not only from web accesses, but also from prefetch hits, for example, increasing the weight of arcs for which a hit notification was provided. The objective is to increase the precision of the prediction algorithm.
- This research work assumed that the prefetching engine uses one single network connection to prefetch objects. This assumption takes into account the behaviour of existing commercial prefetching implementations, for instance, Mozilla web browsers. But it is technically possible to prefetch objects using two simultaneous connections, which could probably improve the prefetching

performance. Thus, we propose to experimentally evaluate how that change would influence the performance of the prefetching technique.

- To investigate what new possibilities the new HTML 5 standard brings to web prediction and prefetching techniques.

The experience obtained throughout this research work allows us to propose new research challenges:

- Taking into account that collaborative prefetching between different elements of the web architecture can achieve the highest values of latency savings [Domènech 06e], the design of a low cost prediction and prefetching mechanism able to work in a real web proxy is an interesting and promising research topic.

- In the research group, we are developing a research line that attempts to reduce the latency of displaying web pages using a pre-validation technique for web objects objects. Previous results of our research group [Domènech 10] show that the number of conditional requests to validate web objects in cache is very high and negatively affects the users perceived latency. Consequently, we have developed and patented a mechanism that includes, in previous requests, information about the freshness of the objects to be requested in the near future. This technique uses a prediction algorithm to guess next user accesses to assure the freshness of the predicted object in advance. A prediction algorithm that can accurately reconstruct the tree of references between web objects will be more appropriate for this purpose than the traditional ones. So, the RG algorithm can provide a good start point to improve the precision of the prediction and consequently to obtain better performance in this technique reducing the users perceived latency with a low computational cost.

# Appendix A

# *CARENA*

## A.1 Introduction

The most straightforward and accurate way to model a single user consists in instrumenting its own browser to collect the logs and other navigation characteristics. Furthermore, if the adequate information is collected, the gathered navigation session can be entirely and accurately replayed again. This fact enables that one gathered session can be replayed as many times as desired, like some workload generators do [RadView , Peña 04, Peña 05].

The user behavior characterization can also be done by analyzing the log files obtained either in a proxy or in a web server. These files collect information about the web users' activity. Nevertheless, they usually collect data for a large set of users, so clustering techniques must be applied to identify typical patterns.

We analyzed a representative set of developed tools to capture web client behavior information (both commercial and academic), and we found some shortcomings that make them not suitable for our purposes. As a consequence, we developed an open-source tool, called CARENA (CApture and REplay NAvigations). It is a browser extension that captures the HTTP headers and visualizes them in real time. Our tool gathers precise information about the user's navigation behavior and has been implemented on the top of Live HTTP Headers Mozilla extension [Savard 02]. The CARENA code has been added in a modular way, which eases its migration to other Live HTTP Headers releases. The main features of CARENA are: i) it captures the user logs and additional information related with the navigation session, ii) it saves the navigation session into a XML structured file, iii) it permits to import navigation sessions and, iv) currently, it permits to replay, in a precise way, a single navigation session since it accurately captures the user think times. The captured sessions could be used to feed specific tools such as workload generators. See [Peña 04, Peña 05] for further details.

The remainder of this appendix is organized as follows. Section A.2 discusses the main log information characteristics. Section A.3 discusses some related work. Section A.4 explains the details of the proposed tool. Section A.5 shows a CARENA working example.

## A.2 Logging web user requests

In order to obtain precise workload models, different timing points of the web user sessions should be identified and captured; e.g., the start of the document downloading. Log analysis can provide valuable information about the web user behavior, although the collected information varies depending on the network point where the logs are collected; e.g., the server, the proxy or the browser.

Servers' logs usually collect information for a large set of users; therefore they collect much more information than logs gathered for a single web browser. Discerning particular user behaviors from web servers logs requires clustering techniques and specific software for analysis. In addition, the use of search engines makes it difficult to distinguish between human and robot navigations.

Like server logs, proxy logs also collect information for a large set of users; therefore, they also present the drawback that this fact involves. These logs collect information from both the client proxy side and the network side, and have been extensively used for web caching and prefetching studies.

Finally, since browser logs collect navigation information for a single user, these logs are the most precise to characterize particular user's behavior. The main shortcoming is that monitoring browser navigations requires to instrument the browser code of the users that are part of the experiment, and to have the additional support for capturing, preparing and analyzing the extracted data by each browser. Despite all these advantages, few attempts have been made to instrument browsers because they do not usually offer open source code [Reeder 00].

## A.3 Related work

From the beginning of the WWW, researchers have concentrate on identifying the main features of user's navigations in order to characterize this workload, to detect user's behavior patterns, or for performance evaluation purposes. Despite this interest, only few tools have been developed to help researchers in these tasks. The first attempts were proposed by Catledge and Pitkow in [Catledge 95] and Tauscher and Greenberg in [Tauscher 97]. Both approaches instrumented the XMosaic Web browser. The main goal was to capture all user and browser events generated. Each activity record included timestamps, the visited URL, the page title, the final action, the invoking method and the user id, among other events.

In [Catledge 95], Catledge and Pitkow studied the user behavior in order to understand their strategies when navigating the Web; for this purpose, they analyzed the

log files from a client-side point of view. They calculated the time between each event for all events among the users, and determined session boundaries by analyzing these times, adopting the heuristic approach in which a lapse of 25.5 minutes or greater indicated the end of a "session." Their study concluded with the characterization of user navigation patterns as serendipitous browsing, general browsing and searcher.

In [Tauscher 97], Tascher and Greenberg studied the history mechanisms that web browsers use to manage the recently requested pages. Their main interest was the analysis of re-visit patterns in the navigations. As a result, they formulated some empirically-based principles of how users revisit pages using graphical browser features like the Xmosaic.

The study developed by Choo *et al.* [Choo 99] attempts to understand how the staff of seven different companies used the Web to seek information related to their daily work. To gather relevant information about WWW user navigations, they used a triangulation information approach, by collecting information from three sources (e.g. questionnaire survey, web usage logs and personal interviews). To obtain the user's logs they developed the tool called WebTracker. It is a typical Windows application that watches the browser and collects menu choices, button bar selections, keystroke actions, and mouse clicks. All these actions are associated with the open web page URL. The browser actions are recorded in an ASCII text file. By combining the information obtained from the recorded logs with those data extracted from the questionnaires and interviews, they were able to reconstruct the whole navigation process.

Reeder *et al.* [Reeder 00] developed a new tool called WebLogger, which captures and records a significant number of user and application events during the browsing session. These captures are documented into a log file at three conceptual levels, the Input level (e.g. user actions on the mouse or keyboard), Interface level (e.g. user actions on the interface elements of Internet Explorer (IE)) and Application level (e.g. high-level actions of IE, such as retrieving an URL). This tool works under Windows operating systems and was developed to interact with the Microsoft Internet Explorer (IE) web browser. The information recorded contains the event name, a list of specific parameters related with the event, the cumulative elapsed time since WebLogger was started (millisecond precision), the differential elapsed time since the previous recorded event (millisecond precision), and the current Windows system.

The HTTPLook [Sniffer ] and ieHTTPHeaders [Blunck 03] are shareware and freeware Internet tools respectively, developed to provide some information about HTTP headers. These tools only work on Windows OS. The HTTPLook [Sniffer ] sniffes the information transmitted between the client and the server. It captures and records information related with the web object type, the used technology, the IP address, messages arrival time and transmission time. ieHTTPHeaders [Blunck 03] is a bar developed for the IE web browser that allows to show the sent and received HTTP Headers. Since these two tools were not developed for research purposes, they do not include analysis or statistical capabilities.

Finally, a small set of web workload generators like [RadView ] and [LoadRunner ] capture and record the user logs in order to obtain accurate information to partially generate the workload. Both are proprietary tools.

After this thorough study, we can summarize that, in general, the output file format of the tools mentioned in [Sniffer , Blunck 03, Catledge 95, Tauscher 97, Reeder 00] is not so easy readable. Our tool, CARENA, solves this drawback because its output XML format contributes to the identification of all the elements that compose the navigation (e.g. frames, hidden frames, whole document, objects and attributes), and the log file can be transformed into other generic format, (plain text, rtf,...) or a specific one. This last feature has special interest in the GUERNICA workload generator [Peña 04, Peña 05].

The information captured is quite limited in most of the studied tools [Sniffer , Blunck 03, Catledge 95, Choo 99, Reeder 00]. In general, just some user and browser-specific events are captured (e.g. mouse click, keystroke action, open file action). In contrast, CARENA captures more relevant and accurate information about the network and the user; for instance: request and response HTTP headers size and time stamp, object size, user think time, status of whole document, document retrieval latency, start document load time stamp, end document load time stamp, etc.

None of the discussed tools establish an estimation of the user think time. In some web workload generators this time is usually taken as a constant [Peña 04] while others estimate it by statistical distributions [Peña 05]. Unlike these tools, CARENA carefully and accurately calculates this time.

In this scenario, our proposal provides important improvements and, at the same time, is presented as an open-source tool. In the next Section we present in detail the main architecture and features of CARENA.

## A.4 The CARENA Solution

CARENA is a tool for capturing web navigation sessions at the client browser. This tool is implemented as a web browser extension that can easily be installed on any Mozilla or Mozilla Firefox browser independently of the underlying operating system. With this tool the user is able to capture all the navigation session, including the page requests due to user clicks and their embedded objects. The navigation session is arranged around a XML structure and it can be later saved into a file, imported and replayed. Our work was done on top of Daniel Savard's Live HTTP Headers Mozilla extension [Savard 02]. Live HTTP Header is notified when Mozilla sends or receives HTTP requests or responses and captures the headers. It shows these HTTP headers on a window in real time, allowing to replay a single request, to edit it before replaying or to save the headers to a text file. In CARENA we added several new features to fulfill our requirements.

The main CARENA feature is that it retrieves additional information for each requested object. So, for each object CARENA not only retrieves its HTTP request and response headers, but also timestamps when the request is sent and the response

is received; it parses the HTTP headers to obtain the HTTP method and version, the response object size, the HTTP status messages, the object and referrer URLs and finally it estimates the object latency from the request and response timestamps. If the user interrupts the navigation by clicking on the Stop button, a status attribute indicative of this fact is added to the representative XML structure of the webpage and all the objects requested and not yet received.

The CARENA tool detects if an object was directly requested by the user or if it was an embedded object. Objects are grouped in document structures; each structure contains both the HTML object whose URL the user requested and all the objects retrieved due to the request. The information stored for every document includes, among others, the document URL, the number of retrieved objects, the time when the first object was requested and the time when the last object of the mass documents was received, the document latency, the document status indicating if any fail occurred, and the user think time. All this information can be saved to an XML structured file and imported at a later time. Those files can be easily treated or parsed by external tools since they are plain text files with XML structure.

Navigation sessions can be later replayed by emulating the original user behavior and requesting only the objects that the user really requested in its navigation session. Proceeding this way, Mozilla automatically retrieves all the embedded objects. The replay process takes into account the previously calculated user think time in order to defer the emulated user requests as in the capture session.

### A.4.1 Programming Environment

Mozilla can be used by developers as a platform for creating applications that can be installed locally or run remotely over the Internet. It is a powerful and easy platform development framework to develop cross-platform applications. One does not need to get involved with the Mozilla source code to create a Mozilla application. A simple Mozilla binary that you download and install is the only development platform needed.

Currently, an Integrated Development Environment (IDE) does not exist, but it is possible to use many tools that make it easy the application development. Some of these tools are: Venkman, DOM Inspector, XUL Maker, etc.

Venkman is a JavaScript debugger with support for breakpoints, local variable inspection, watch variables, single step, stop on error, code reformatting, etc. DOM Inspector is a tool for inspecting and editing the structure and widgets of the interface while the application is running. This was very helpful in the design and analysis of XML structures.

The Mozilla development framework is built around several technologies. Thanks to the combination of these technologies, Mozilla allows developers to create applications on top of it. Mozilla structure consists of two layers, as depicted in Figure A.1.

Gecko is the software component in Netscape, Mozilla and Mozilla-based browsers that handles the parsing of the HTML, the layout of the pages, the document

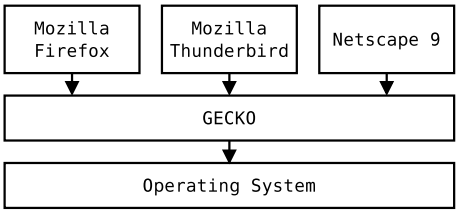



Figure A.1: Mozilla Structure

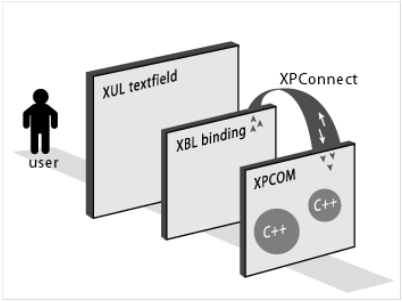



Figure A.2: XPCOM/XPConnect

object model, and even the rendering of the entire application interface. It is a fast, standards-compliant rendering engine that implements the W3C DOM standards. Mozilla Navigator, Mozilla Mail, etc., are written in languages such as XUL, JavaScript, and XPCOM; therefore, Gecko is the interpreter that executes them. Some advantages of this are: modularity, platform independence and that the components can be added or removed easily. XPCOM and XPConnect are complementary technologies that enable the integration of external libraries with XUL applications.

XPCOM, which stands for Cross Platform Component Object Model, is a framework for writing cross-platform, modular software. This means that it is a framework which allows developers to break up monolithic software projects into smaller pieces. These pieces, known as components, are then assembled back together at runtime. XPCOM components can be written in C, C++, and JavaScript, and they can be used from C, C++, and JavaScript. As an application, XPCOM uses a set of core XPCOM libraries to selectively load and manipulate XPCOM components [Parrish 01].

XPConnect is a technology which enables simple interoperation between XPCOM and JavaScript. XPConnect allows JavaScript objects to transparently access and manipulate XPCOM objects. It also enables JavaScript objects to present XPCOM compliant interfaces to be called by XPCOM objects. In other words, XPConnect is the bridge between JavaScript and XPCOM components, as shown in Figure A.2.

XPFE (cross-platform front end) [Boswell 02] was designed as a flexible interface working on any operating system. XPFE uses a number of existing web standards,

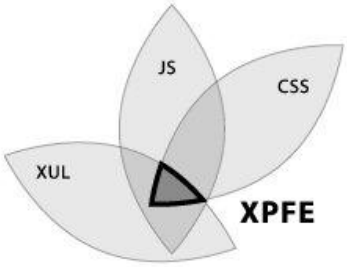



Figure A.3: Cross-Platform Front End

such as Cascading Style Sheets, JavaScript, and XML (the XML component is a new language called XUL, the XML-based User-interface Language). In its simplest form, XPFE can be seen as the union of each technology. As shown in Figure A.3. JavaScript creates the functionality for a Mozilla-based application, Cascading Style Sheets format the look and feel, and XUL creates the application structure. Thanks to the fact that XPFE is independent of the platform, the applications created with XPFE are also platform independent.

XPInstall, or Mozilla Cross Platform Install, provides a standard way of packaging XUL application components with an install script that Mozilla can download and execute. XPInstall enables users to effortlessly install new XUL applications over the Internet or from corporate intranet servers.

XUL (XML-based User-interface Language) is Mozilla's XML-based User interface Language, and it allows the building of feature-rich cross platform applications that can run connected or disconnected from the Internet. It creates the structure and content of an application. The XUL language defines attributes that allow the programmer to define how to react to the actions. To define the dynamic behavior of the application, one can define JavaScript functions that will be called when certain user interface events happen.

JavaScript is the core scripting language used in Mozilla. Three distinct levels of JavaScript are identified [Boswell 02], as shown in Figure A.4. The user interface level manipulates content through the DOM. The client layer calls on the services provided by XPCOM. JavaScript calls methods and gets data from scriptable components. Finally, the application layer is available to create XPCOM components.

JavaScript interface implementation is an API to access HTML and XML documents. It provides web developers with a structural representation of the documents and defines the way the structure is accessed by using JavaScript. Some methods like getElementById, getElementsByTagName, createElement, and createTextNode are provided by the DOM interface, which permits querying and handling documents.

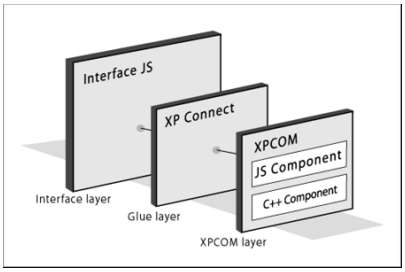



Figure A.4: Javascript Layers Architecture

The Observer Service allows a client listener to register and unregister for notifications of specific string referenced topics. Service also provides a way to notify registered listeners and a way to enumerate registered client listeners. The internal Mozilla notification system [Mozilla 05] helps to capture data. This system notifies the name of the event to the observer service, which deals with the lists of components watching for events. When the observer receives notification for an event, it passes that notification to all listening components for that event. With this technique, the system simplifies the way to react when another component triggers an action.

CARENA deals with the following Mozilla events: object request, object response, document start, document end and document failure. Mozilla notifies CARENA when any of these events occurs; then, CARENA processes and stores the information on the internal DOM structure and shows the headers in the CARENA window.

### A.4.2 Capturing

CARENA uses an XML object structure for the captured information. This structure is accessed trough the Mozilla DOM Core JavaScript interface implementation. HTTP request and response headers for a requested web object are stored into a JavaScript object that was created at the moment of the request. Headers are parsed to extract information like file size, referrer web object, etc., and stored as object's attributes. Timestamps are retrieved using the core JavaScript Date object method getTime that gets the system current local time.

When a web object is directly requested by the user, a new $< document >$ element is created into the XML structure, and a $< object >$ element is added as the first child. New $< object >$ element children are created as soon as new requests are issued as a result of the current user request. When an HTTP response is received, CARENA locates the object in the XML structure where the HTTP request was previously stored and then new data is added. This means that each time the user clicks on a hyperlink or on a bookmark, or types a new address into the address bar, a new $< document >$ element is created in the XML structure; and all the embedded objects of that web document will be added as children.

When Mozilla reports that the document loading process has concluded (either successfully or as a result of an error occurred in the reception of the document), new attributes are added to the document element. These attributes include the number of retrieved objects, the time when the last object of the document was completely received, the document latency, the document status indicating if some fail occurs or not, and the think time.

Frames and iframes are stored as document children, as happens with web objects but using a different XML tag. Since a document loaded on a frame or iframe can contain embedded web objects, frame elements can be parents of other objects.

When a HTTP response is a redirection (3XX status responses), we store the new URL for that object as an element attribute. If the browser decides to follow the redirection, the new request for the web object will be stored as a new object of the current $< document >$.

### A.4.3 Saving and Importing

Since we have the object requests, responses and additional information stored in a XML structure, we can save it into a file. This file contains all the information available in memory, including headers, URLs, referrers, timestamps, latency, and what is most important, the relation among objects, documents and frames.

The XML session file can be used by external tools to easily read, modify or even write their own navigation session files using existing XSL manipulation libraries. The XML file can be easily opened, read and parsed to retrieve all suitable information.

In order to import a saved session, the user must select the XML file to be imported in a dialog window. After cleaning the XML structure, the XML file is loaded and parsed in order to fill up a new XML structure. During parsing, a content model based on the XML DOM is built [Boswell 02]. Additionally the HTTP requests and responses are displayed in the headers window. Once the importing process concludes, all variables have exactly the same values they had when the file was firstly saved.

Session XML files can be easily edited and modified by other tools before importing. This allows to modify previous navigation sessions think times, headers, requests, etc., before replaying the session and comparing the results.

### A.4.4 Replying

The original Live HTTP Headers allow to select an HTTP header request, to modify it and send it again. In CARENA we added the possibility of replaying the entire navigation session, not only a single request. This new 'Replay All' feature repeats the page HTTP request headers previously stored in the XML structure. The same URLs that the user directly requested when the navigation session was captured are now sent. CARENA does not request the original embedded objects that were retrieved when capturing the session; this job is done by the web browser that will retrieve any embedded object as usual. If the original website has not changed since capturing,

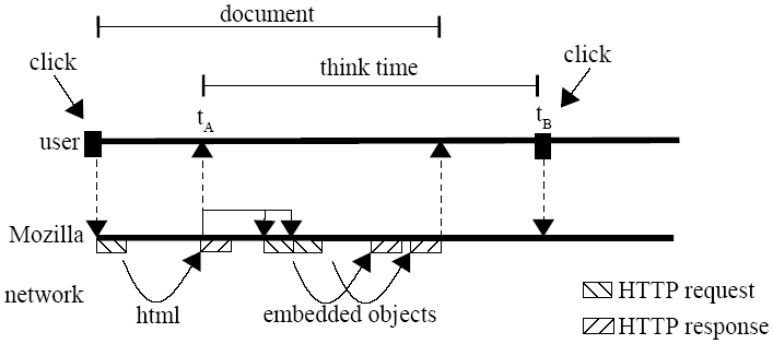



Figure A.5: Details of the user think time

then the embedded objects will be the same; otherwise, they will probably be different ones, depending on the changes that the website had had.

The original user think time is used to accurately replay the navigation session. This think time is usually a weakness in most current web workload generators; hence, it is usually simplified or taken as a constant. The user think time estimation is one of the main features of our tool, since it is precisely estimated in order to provide an accurate navigation replaying. As shown in Figure A.5, the original user think time is calculated as the difference between the captured values of tB and tA. tA stamps the time at which the header part of the response arrives and the web page begins to render on the user screen, and tB represents the time at which the user submits a request for a new page. We could have calculated the think time as the difference between the time when the last object of a document is completely received and the next click, but due to the nature of incremental rendering of current browsers we selected the method explained before.

The think time calculation may deviate in some situations. Some factors that may affect the precision are the file size, the number of embedded objects, the relevance of embedded objects, the user previous knowledge of the content, and the position of the relevant links in the webpage.

When replaying navigations, many website or network failures can arise; for instance, pages requested in original sessions may not exist in future sessions. In order to prevent these kinds of failures, we use a timeout. When the timeout expires, the tool assumes that the page is unavailable, therefore it executes the next request.

It is also possible to clear the browser disk and memory caches before starting the replay, in order to make it even more accurate.

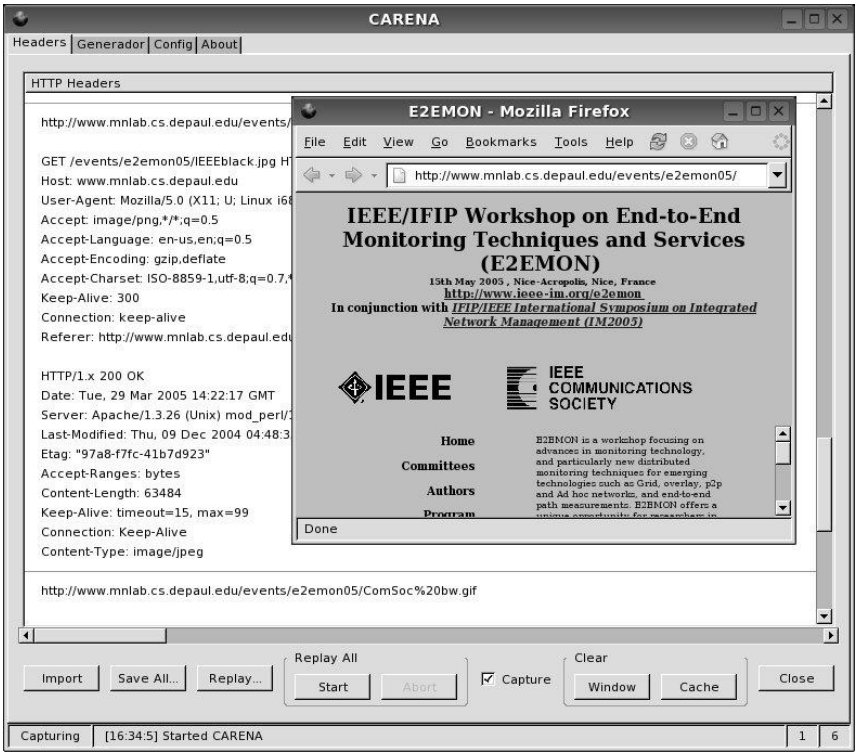



Figure A.6: Navigating in Mozilla while CARENA captures the session

## A.5 Working Example

Using CARENA to capture navigation sessions is extremely easy. It is started through the "Tools - Web developer" Mozilla menu. Once the Mozilla web browser and CARENA are open, any object the web browser requests to the network will be captured by our tool.

Figure A.6 shows an example while retrieving an example website. HTTP headers are shown on the tool main window while browsing as soon as objects needed to compose the page are being retrieved.

By using the "Save All" button, the navigation session can be saved to an XML structured file that contains all the gathered information, including HTTP headers, timestamps and additional information. Figure A.7 illustrates a navigation session where the user has requested three documents, as shown in the example. The first document requested consists of three frames, and the first object of this document is the frame definition document while each document loaded in each frame appears into the $< frame >$ label.

Figure A.7: Part of the XML structured file

```
<?xml version="1.0"?>
<CARENA>
  <document url="http://www.mnlab.cs.depaul.edu/events/e2emon05/"
            numObjects="4"
            timeStart="1108463215920"
            timeEnd="1108463217436"
            docLatency="1516"
            docStatus="EndDocumentLoad"
            docThinkTime="11016">
    <object url="http://www.mnlab.cs.depaul.edu/events/e2emon05/"
            requestT="1108463215920"
            status="HTTP/1.x 200 OK"
            usercl="1"
            loadFlags="589824"
            method="GET"
            version="1.1"
            size="621"
            referrer="undefined"
            responseT="1108463216233"
            latency="313">
      <request requestL="423">
        <reqHeader>GET /events/e2emon05/ HTTP/1.1</reqHeader>
        <reqHeader>Host: www.mnlab.cs.depaul.edu</reqHeader>
        ...
        <reqHeader>Connection: keep-alive</reqHeader>
      </request>
      <response responseL="296">
        <respHeader>HTTP/1.x 200 OK</respHeader>
        ...
        <respHeader>Content-Type: text/html</respHeader>
      </response>
    </object>
    <frame url="http://www.mnlab.cs.depaul.edu/events/e2emon05/top.htm"
           numObjects="3">
      <object url="http://www.mnlab.cs.depaul.edu/events/e2emon05/top.htm"
              requestT="1108463216249"
              referrer="http://www.mnlab.cs.depaul.edu/events/e2emon05/"
              status="HTTP/1.x 200 OK"
              usercl="0"
              loadFlags="65536"
              method="GET"
              version="1.1"
              size="1604"
              responseT="1108463216405"
              latency="156">
        <request requestL="488">
          <reqHeader>GET /events/e2emon05/top.htm HTTP/1.1</reqHeader>
          ...
        </request>
        <response responseL="296">
          <respHeader>HTTP/1.x 200 OK</respHeader>
          ...
        </response>
      </object>
      ...
    </frame>
  </document>
  <document url="http://www.mnlab.cs.depaul.edu/events/e2emon05/dates.htm"
            numObjects="1"
            timeStart="1108463231405"
            docFromFrame="http://www.mnlab.cs.depaul.edu/events/e2emon05/left.htm"
            timeEnd="1108463231561"
            docLatency="156"
            docStatus="">
    <object url="http://www.mnlab.cs.depaul.edu/events/e2emon05/dates.htm"
            requestT="1108463231405"
            referrer="http://www.mnlab.cs.depaul.edu/events/e2emon05/left.htm"
            status="HTTP/1.x 200 OK"
            usercl="1"
            loadFlags="589824"
            method="GET"
            version="1.1"
            size="824"
            responseT="1108463231561"
            latency="156">
     ...
     </object>
  </document>
</CARENA>
```

Table A.1: Deviations from the original session

| Deviation type | Repetition number | URI | | |
|---|---|---|---|---|
| | | /index.htm | /organizers.htm | /dates.htm |
| docLatency | 1 | 0.00% | −30.13% | −30.13% |
| | 2 | 9.30% | −40.38% | −30.13% |
| | 3 | 8.31% | −30.77% | −19.87% |
| | 4 | 10.36% | −40.38% | −30.13% |
| | 5 | 9.30% | −40.38% | −30.13% |
| docThinkTime | 1 | −0.42% | 0.54% | |
| | 2 | −0.71% | 3.93% | |
| | 3 | −0.99% | 0.57% | |
| | 4 | −1.70% | 1.11% | |
| | 5 | −0.99% | 3.93% | |

The "Import" button can be used to retrieve a saved session. And the replay of a single request can be achieved by selecting it from the main window and by clicking on the "Replay" button. The "Replay All" button allows replaying a whole session. The session replied can be the one being captured or a previously saved one. Table A.1 summarizes five repetitions of a session retrieving three documents, showing the deviations of the latency and think time from the original session. It reveals that while latencies suffer great deviations due to the instantaneous network and server load, the think time is more accurate, because it only depends on the load of the user machine.

## A.6 Conclusions

In this appendix we have proposed a tool to help web performance evaluation studies through a Mozilla extension that captures and replays browsing sessions. Our tool not only measures time-related variables but also permits us to accurately replay complete navigation sessions.

The capture functionality can be used to get accurate data about the client workload, to characterize the user think time, or to know about the structure of the visited web pages, such as the number, type, size, etc. of embedded objects. By analyzing the captured navigation sessions, a complete Web taxonomy can be performed. For instance: i) HTTP headers patterns and frequencies, ii) Response times from the client point of view, considering the whole web page or each web object individually, iii) web structure and content; for example, number of objects per page, object and page size, full page transmission time, and reutilization of objects in different parts of the website, iv) web usage; that is, analyzing how people access web pages and use them in order to obtain the navigational behavior and also to discover the user access patterns, browser patterns, re-visit frequency of web pages and user think time. With

the above results, suggestions about design structure and usability of WWW pages, sites and browser can be made.

The replay functionality can be useful as a load generator for comparison purposes in a wide range of web performance research projects; for instance, proxy management algorithms or prefetching algorithms. It can also be used to automate functionality test processes of web applications.

The output file characterizes a real web navigation session, but other applications can be proposed. For instance: i) importing and replaying synthetically generated navigation sessions, ii) importing previous navigation sessions and replaying them in different network environments or conditions in order to compare the impact of those differences, iii) importing a modified navigation session and replaying it with the same network conditions in order to compare the impact of that modification.

The size of the navigation session file depends on the number of browsed pages, the number of embedded objects they contain, and how many of them are already cached on the browser. For instance, a page with 13 objects generates around 20 kilobytes, and a page with 40 objects generates around 60 kilobytes (an object approximately takes up to $1,500$ bytes on XML formatted file).

With respect to the overhead introduced by CARENA, the CPU time is negligible with respect to the one used by Mozilla; the memory needed is barely noticeable since CARENA code contains less than $2,000$ of JavaScript lines, and most of the requirements (i.e., XPCOM, XPFE, XUL, DOM, JavaScript machine) are already loaded and used by Mozilla for browsing purposes. Furthermore, memory consumption for data structures is not a problem when navigation sessions are shorter than several hours.

# Bibliography


[Albrecht 99] D. Albrecht, I. Zukerman & A. Nicholson. *Pre-Sending documents on the WWW: A comparative study.* Proc. of the 16th International Joint Conference on Artificial Intelligence, Stockholm, Sweden, 1999.

[Armstrong 93] J. Armstrong, M. Williams, R. Virding & C. Wilkström. Erlang for concurrent programming. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Armstrong 07] J. Armstrong. Programming erlang: Software for a concurrent world. Pragmatic Bookshelf, 2007.

[Balamash 07] A. Balamash, M. Krunz & P. Nain. *Performance analysis of a client-side caching/prefetching system for Web traffic.* Comput. Netw., vol. 51, no. 13, pages 3673–3692, 2007.

[Bestavros 95] A. Bestavros. *Using speculation to reduce server load and service time on the WWW.* Proc. of the 4th ACM International Conference on Information and Knowledge Management, Baltimore, USA, 1995.

[Bestavros 96] A. Bestavros & C. Cunha. *Server-initiated Document Dissemination for the WWW.* IEEE Data Engineering Bulletin, 1996.

[Blunck 03] Jonas Blunck. *ieHTTPHeaders http://www.blunck.info/iehttpheaders.html*, 2003.

[Boswell 02] David Boswell & Brian King. Creating applications with mozilla. O'Reilly Editors, September 2002.

[Bouras 04] C. Bouras, A. Konidaris & D. Kostoulas. *Predictive prefetching on the web and its potential impact in the wide area.* World Wide Web: Internet and Web Information Systems, 7, Kluwer Academic Publishers, The Netherlands, vol. 7, no. 2, pages 143–179, 2004.

[Catledge 95] Lara D. Catledge & James E. Pitkow. *Characterizing Browsing Strategies in the World-Wide Web.* Computer Networks and ISDN Systems, vol. 27, pages 1065–1073, 1995.

[Changa 08] T. Changa, Z. Zhuangb, A. Velayuthamc & R. Sivakumara. *WebAccel: Accelerating Web access for low-bandwidth hosts.* Computer Networks, vol. 52, no. 11, pages 2129–2147, 2008.

[Choo 99] Chun Choo, Brian Deltor & Don Turnball. *Information Seeking on the Web: An Integrated Model of Browsing and Searching, Web Tracker: A Tool for Understanding Web Use.* Asis Annual Meeting Contributed paper, 1999.

[Crovella 98] Mark Crovella & Paul Barford. *The network effects of prefetching.* In Proceedings of the IEEE INFOCOM'98 Conference, San Francisco, USA, 1998.

[Davison 02] B. D. Davison. *Predicting web actions from HTML content.* Proc. of the 13th ACM Conference on Hypertext and Hypermedia, College Park, USA, 2002.

[de la Ossa 04] B. de la Ossa, I. J. Niño, J. A. Gil, J. Sahuquillo & A. Pont. *A Tool to Capture and Replay Web Navigation Sessions.* Second International Workshop Conference on Performance Modelling and Evaluation on Heterogeneous Networks (HETNETs-04), Ilkley, United Kingdom, 2004.

[de la Ossa 06] B. de la Ossa, J. A. Gil, J. Sahuquillo & A. Pont. *Prefetching next web user's accesses in a real environment.* Proc. of the XVII Jornadas de Paralelismo (JP 2006), 2006.

[de la Ossa 07a] B. de la Ossa, J. A. Gil, J. Sahuquillo & A. Pont. *Delfos: the Oracle to Predict Next Web User's Accesses.* Proc. of IEEE 21st International Conference on Advanced Information Networking and Applications, Canada, 2007.

[de la Ossa 07b] B. de la Ossa, J. A. Gil, J. Sahuquillo & A. Pont. *Improving Web Prefetching by Making Predictions at Prefetch.* Proc. of the 3rd EURO-NGI Conference on Next Generation Internet Networks Design and Engineering for Heterogeneity (NGI'07), pages 21–27, 2007.

[de la Ossa 07c] B. de la Ossa, J. A. Gil, J. Sahuquillo & A. Pont. *Web Prefetch Performance Evaluation in a Real Environment.* Proc. of the IFIP/ACM Latin America Networking Conference 2007 (LANC 2007), XXXIII Conferencia Latinoamericana de Informática (CLEI 2007), pages 65–73, 2007.

[de la Ossa 07d] B. de la Ossa, A. Pont, J. Sahuquillo & J.A. Gil. *Supplying predictions with prefetch responses to efficiently reduce web user's perceived latency.* Proc. of the XVIII Jornadas de Paralelismo (JP 2007), pages 389–396, 2007.

[de la Ossa 09a] B. de la Ossa, J. Sahuquillo, A. Pont & J. A. Gil. *An Empirical Study on Maximum Latency Saving in Web Prefetching.* Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'09), pages 556–559, 2009.

[de la Ossa 09b] B. de la Ossa, J. Sahuquillo, A. Pont & J. A. Gil. *Upper Bound of Web Page Latency Saving in an Experimental Prefetching System.* Proceedings of the XX Jornadas de Paralelismo (JP 2009), pages 587–592, 2009.

[de la Ossa 10a] B. de la Ossa, A. Pont, J. Sahuquillo & J. A. Gil. *Referrer Graph: a Low-cost Web Prediction Algorithm.* Proc. of the 25th Symposium On Applied Computing, pages 831–838, 2010.

[de la Ossa 10b] B. de la Ossa, J. Sahuquillo, A. Pont & J. A. Gil. *Key Factors in Web Latency Savings in an Experimental Prefetching System.* Journal of Intelligent Information Systems, 2010.

[de la Ossa 11] B. de la Ossa, J. A. Gil, J. Sahuquillo & A. Pont. *Referrer Graph: a cost-effective algorithm and pruning method for predicting web accesses.* International Journal of Computer and Telecommunications Networking, 2011.

[Deshpande 04] M. Deshpande & G. Karypis. *Selective Markov models for predicting Web page accesses.* ACM Trans. Internet Technol., vol. 4, no. 2, pages 163–184, 2004.

[Domènech 04a] J. Domènech, A. Pont, J. Sahuquillo & J. A. Gil. *An experimental framework for testing web prefetching Techniques.* Proc. of the 30th Euromicro Conference, Rennes, France, 2004.

[Domènech 04b] J. Domènech, J. Sahuquillo, J. A. Gil & A. Pont. *About the heterogeneity of web prefetching performance key metrics.* Proc. of the 2004 International Conference on Intelligence in Communication Systems (INTELLCOMM 04), Bangkok, Thailand, 2004.

[Domènech 05] J. Domènech, J. Sahuquillo, A. Pont & J. A. Gil. *How current web generation affects prediction algorithms performance.* Proc. of the SoftCOM 2005 International Conference on Software, Telecommunications and Computer Networks, Split, Croatia, 2005.

[Domènech 06a] J. Domènech, J. A. Gil, J. Sahuquillo & A. Pont. *DDG: Efficient Prefetching Algorithm for Current Web Generation.* 1st IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), Boston, 2006.

[Domènech 06b] J. Domènech, J. A. Gil, J. Sahuquillo & A. Pont. *Web prefetching performance metrics: A survey.* Performance Evaluation, vol. 63, no. 9-10, pages 988–1004, 2006.

[Domènech 06c] J. Domènech, A. Pont, J. Sahuquillo & J. A. Gil. *A Comparative Study of Web Prefetching Techniques Focusing on User's Perspective.* IFIP International Conference on Network and Parallel Computing (NPC 2006), 2006.

[Domènech 06d] J. Domènech, A. Pont, J. Sahuquillo & J. A. Gil. *Cost-Benefit Analysis of Web Prefetching Algorithms from the User's Point of View.* Proc. of the 5th International IFIP Networking Conference, Coimbra, Portugal, 2006.

[Domènech 06e] J. Domènech, J. Sahuquillo, J. A. Gil & A. Pont. *The Impact of the Web Prefetching Architecture on the Limits of Reducing User's Perceived Latency.* Proc. of the International Conference on Web Intelligence, 2006.

[Domènech 10] J. Domènech, J. A. Gil, J. Sahuquillo & A. Pont. *Speculative Validation of Web Objects for Further Reducing the User-Perceived Latency.* In Proc. of the 9th International IFIP Networking Conference, pages 239–250, Chennai, India, 2010.

[Dongshan 02] Xing Dongshan & Shen Junyi. *A New Markov Model For Web Access Prediction.* Computing in Science and Engineering, vol. 4, no. 6, pages 34–39, 2002.

[Dornfest 05] Rael Dornfest. *Google Web Accelerator considered overzealous, article at O'Reilly*, 2005.

[Duchamp 99] D. Duchamp. *Prefetching Hyperlinks.* Proc. of the 2nd USENIX Symposium on Internet Technologies and Systems, 1999.

[Fan 99] L. Fan, P. Cao, W. Lin & Q. Jacobson. *Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance.* Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling Of Computer Systems, pages 178–187, 1999.

[Fielding 97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk & T. Berners-Lee. *HTTP/1.1 http://tools.ietf.org/html/rfc2068*, 1997.

[Fisher 03a] D. Fisher & G. Saksena. *Link prefetching in Mozilla: A Server driven approach.* Proc. of the 8th International Workshop on Web Content Caching and Distribution (WCW 2003), 2003.

[Fisher 03b] Darin Fisher. *Link prefetching in Mozilla FAQ*, 2003.

[Google 05] Google. *Google Web Accelerator, http://webaccelerator.google.com/*, 2005.

[Gündüz 03] Sule Gündüz & M. Tamer Özsu. *A Web page prediction model based on click-stream tree representation of user behavior.* Proc. of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining, 2003.

[Ibrahim 00] T. Ibrahim & C. Xu. *Neural Nets Based Predictive Pre-Fetching to Tolerate WWW Latency.* Proc. of the 20th IEEE International Conference on Distributed Computing Systems, Taipei, Taiwan, 2000.

[Kokku 03] R. Kokku, P. Yalagandula, A. Venkataramani & M. Dahlin. *NPS: A Non-Interfering Deployable Web Prefetching System.* Proc. of the USENIX Symposium on Internet Technologies and Systems, 2003.

[Kroeger 97] T. M. Kroeger, D.E. Long & J. C. Mogul. *Exploring the Bounds of Web Latency Reduction from Caching and Prefetching.* Proc. of the 1st USENIX Symposium on Internet Technologies and Systems, 1997.

[LoadRunner ] LoadRunner. *HP LoadRunner http://www.mercury.com/us/products/performance-center/loadrunner/.*

[Markatos 98] E. Markatos & C. Chronaki. *A Top-10 Approach to Prefetching on the Web.* Proc. of INET, Geneva, Switzerland, 1998.

[Mozilla 05] Mozilla. *XPCOM https://developer.mozilla.org/en/XPCOM*, 2005.

[Nanopoulos 03] Alexandros Nanopoulos, Dimitrios Katsaros & Yannis Manolopoulos. *A Data Mining Algorithm for Generalized Web Prefetching.* IEEE Trans. Knowl. Data Eng., vol. 15, no. 5, pages 1155–1169, 2003.

[Niño 05] I. J. Niño, B. de la Ossa, J. A. Gil, J. Sahuquillo & A. Pont. *CARENA: A Tool to Capture and Replay Web Navigation Sessions.* Proc. of the Third IEEE/IFIP Workshop on End-to-End

Monitoring Techniques and Services (E2EMON'05), Nice, France, 2005.

[Padmanabhan 96] V. Padmanabhan & J. C. Mogul. *Using Predictive Prefetching to Improve World Wide Web Latency.* Proc. of the ACM SIGCOMM Conference, 1996.

[Palpanas 99] T. Palpanas & A. Mendelzon. *Web Prefetching Using Partial Match Prediction.* Proc. of the 4th International Web Caching Workshop, San Diego, USA, 1999.

[Parrish 01] Rick Parrish. An introduction to xpcom. Developer Works, SOA and Web Services, IBM, 2001.

[Peña 04] Raúl Peña, Julio Sahuquillo, Ana Pont & José Antonio Gil. *Modeling users' dynamic behavior in web application environments.* 2nd International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks, 2004.

[Peña 05] Raúl Peña, Julio Sahuquillo, Ana Pont & José Antonio Gil. *Modeling users' dynamic behavior in e-business environments using Navigations.* International Journal of Electronic Business, 2005.

[Rabinovich 02] Michael Rabinovich & Oliver Spatscheck. Web caching and replication. Addison-Wesley, 2002.

[RadView ] RadView. *RadView WebLOAD: Recording WebLOADTM Agendas. http://www.radview.com.*

[Reeder 00] Robert W. Reeder, Peter Pirolli & Xerox PARC Tech. Stuart K. Card. *WebLogger: A Data Collection Tool for Web-use Studies.* Report UIR-R-2000-06, 2000.

[Savard 02] Daniel Savard. *LiveHTTPHeaders http://livehttpheaders.mozdev.org/*, 2002.

[Schechter 98] S. Schechter, M. Krishnan & M. D. Smith. *Using Path Profiles to Predict HTTP Requests.* Proc. of the 7th International World Wide Web Conference, 1998.

[Sniffer ] HTTP Sniffer. *Sniffer HTTPLook http://www.httpsniffer.com/.*

[Tauscher 97] Linda Tauscher & Saul Greenberg. *Patterns of Revisitation in World Wide Web Navigation.* Proc. of the SIGCHI conference on Human factors in computing systems, 1997.

[Teng 05] W. Teng, C. Chang & M. Chen. *Integrating Web Caching and Web Prefetching in Client-Side Proxies.* IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 5, pages 444–455, 2005.

[Yang 03] Qiang Yang, Joshua Zhexue Huang & Michael Ng. *A data cube model for prediction-based web prefetching.* Journal of Intelligent Information Systems, vol. 20, no. 1, pages 11–30, 2003.

[Zhang 03] Wei Zhang, David B. Lewanda, Christopher D. Janneck & Brian D. Davison. *Personalized Web Prefetching in Mozilla.* Rapport technique LU-CSE-03-006, Dept. of Computer Science and Engineering, Lehigh University, Bethlehem, USA, 2003.

[Zhu 02] Jianhan Zhu, Jun Hong & John G. Hughes. *Using Markov Chains for Link Prediction in Adaptive Web Sites.* In ACM SIGWEB Hypertext. Springer, 2002.

[Zukerman 99] I. Zukerman, D. W. Albrecht & A. E. Nicholson. *Predicting users' requests on the WWW.* Proc. of the seventh international conference on User modeling, pages 275–284, 1999.