**Delfos: the Oracle to Predict Next Web User’s Accesses**

B. de la Ossa, J. A. Gil, J. Sahuquillo and A. Pont  
Department of Computer Engineering, Polytechnic University of Valencia  
Camino de Vera, s/n, 46022 Valencia (Spain)  
berospe@doctor.upv.es, {jagil, jsahuqui, apont}@disca.upv.es

---

**Abstract**—Despite the wide and intensive research efforts focused on web prediction and prefetching techniques aimed to reduce user’s perceived latency, few attempts to implement and use them in real environments have been done, mainly due to their complexity and supposed limitations that low user available bandwidths imposed few years ago. Nevertheless, current user bandwidths open a new scenario for prefetching that becomes again an interesting option to improve web performance. This paper presents Delfos, a framework to perform web predictions and prefetching on a real environment that tries to cover the existing gap between research and praxis. Delfos is integrated in the web architecture without modifying the standard HTTP 1.1 protocol, and acts inserting predictions in the web server side, while prefetches are carried out by the client. In addition, it can be also 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.

**Index Terms**—Web prediction, web prefetching, performance evaluation.

---

**I. INTRODUCTION**

A LOT of research effort has concentrated on techniques to improve the World Wide Web performance. Web caching and prefetching have been proposed to reduce user’s perceived latency. While caching techniques are widely used in real environments, prefetching has been an interesting subject for research but few attempts to use in real environments can be found.

Prefetching techniques allow a web browser to request an object before the user asks for it. Obviously, the web browser must prefetch using accurate information in order to achieve reasonable performance that justifies the additional resources consumed (bandwidth, extra server load).

Prefetching can be done either by the web browser or by a proxy. To this end, predictions need to be provided in order to select the objects to prefetch. These predictions can be performed by the web server as described in most research works, but they can also be done by the web browser [1] or by an intermediate, e.g., a proxy [2].

The limitations on the available user’s bandwidth constrained the benefits of prefetching in the past. This fact together with the difficulty of implementing these techniques without introducing changes in the current protocols have introduced a gap between academic results and available products. But the current user’s bandwidth opens again new possibilities for prefetching to improve web performance.

This paper presents **Delfos**, a predicting framework integrated in a real web architecture, (i.e., the web server and the web browser). Delfos is the Spanish name of Delphi, the famous oracle of Apollo perched on the sides of Mt. Parnassos where ancient Greeks went to know the future. It implements prediction which is used to perform prefetching with no modification in the HTTP protocol, making it suitable to use with current browsers, web servers and protocols. **Delfos** considers predictions done on the web server side and prefetchs done on the client side, as current Gecko-based web browsers do. It also provides detailed statistic reports which permit to evaluate the performance of either the prediction engine, the prefetching engine or both, helping in the design of new and more efficient algorithms and structures.

Intensive research work has been published focusing on prediction and prefetching algorithms, but few of them include performance comparison results among the different proposals by using simulation or emulation tools. The main advantage of using these tools is their flexibility and speed 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. **Delfos** also covers this lack in web research topics.

In summary, the main contributions of this work are: the design and implementation of a framework to perform efficiently and easily web prefetching techniques, and to provide a flexible tool to evaluate and compare the performance of these techniques under real conditions.

The remainder of this paper is organized as follows. Section II describes the related work. Section III presents and gives details of our proposed framework. Section IV presents some experiments and working examples using **Delfos**. Finally, section V presents the concluding remarks.

**II. RELATED WORK**

In this section we make a brief review of some previous attempts to implement web prefetching techniques or to evaluate their performance.
Below we discuss a representative subset of software products with certain ability to perform web prediction or prefetching. We grouped software products in three categories: servers, proxies and clients, as summarized on Table I. Regarding web servers, only three products were found.

Kokku et al. [3] 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 neither in the web browser nor 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, as it 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. [4] propose a free available framework for prefetching, it is an 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.

There are several web proxies with prediction and prefetching capabilities. Half of them (Wcol and AllegroSurf) prefetch all the hyperlinks of a html document, unnecessarily wasting bandwidth. This massive and indiscriminate prefetching can be problematic, and has been criticized by system administrators, web designers and users [5]. No information is available about the prediction algorithms used in the other proxies, which leads to consider they use a similar method. Packeteer SkyX Accelerator is a gateway designed to accelerate connections in the local network using an undisclosed prefetching method. 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 do: prefetch 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). We use Mozilla Firefox in our experiments since it already implements all the required features regarding prefetching, it is widely used by both casual and expert users, it is published with a free and open source license and its full source code is also freely available.

Google Web Accelerator [6] is a free web browser extension available for Mozilla Firefox and Microsoft Internet Explorer on Microsoft Windows operating systems. It includes, among other features, web prefetching. It prefetchs hints included in the HTML body, but also prefetchs all the links in the pages being visited, even if no hints are provided.

FasterFox is an open and free extension for Mozilla web browsers that prefetchs all of the links on the current page during idle time.

PeakJet is a commercial product for the end user that includes several tools to improve the user access to the web. It includes a web browser independent cache with prefetching capability, based either on history or on links, therefore it can prefetch links on the current web page that were visited by the user sometime in the past or all the links on the current web page.

Another commercial product for the end user that prefetchs all the links in the page being visited, and store the objects in the browser cache is NetAccelerator. It includes the possibility to refresh the cache content in order to avoid obsolete objects.

Wei Zhang et al. [7] 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) [8]. In the case this one provides few hints, another algorithm based on the page content is additionally used.

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

Regarding to prediction algorithms, different proposals can be found in the research literature. They can be classified depending on the type of information gathered and the data structure used for the prediction: object popularity [9], Markov models [10]–[12], web structure [13]–[15], Prediction by Partial Matching [2], [8], [16], data mining [17] and genetic algorithms [18], [19].

But few research works have been addressed to compare the performance between different prediction algorithms mainly because the difficulty to reproduce environments and workloads [1]. Two algorithms based on Markov models, proposed by Zukerman [12] and by Bestavros [10], are compared in [20]. The comparison is only performed at the algorithmic level, without considering details related to the latency perceived by the user. Another work [21] compares two algorithms, one based on the idea of popular objects of Markatos [9] and another based on a variation of Prediction by Partial Matching. These comparisons were done from the point of view of the prediction and its precision, and to the knowledge of the authors there is only a fair attempt to compare them from the user’s perspective [22].

III. Delfos PROPOSAL

Delfos is a framework to perform prefetching in a real system. Because its flexibility, it can also be used to develop,
TABLE I
Known software with prediction or prefetching capabilities

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Google Search</td>
<td>Search results often embed hints on HTML</td>
</tr>
<tr>
<td></td>
<td>JosephDom</td>
<td>Benchmarking framework (emulator-simulator)</td>
</tr>
<tr>
<td></td>
<td>Wcol</td>
<td>Prefetchs all links</td>
</tr>
<tr>
<td></td>
<td>Squid-prefetch</td>
<td>Prefetchs all links (small Perl script)</td>
</tr>
<tr>
<td>Proxy</td>
<td>AllegroSurf</td>
<td>Prefetchs all links</td>
</tr>
<tr>
<td></td>
<td>Paketeer SkyX Accelerator</td>
<td>Prefetchs links</td>
</tr>
<tr>
<td></td>
<td>Robtex Viking Server</td>
<td>Prefetchs links</td>
</tr>
<tr>
<td></td>
<td>Mozilla</td>
<td>Used in our experiments</td>
</tr>
<tr>
<td></td>
<td>Google Web Accelerator</td>
<td>Prefetchs all links in HTML</td>
</tr>
<tr>
<td>Client</td>
<td>FasterFox</td>
<td>Prefetchs all links in HTML</td>
</tr>
<tr>
<td></td>
<td>PeakJet 2000</td>
<td>Prefetchs all links or only previously visited</td>
</tr>
<tr>
<td></td>
<td>NetAccelerator</td>
<td>Prefetchs all links</td>
</tr>
<tr>
<td></td>
<td>Personalized Mozilla</td>
<td>Predicts and prefetchs based on history</td>
</tr>
</tbody>
</table>

Test and evaluate prefetching techniques. 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 to work with Delfos.

Fig. 1 depicts the framework architecture. It comprises three main parts: the web client, the web server and the prediction engine. The web client includes a web browser with prefetching support (Mozilla) and a tool to capture and replay web navigation sessions (CARENA, [23]). The web server (Apache 2) includes a module (Mod-prefetch) to query predictions and provide them to the web client. The prediction engine (Eprefes) performs predictions and provides hints to the web server. Below we detail how these parts work.

A. Mozilla Firefox

Mozilla is able to prefetch hints if they are included in the response HTTP headers or embedded on the HTML file [24]. This prefetching mechanism was first proposed by Padmanabhan and Mogul [11], and standardized in HTTP/1.1 RFC 2616. The web server can provide one or more URIs if it considers that the user is likely to visit them soon. These URIs or 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">

When implementing prefetch in Mozilla, some interesting aspects concerning to what and when to prefetch must be considered. 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, so web servers can filter those requests, for example, in case of overload. If the user clicks on a link while the browser is prefetching, the prefetch process is interrupted to satisfy the users’ real request. If there was any prefetching queue, it will be 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.

B. Mod-prefetch for Apache 2

Mod-prefetch is a module for the Apache 2 web server that request hints to the prediction engine and submits them in the HTTP response headers to the web browser. See section III-C.2 for more information.

Fig. 2 shows the communication between Mod-prefetch and the prediction engine. When the web server receives a request from a web browser, 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. If the response includes hints, they are added to the HTTP response headers as described in HTTP/1.1, and sends them to the web client together with the rest of the HTTP message.

C. 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 it 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 milisecond.
To verify Eprefes accuracy, experiments were run both on it and on the simulator proposed by Domènech et al [4], obtaining negligible deviations.

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’s discuss them in more detail.

Eprefes is independent of the web server that queries it and 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 new 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 II gives a general view of the available modules and their purpose. Fig. 3 shows the server architecture, the relation among the different modules, the called functions and parameters, and the results returned by them.

Most modules have configurable parameters, for example, the maximum number of hints that can be provided as response to a prediction request. They can be set not only in the configuration file before start up, but can also be modified at runtime by other modules, i.e., a new module that would allow to modify such parameters using a web interface or shell commands which can be very useful for adaptive policies.

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.

2) Connectivity: 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. On the other hand, the response message is simply a list of hints.

3) Serve requests: mod-serve manages each received request depending on the message type, that can be a prediction, a prefetch or a fetch 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. Besides, these hints are also notified to the statistics module.

mod-trainer is an optional module designed to provide statistics when using web server log files as input for the prediction engine instead of real web clients with prefetching capability. It intercepts the hints provided by the prediction module and generates fictitious prefetch requests. If the legitimate user later requests an object that was virtually prefetched, the module intercepts this request and converts it to a fictitious fetch request.

Currently, mod-trainer prefetchs all hints if the corresponding objects are not on the client cache yet without considering whether the client has idle time enough or not to prefetch them. In a real scenario, browsers may not be able to prefetch all the hints. As a consequence the results obtained when using mod-trainer are optimistic and suppose an upper bound.
**Fig. 3.** *Eprefes* architecture

<table>
<thead>
<tr>
<th>Subject</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity</td>
<td>mod-socket</td>
<td>Listen for TCP connections</td>
</tr>
<tr>
<td>Serve</td>
<td>mod-serve</td>
<td>Manage requests depending on the request type</td>
</tr>
<tr>
<td></td>
<td>mod-trainer</td>
<td>Optional: Read log files to train prediction algorithm</td>
</tr>
<tr>
<td>Statistics</td>
<td>mod-stats</td>
<td>Maintain variables and calculate performance indexes</td>
</tr>
<tr>
<td></td>
<td>mod-report</td>
<td>Generate reports periodically</td>
</tr>
<tr>
<td>Prediction</td>
<td>mod-palmen</td>
<td>Make predictions using the Palpanas and Mendelzon’s algorithm (PPM)</td>
</tr>
<tr>
<td></td>
<td>mod-padmog</td>
<td>Make predictions using the Padmanabhan and Mogul’s algorithm (DG)</td>
</tr>
</tbody>
</table>

**TABLE II**

**MODULES IN Eprefes**

4) **Statistics gathering:** mod-stats 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, prefetches 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.

Additionally, four performance indexes are calculated: precision and recall measured both in number of objects and size of objects [25].

\[
\text{Precision} = \frac{\text{Prefetch hits}}{\text{Prefetchs}}; \ \text{Recall} = \frac{\text{Prefetch hits}}{\text{User requests}}
\]

For all of them, the mean value and the confidence interval is 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.

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

5) **Prediction algorithms:** We have currently implemented the two prediction algorithms more widely referred in the literature; Dependency Graph (DG) proposed by Padmanabhan and Mogul [11] and Prediction by Partial Matching (PPM) proposed by Palpanas and Mendelzon [8], on modules mod-padmog and mod-palmen respectively.

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

mod-padmog implements the prediction algorithm Depen-
show the evolution of the precision and the recall.

mod-palmem implements the prediction algorithm proposed by Palpanas and Mendelzon [8]. It is based on Prediction by Partial Matching (PPM), a particular version of Markov model algorithms.

D. Trainer

Prediction algorithms require a training process long enough before being able to provide precise hints. We developed a program to train the prediction engine using log files previously captured by the web server. In this way, it is not necessary to wait until user’s accesses train the prediction engine. The program can also be used for stressing and benchmarking the prediction engine or for evaluation purposes. Trainer is an optional tool which accepts trace files in Common Log Format or Combined Log Format, both of them are standard formats in web server software. Those trace files are 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.

Trainer reads the trace file sequentially and sends messages to the prediction engine using the same communication method that the previously described Mod-prefetch does. The provided hints are printed on screen and can be written to a file, which permits to compare the results obtained on different experiments.

Trace files are slightly converted before being parsed by the Trainer. 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).

E. CARENA

CARENA [23] 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 important to obtain precise and reliable performance results. CARENA is multiplatform, open source, lightweight, standards based, easily installable and usable, programmed in JavaScript and XUL. We use it to test the correct behaviour of Delfos.

IV. EXPERIMENTAL RESULTS

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, Delfos was configured in the same way through the different experiments. Common options are: maximum of 100 hints allowed on a HTTP response, interval length of 100000 user requests and subintervals length of 5000. Regarding mod-palmem (PPM) specific options: threshold 0.2, maximum order 1, minimum order 1 (see section III-C.5 for references). And mod-padmog (DG) specific options: lookahead window size 1. Previous work [22] demonstrates that those values provide relatively good ratio cost-benefit.

A. Performance indexes

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 on 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 III-C.3) is enabled to generate prefetchs based on the predictions and hits based on the real user requests logged. Those 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 on a standard PC (Intel Pentium 4 3.4 GHz, 6800 bogomips, 1 GB of RAM).

The length of the experiments is measured in processed user requests. The trace file was logged in combined log format by an Apache 2 serving the web site of School of Computer Science from the Polytechnic University of Valencia. The trace file used on those experiments includes five million user requests, starts on October, 1st 2005 and ends on March, 23th 2006, it contains on average 26000 user requests per day, 15000 different objects requested, 50 gigabytes transferred in total, 285 megabytes transfered per day in average, and no previous training phase was used.

Fig. 4 shows the evolution of the precision and the recall, both of them measured per object and per byte. Decreasing the prediction algorithm’s threshold increases the prediction (and hence the prefetching) aggressiveness, which also increases the cost in bandwidth usage, and also the benefit in latency.
reduction. 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 to see not only average values but also confidence intervals helps to detect transitional phases.

Fig. 5 shows the evolution of recall and recall per byte along the time. In addition to the standard accumulated indexes shown before (labeled \textit{EXP}), this figure includes the \textit{INT} indexes that only consider the values of the last interval to calculate the indexes. Each interval includes 100,000 user requests. As expected, the indexes that do not consider old values (\textit{INT}) are more variable than the indexes that consider all the values from the beginning of the experiment (\textit{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 ran using the DG prediction algorithm. Fig. 6 shows precision and recall indexes obtained in this experiment. Since the same environment with similar characteristics was used to ran this and the previous experiment, the results can be compared side by side to detect differences on 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 reduction and bandwidth consumption. Fig. 7 depicts the latency reduction per object as a mean value and the confidence interval. On the other hand, Fig. 8 depicts the object traffic increase. Comparing the results of those figures, the PPM prediction algorithm configured on this experiment required 20% of object traffic increase to provide 10% of latency reduction per object.

B. System statistics

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

Some statistics are equally defined for all prediction algorithms and hence can be used to compare how the algo-
rithms operate. For example, Fig. 9(a) shows total memory consumption of data structures on the experiment using PPM and Fig. 9(b) 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 requirements or computation requirements.

Fig. 11 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 increment on 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. 10. Database operations (Fig. 10(a)) provide an approximate number of operations performed on the database. An approximation to the CPU consumption is depicted in Fig. 10(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 can not 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. 12(a)), number of nodes of order 0 and 1 (Fig. 12(b)).

Fig. 13 shows examples of statistics on a graph-based data structure, in this case the one used by mod-padmog (DG): total number of arcs and nodes, mean nodes occurrence, mean arcs occurrence, and mean arcs probability, mean arcs probability.

Delfos can be used to discover new insights into prefetching thanks to the detailed statistics. As an example, let’s 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, both when using 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 nor recall, but data structures grew, making the algorithm slower.

C. Experiments in real environment

We ran two experiments to verify that Delfos is ready for real usage. The first one analyzes how the number of objects in the browser cache increases due to the prefetch actions performed and the second one deals with document latency reduction achieved due to 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 are simple HTML files with one or two embedded image files. The prediction engine was previously conveniently trained. The PPM prediction algorithm was used. CARENA 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 and to replay exactly the same navigation session several times.

For the first experiment, a navigation session consisting on 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 afterwards required by the user, which translates to a precision of 85.7%.

In summary, this small and limited experiment illustrates how prefetching is able to reduce the latency perceived by the user in a transparent way.

Results of the second experiment are presented on Fig. 14. In this case, the navigation session consists of five document requests. It was repeated twice, the first one without prefetching and the second one with prefetching enabled. The client and 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 reduction. A prefetch from the fourth document was a hit on the fifth document, which explains the 50 ms latency reduction.

V. CONCLUSIONS

In this paper we presented Delfos, a framework that provides web prefetching capabilities in real environments. To the knowledge of the authors, it is the first implementation for real usage that features smart prediction algorithms and provides hints using the method described on HTTP 1.1. Delfos is also a flexible tool that can be used either for research purposes or performance evaluation analysis.

Concerning to its usage in real environments, 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.

In order to make it useful and suitable for research and performance evaluation, it 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.

ACKNOWLEDGMENTS

The authors would like to thank the technical staff of the School of Computer Science from the Polytechnic University of Valencia (www.ei.upv.es) for providing us recent and customized trace files logged by the web server from that school web site.

REFERENCES

Fig. 10. System statistics

(a) Database operations per interval (INT)

(b) Reductions per interval (INT)

Fig. 11. Prediction service time (in milliseconds)

(a) Mean number of children

(b) Number of nodes of order 0, 1

Fig. 12. Statistics provided by mod-palmen (PPM)
Fig. 13. Statistics provided by mod-padmog (DG)

Fig. 14. Latency of documents in a navigation session