# A Toolkit for Caching and Prefetching in the Context of Web Application Platforms

Diplomarbeit

Berichter / Supervisors:
Prof. Erhard Konrad
Dipl.-Inf. Atilla Akarsular

Fachgebiet Wissensbasierte Systeme
Technische Universität Berlin

Vorgelegt von / By

Hans Christian Siefkes

Matr.-Nr. 17 68 97

christian@siefkes.net
http://www.siefkes.net/

14 March 2002

## Abstract

In this thesis, I design and implement a toolkit for caching documents retrieved from URLs as well as any other content. Both file and memory caches are used; the toolkit allows the integration of generic parsers and can store documents in parsed state. The caching procedure follows the expiration and validation rules of HTTP 1.1.

The architecture of the toolkit is pluggable, so different replacement strategies can be integrated. The LRU (Least Recently Used) strategy is provided as a reference implementation.

The toolkit is integrated with the MUNDWERK voice gateway, a Web application platform for voice applications based on the VoiceXML markup language. The attributes and properties defined by the VoiceXML 1.0/2.0 standard for controlling caching and prefetching are fully supported. The toolkit can be integrated with other platforms without changes to the basic architecture.

The toolkit also supports prefetching in a transparent way. The implemented prefetching strategy is based on the PPM (Prediction by Partial Match) strategy, which utilizes higher-order Markov models. It is extended to consider the time spans between requests. A blending algorithm is used to combine the results of Markov predictors of different orders.

The pluggable architecture permits complementing or replacing the PPM algorithm by other strategies. Prefetching is handled asynchronously, using primarily system resources that are not needed for other purposes.

In test runs, prefetching did increase the PRR (perceived retrieval rate) by up to 175%, without causing a significant increase in network traffic. While refined prefetching strategies so far have been mainly a research topic, this toolkit brings them to usage in production.

## Zusammenfassung

Eine Zusammenfassung in deutscher Sprache finden Sie am Ende dieser Arbeit (Appendix B).

# Contents


**List of Figures** 6

**List of Tables** 8

**1 Introduction** 11

**2 Context** 13

2.1 Terms and Definitions . . . . . 13

2.2 Setting . . . . . 15

2.3 Objectives and Limitations of Caching and Prefetching . . . . . 18

**I Caching** 19

**3 Caching—An Overview** 21

3.1 Expiration and Validation in HTTP 1.1 . . . . . 21

3.2 VoiceXML Support for Caching and Prefetching . . . . . 23

3.2.1 VoiceXML 2.0 . . . . . 24

3.2.2 VoiceXML 1.0 . . . . . 26

**4 Caching Strategies** 27

4.1 Standard Strategies . . . . . 28

4.2 Variations of the LRU and LFU Strategies . . . . . 29

4.3 GreedyDual-Size (GDS) . . . . . 30

4.4 Improved Removal Procedure and Least Specific Cost . . . . . 31

**5 Analysis of the Caching Solution** **33**

**6 Design, Implementation, and Integration of the Caching Solution** **35**

**7 Evaluation of the Caching Solution** **37**

7.1 Measuring the Performance of Caching . . . . . 37

7.2 How and What to Evaluate . . . . . 38

7.3 Evaluation Results . . . . . 39

**II Prefetching** **43**

**8 Prefetching—An Overview** **45**

8.1 Types of Prefetching . . . . . 45

8.2 Prefetching Issues and Limitations . . . . . 47

**9 Prefetching Strategies** **51**

9.1 Semantic Prefetching Strategies . . . . . 52

9.1.1 Simple Strategies . . . . . 52

9.1.2 Keyword-based Strategies . . . . . 53

9.2 Statistic Prefetching Strategies . . . . . 53

9.2.1 Dependency Graphs . . . . . 53

9.2.2 First-Order Markov Models . . . . . 54

9.2.3 Higher-Order Markov Models: Prediction by Partial Match 55

9.2.4 A Data Mining Approach: LxS . . . . . 58

**10 Analysis of the Prefetching Solution** **61**

**11 Design, Implementation, and Integration of the Prefetching Solution** **63**

**12 Evaluation of the Prefetching Solution** **65**

12.1 Measuring the Performance of Prefetching . . . . . 65

12.2 How and What to Evaluate . . . . . 66

12.3 Evaluation Results . . . . . 66

12.3.1 Default Setting . . . . . 66

12.3.2 Configuration Variations . . . . . 68

12.3.3 Load Variations . . . . . 73
12.4 Design Change Caused by the Evaluation . . . . . 73

**13 Conclusion** **75**
13.1 Résumé . . . . . 75
13.2 Extensions and Further Work . . . . . 76
13.3 Lessons Learned . . . . . 77

**Bibliography** **79**

**Appendix** **83**

**A Implementation Issues** **85**

**B Zusammenfassung in deutscher Sprache** **87**

# List of Figures


2.1 Typical Setting—Browser Cache . . . . . 15
2.2 Typical Setting—Proxy Cache . . . . . 16
2.3 Setting—Web Application Platform . . . . . 16

3.1 Principles of Validation and Expiration . . . . . 22

7.1 Default Setting: Cache and Byte Hit Rates . . . . . 40
7.2 Default Setting: PRR . . . . . 40
7.3 Various Cache Sizes: Cache Hit Rate . . . . . 41
7.4 Various Minimum Expiration Times: Cache Hit Rate . . . . . 41

8.1 Types of Client- and Server-Based Prefetching . . . . . 46
8.2 Prefetching Based on Proxy resp. Web Application Platform . . . . 47

9.1 Sample Prediction Model of the PPM Strategy . . . . . 57

12.1 Prefetching Default: Coverage . . . . . 67
12.2 Prefetching Default: Accuracy and Usefulness . . . . . 67
12.3 PRR With and Without Prefetching . . . . . 68
12.4 Network Traffic . . . . . 69
12.5 Various Settings: PRR . . . . . 69
12.6 Various History Lengths . . . . . 70
12.7 Variations in the Number of Predictions . . . . . 71
12.8 Blending Variations . . . . . 71
12.9 Minimum Access Variations . . . . . 72
12.10 Variations of the Average Time Factor . . . . . 72

12.11 Load Variations . . . . . 73
12.12 Load Variations: PRR Increase . . . . . 74

# List of Tables

3.1 Cache Control in VoiceXML 2.0 and 1.0 . . . . . 25

4.1 Input Values of Replacement Strategies . . . . . 28
4.2 Sample Session of a Small LRU Cache . . . . . 29

9.1 Input Values of Prefetching Strategies . . . . . 52

*To my parents, Harm and Uta Siefkes.*

# 1 Introduction

[1] ***cache*** \'kash\ n ***1:*** *a place for hiding, storing, or safeguarding treasure or food and supplies*

— *Webster's New Encyclopedic Dictionary*

*Cache:*
*A very expensive part of the memory system of a computer that no one is supposed to know is there.*

— *From the* fortune *database*

Speed and bandwidth are still a bottleneck of the Web. A user's access to Web documents already visited some time ago can be sped up by using a cache, while prefetching can decrease the time needed to load documents likely to be visited by the user in the future.

While caching techniques are well-known and widely in use, refined heuristic or stochastic prefetching continues to be a topic limited to research projects. Most currently deployed systems employ only very simple prefetching strategies (e.g., prefetching all linked documents), if at all.

In this thesis I will bridge this gap by designing, implementing and documenting a toolkit for caching and prefetching which is used as part of an industry-strength platform for Web applications. Thus my aim is to realize and test a prefetching solution actually employed in production, not only in test scenarios.

I will start by explaining the reasons for using caching and prefetching. I will define the concepts involved and describe the setting of the application platform covered by my implementation.

The rest of this work is split into two parts, the first one dealing with caching, the second one with prefetching. Both parts follow the same structure.

I will review the state of the art in caching and prefetching, describing and evaluating techniques and approaches currently in use in the fields of Artificial Intelligence and Information Retrieval. Following evaluation, I will choose those techniques for implementation that are well suited for the situation and consider whether and how they can be further refined.

Based on the selected approaches I will design and implement a caching and prefetching toolkit, integrating it in the application platform. I will document the implementation and describe the results of test runs and actual usage in production.

The following programs were utilized to create this work: *IBM Visual Age for Java*[1] for software development, *GNU Emacs*[2] and LaTeX[3] for writing and typesetting the text and the tables, *ArgoUML*[4] for generating the UML diagrams, and *Sun StarOffice*[5] for creating the other diagrams and graphics.

I want to thank my supervisors at the Technical University of Berlin, Professor Erhard Konrad and Atilla Akarsular, as well as Mario Ellebrecht and the whole MUNDWERK team.

[1] http://www-4.ibm.com/software/ad/vajava/
[2] http://www.gnu.org/software/emacs/
[3] http://www.ctan.org/
[4] http://argouml.tigris.org/
[5] http://www.sun.com/staroffice/

# 2 Context

*I've touch'd the highest point of all my greatness;*
*And from that full meridian of my glory*
*I haste now to my setting. I shall fall,*
*Like a bright exhalation in the evening*
*And no man see me more.*

— *Shakespeare*

## 2.1 Terms and Definitions

Wessels [2001, 1] explains *caching* thus: "The term *cache* has French roots and means, literally, *to store*. As a data processing term, caching refers to the storage of recently retrieved computer information for future reference. The stored information may or may not be used again, so caches are beneficial only when the cost of storing the information is less than the cost of retrieving or computing the information again."

A request for an object found in the cache is called a *cache hit*, otherwise it is a *cache miss*. The *cache hit rate* (also called *document hit rate* or *hit ratio*) is the percentage of all requests that are cache hits, i.e., the probability that documents are found in the cache when needed. The *byte hit rate* considers the size of documents, counting bytes instead of requests. It measures how much bandwidth is saved by a cache.

In a distributed environment, as the Web is, interactions take place between *clients* and *servers*. A client starts an interaction by sending a *request* to a server. The server acts on the request and sends back a *response*. In the context of the

Web, the most common clients are *Web browsers* (e.g., Netscape or Internet Explorer). *Proxies* act as intermediaries between clients and servers, taking the role of a client when interacting with a server and the role of a server when interacting with a client.

A server's response often contains a *document* that has been requested by a client. Typical documents in the usual Web interactions are HTML and other text files, graphics files, and client-side code (Java, Java Script), while the most common documents in the context of voice applications are VoiceXML documents, grammar files, and audio prompts (cf. sec. 3.2). Documents are sometimes referred to as *objects*, *items*, or *pages*—these terms are used synonymously in this thesis.

Documents can be either *dynamic* or *static*. Dynamic documents are generated on the fly when the server receives a request, while static documents are produced independently of requests.

The protocol commonly used for interaction between Web servers and clients is called *HTTP (Hypertext Transfer Protocol)*. The current version of HTTP is 1.1; it offers more advanced cache control mechanism than previous versions.

Documents form the optional *body* part of HTTP *messages* (requests and responses), while the *header* contains information characterizing the message or the document (e.g., its length or date of last access). An important part of the header of a response is the *status code* which informs about the success or failure of the request—the most common status code is *200 (OK)*, another frequently encountered one is *404 (Not Found)* [cf. Wessels 2001, 1–7, 37].

A cache has only a limited amount of storage. When no storage space is left, it must remove some documents from the cache to make room for new ones. For this purpose, *replacement strategies* (a.k.a. *caching strategies*) are employed. They assign a *value* to each item, removing the least valuable items when required. It depends on the caching strategy how this value is computed.

Sometimes caches are organized in *cache hierarchies*, i.e., two or more caches are combined, each of which is faster—but also smaller—than the previous one. The most valuable objects (as judged by the caching strategy) are kept in the fastest cache. When objects are removed from a cache to make room for new ones, they are moved one step down in the hierarchy, to the next slower cache.

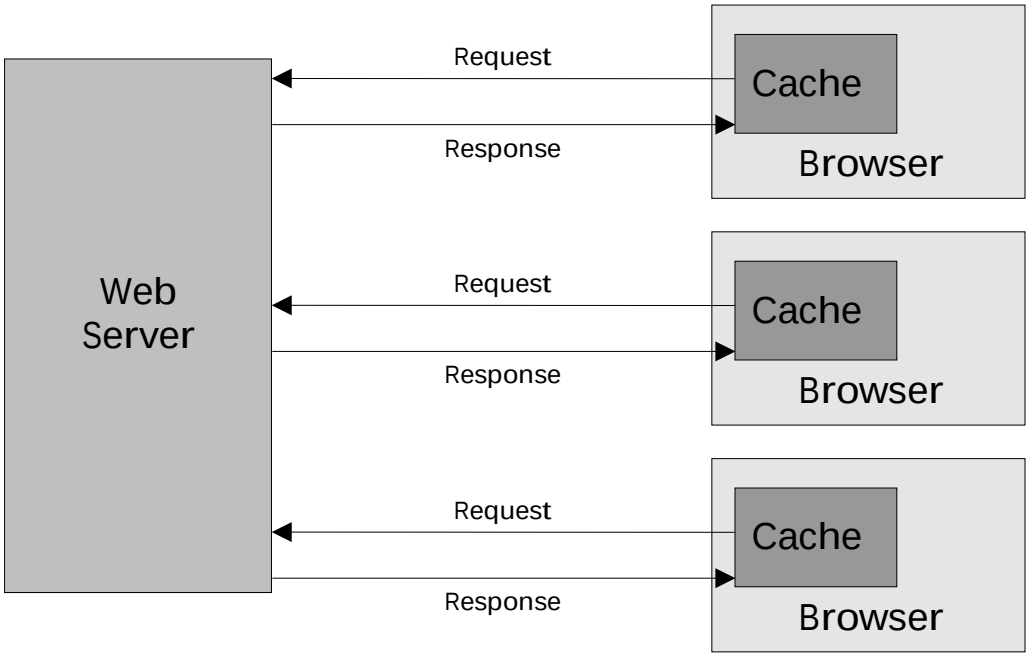


Figure 2.1: Typical Setting—Browser Cache

A *session* spans the sequential interactions between a client and a server. In the voice context, a session usually comprises a telephone call.

*Prefetching* means to anticipate probable future requests and to fetch the most probable documents before they are actually requested. Speculative requests initialized by the prefetcher are called *prefetch requests*, other requests are called *actual requests*. Which documents are prefetched in a certain situation depends on the *prediction model* used for prefetching.

## 2.2 Setting

There are two places where caches are typically found in the World Wide Web: either client-side, as part of the user's Web browser (fig. 2.1); or between clients and servers, as part of a proxy (fig. 2.2).

This caching toolkit is employed in another setting, which combines traits of both of these settings: a *Web application platform*. Web application platforms provide an infrastructure for developing, deploying, and adapting Web appli-

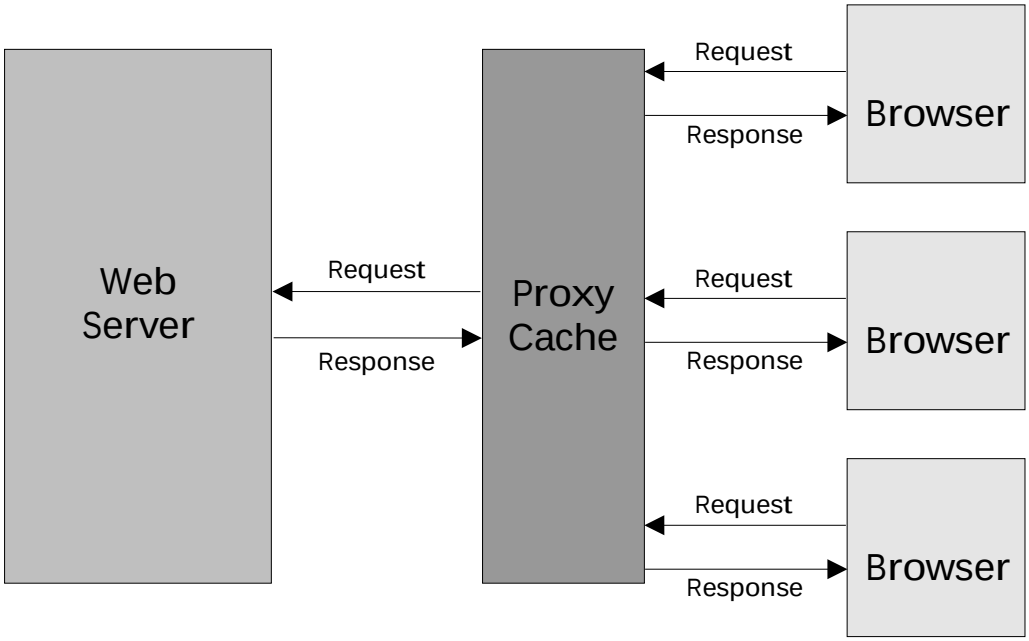


Figure 2.2: Typical Setting—Proxy Cache

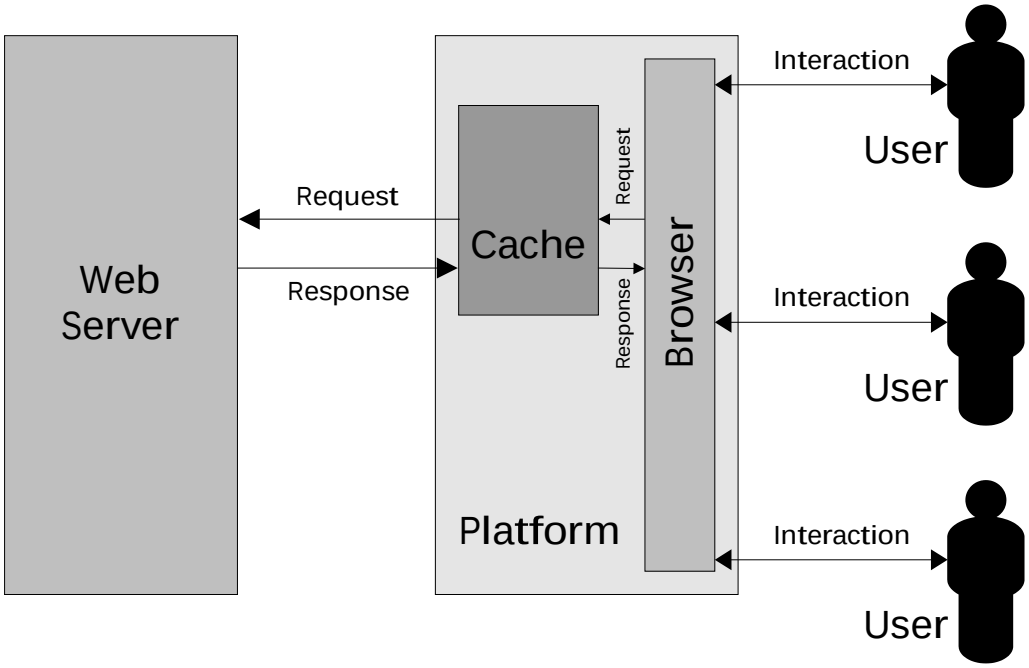


Figure 2.3: Setting—Web Application Platform

cations. Typical Web application platforms are *Macromedia ColdFusion Server,*[1] *IBM WebSphere Application Server,*[2] and *Zope.*[3] Some are especially aimed at electronic commerce, e.g., *Intershop*[4] or *SAP Web Application Server.*[5]

Other Web application platforms act as gateways between the Web and users not directly connected to the Web. They provide services to a number of people using another medium for communication (e.g., a telephone). The data used by a gateway is loaded via the Web. It thus combines properties of a proxy (serving numerous persons) with properties of a client (handling the data itself instead of passing it on). This setting is shown in figure 2.3.

The Web application platform treated in this thesis is a voice gateway. It allows voice-activated services through the telephone. Speech interfaces and language engineering are an emerging trend in user interaction.[6] Voice gateways incorporate these technologies to establish voice-controlled Internet-like services over the phone.

The Web serves as a back-end for delivering data to the voice gateway. The *VoiceXML* markup language is used for dialog design (cf. sec. 3.2).

A testbed and showcase for this gateway is *Stadtlotse*, a voice portal for the Berlin area (in German language). It can be reached by calling 030 / 52 00 51 (in Germany). Other applications are the *Jamba* mobile portal, which allows downloading cell phone ring tones (0190 / 7-JAMBA, i.e., 0190 / 75 26 22), and *clever-tanken.de*, a service for finding the cheapest gas station in a specified area (0190 / 15 25 35).

In this work, the toolkit is integrated and employed in the voice gateway, but it is not limited to this context. Its basic design is independent of this special setting; the adaption to voice context and the data handled is provided as part of the integration.

[1] http://www.macromedia.com/software/coldfusion/
[2] http://www-4.ibm.com/software/webservers/appserv/
[3] http://www.zope.org/
[4] http://www.intershop.com/index.htm?callname=www/products/intershop_4
[5] http://www.sap.com/solutions/technology/web_app_server.htm
[6] More than 600 companies are now members in the *VoiceXML Forum*, an industry organization for promoting speech-enabled applications and the *VoiceXML* markup language. The Forum was founded in 1999 [cf. VoiceXML Forum 2002; XML Cover Pages 2002].

Sometimes the setting of the voice gateway was considered for design decisions, e.g., for determining which caching strategy to implement. Whenever this is the case, it will be explicitly mentioned. Usually this does not affect the core design of the framework, but only parts that are meant to be interchangeable (e.g., the chosen caching strategy can be substituted by any other one). So the main architecture of the toolkit is independent of this special setting.

## 2.3 Objectives and Limitations of Caching and Prefetching

Caching offers three major benefits: documents are retrieved faster (latency reduction), less bandwidth is consumed, and the load placed on Web servers is reduced [cf. Wessels 2001, 11].

However, caching cannot offer a perfect way of speeding up an application. Increasing the cache size beyond a certain point does not give much additional advantage. If the cache is big enough to store documents for a very long time, they often will be out-of-date when the next request comes, so caching them does not make much sense. The bottlenecks that remain when a cache is used are the fetching of documents that have never been requested, the re-fetching of documents that are out-of-date, and the revalidation of documents that might be out-of-date.

Prefetching can resolve some of these issues. By predicting which documents will soon be requested, it can revalidate and if necessary update these documents if they are already cached, or otherwise add them to the cache.

Prefetching can thus reach further latency reduction; but this comes at a cost, as it will typically increase both bandwidth and server load because predictions cannot be 100% accurate.

# Part I

# Caching

# 3 Caching—An Overview

## 3.1 Expiration and Validation in HTTP 1.1

The decision on whether an object can and should be cached depends on HTTP header data as well as some heuristic considerations. A server can control whether a document can be cached by including a *Cache-Control* line in the response header. A number of different Cache-Control directives exist, e.g., *Cache-Control: no-store* completely prohibits caching, while *Cache-Control: private* means that this version must not be given to anybody except the person that originally requested it. Several directives can be combined in a comma-separated list.

Heuristics are often used to identify dynamic content. File names ending in *.cgi, .asp, or .jsp* or path names including *cgi-bin* or *servlet* typically characterize dynamically generated documents. They often will not be reused and thus might not be cache-worthy.

When a document is found in the cache the cached copy can be *validated* to make sure it is not *stale* (out-of-date). This is done by adding an *If-Modified-Since* line with the time stamp of the cached document in the request.

When the document has not changed, the server's response contains the status code *304 (Not Modified)* and no body part, otherwise the status code is *200 (OK)* and a *fresh* (up-to-date) version of the document is send as the message body.

When a server includes an *Expires* line in a response header it guarantees that the document will not change before the specified date. Thus no validation is

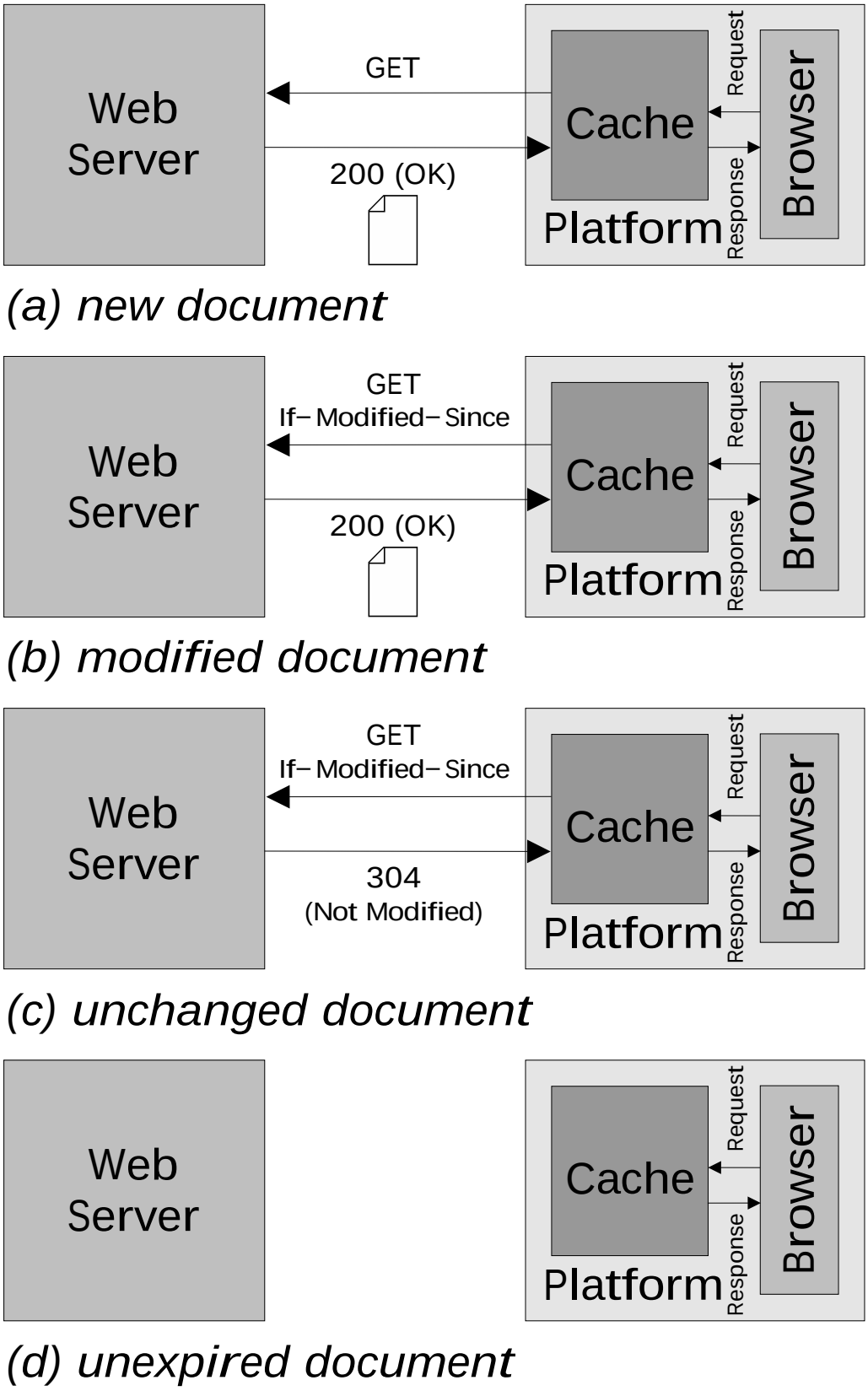


Figure 3.1: Principles of Validation and Expiration

necessary during this time span. Figure 3.1 shows the possible situations when fetching a document.

When no expiration date is given the cache is allowed to estimate a time span the document probably will not change. Doing this saves bandwidth and response times by avoiding overly frequent validation, but at the risk of returning stale data [cf. Wessels 2001, 27–30, 33–35, 38f].

One method to calculate the risk that an object becomes stale depends on the age of the object—typically recently modified objects are more likely to change again than older objects. Thus a *staleness* factor is calculated for each object:

$$staleness = \frac{now - last\ update}{last\ update - last\ modification}$$

Objects whose staleness factor is less than a certain threshold (e.g., 10%) are served from the cache without validation. Caches implementing such an approach are called *weakly consistent* [cf. Wessels 1995, sec. 2.1].

## 3.2 VoiceXML Support for Caching and Prefetching

*XML (Extensible Markup Language)* is a meta-format for defining structured document formats (often called *markup languages* because they define the markup of documents). *VoiceXML* is one of the markup languages based on XML. It covers speech-based human-computer interactions.

In XML, *tags* and *attributes* are used for formatting documents. Tags (or *elements*) are used to "tag" data, i.e., to define the role of the data. They are written between angle brackets. Attributes can provide further information about a tag.

The following example contains the `emphasis` tag and its `level` attribute defined in VoiceXML 2.0:

```
<emphasis level="strong">Beware of the dog.</emphasis>
```

In VoiceXML, *properties* can be used to specify default values for certain attributes (the `property` tag is used for this).

### 3.2.1 VoiceXML 2.0

The drafted 2.0 version of the VoiceXML standard gives authors refined control over caching and prefetching functionality. The caching behavior depends on two attributes, `maxage` and `maxstale`.

`maxage` specifies the maximum age of an un-expired resource. If the age of a resource is greater than the specified `maxage` value, it must be re-fetched or revalidated, even if it is not expired. The `maxstale` attribute allows an expired resource to be used without revalidation, if it did expire during the specified time span.

Prefetching can be controlled by the `fetchhint` attribute: setting it to `safe` disables prefetching, while `prefetch` allows it—this is the default. These three attributes can be applied to all tags that require fetching a resource from an URL (`audio`, `choice`, `goto`, `grammar`, `link`, `object`, `script`, `subdialog`, `submit`).

Default values for a whole application or parts of it can be specified by setting a property whose name combines the name of the attribute and the kind of document for which they apply. Five document types can be specified as prefix:

- `audio` for audio prompts,
- `document` for VoiceXML documents,
- `grammar` for external grammars,[1]
- `object` for contents of the `object` tag, and
- `script` for scripts.

For example, the `audiofetchhint` property sets a default value for the prefetching of audio prompts; the `grammarmaxage` and `grammarmaxstale` properties influence the caching of grammar files.

This new version of the standard explicitly requests that the cache correctness rules of HTTP 1.1 are followed, e.g., that caching-related HTTP headers are evaluated. The `meta` element allows the specification of header data inside a VoiceXML document, by specifying the name of an HTTP response header (e.g., *Expires*) in the `http-equiv` attribute and the value of this header in the `content` attribute [cf. W3C 2001b, sec. 6.1, 6.2, 6.3.5].

[1] In VoiceXML, grammars are either *inline*, i.e., included in the VoiceXML document, or *external*, i.e., contained in an external file.

Table 3.1: Cache Control in VoiceXML 2.0 and 1.0

VoiceXML 2.0

| **Tag** | **choice, goto, link, subdialog, submit** | **audio** | **grammar** | **object** | **script** |
|---|---|---|---|---|---|
| **Attribute** | fetchhint | | | | |
| **Property** | documentfetchhint | audiofetchhint | grammarfetchhint | objectfetchhint | scriptfetchhint |
| **Values** | prefetch *(prefetching allowed)* / safe *(prefetching disabled)* | | | | |
| **Attribute** | maxage | | | | |
| **Property** | documentmaxage | audiomaxage | grammarmaxage | objectmaxage | scriptmaxage |
| **Values** | *x* / *x*ms / *x*s *(validate resource older than x milliseconds/seconds)* | | | | |
| **Attribute** | maxstale | | | | |
| **Property** | documentmaxstale | audiomaxstale | grammarmaxstale | objectmaxstale | scriptmaxstale |
| **Values** | *x* / *x*ms / *x*s *(do not validate resource that expired at most x milliseconds/seconds ago)* | | | | |

VoiceXML 1.0

| **Tag** | **choice, goto, link, subdialog, submit** | **audio** | **grammar, dtmf** | **object** | **script** |
|---|---|---|---|---|---|
| **Attribute** | fetchhint | | | | |
| **Property** | documentfetchhint | audiofetchhint | grammarfetchhint | objectfetchhint | scriptfetchhint |
| **Values** | prefetch *(prefetching allowed)* / safe *(disabled)* / stream *(audio only: start playing while loading)* | | | | |
| **Attribute** | caching | | | | |
| **Property** | caching | | | | |
| **Values** | fast *(caching allowed)* / safe *(caching disabled)* | | | | |

### 3.2.2 VoiceXML 1.0

The caching support of the previous version of the standard, VoiceXML 1.0, is less precise. Instead of the `maxage` and `maxstale` properties only a single attribute, `caching`, exists. Setting it to `safe` prohibits caching, while the default value, `fast`, allows it. The single `caching` property can be used to supply a default value for all types of documents; special defaults for audio files, VoiceXML documents, or grammars cannot be defined.

The `fetchhint` attribute, on the other hand, is a bit more refined: For audio prompts a third option, `stream`, exists. It means that the voice browser should not wait until the resource is completely fetched but start playing it while it is still loading [cf. Edgar 2001, 148, 261f, 442f].

Table 3.1 contains a synopsis of the cache-control mechanisms in both versions of the VoiceXML standard.

# 4 Caching Strategies

*Strategy:*
*A long-range plan whose merit cannot be evaluated until sometime after those creating it have left the organization.*
— *From the* fortune *database*

Caching strategies decide which documents to remove from the cache when space is needed to cache new ones. Their goal is to make the usage of the cache as efficient as possible, by trying to optimize one or several of the performance criteria of caching (cf. sec. 7.1): cache hit rate, byte hit rate, or response time. Additionally, the strategies themselves should work efficiently, i.e., their CPU and memory usage should be as low as possible.

To fully optimize the cache usage, all future requests must be known, which is obviously impossible. Therefore the strategies use heuristics to decide which documents to replace and which to keep. The input values considered for this purpose include:

- The time of last access to the document;
- Previous times of access;
- The time the document was inserted into the cache;
- Number of accesses;
- Document size;
- Cost of retrieval, e.g., the time passed while fetching the document;
- Time to live, i.e., the time left until the document expires.

Table 4.1 shows which input values are utilized by the caching strategies presented in this chapter.

| | Time of last access | Time of previous accesses | Time of being added | Number of accesses | Size | Cost of retrieval | Time to live |
|---|---|---|---|---|---|---|---|
| **LRU** | x | | | | | | |
| **LRU-*k*** | | x | | | | | |
| **LRU-Threshold** | x | | | | x | | |
| **LFU** | | | | x | | | |
| **LFUDA** | x | | | x | | | |
| **FIFO** | | | x | | | | |
| **RAND** | | | | | | | |
| **SIZE** | | | | | x | | |
| **Log(Size)+LRU** | x | | | | x | | |
| **LRV** | x | | | x | x | | |
| **LRV-TTL** | x | | | | x | x | x |
| **GDS** | x | | | | x | x | |

Table 4.1: Input Values of Replacement Strategies

## 4.1 Standard Strategies

*Least Recently Used (LRU)* is the most common caching strategy. The items not used for the longest time are removed from the cache to make way for new ones. This algorithm leads to a high hit rate while causing little overhead.

Table 4.2 shows a sample session of a small LRU cache. The cache can keep five documents; each document is represented by a letter.

A similar strategy is called *Least Frequently Used (LFU)*. Items are not weighted by the time of last access but by the number of accesses. This algorithm makes it hard for new items to become as valuable as older ones, thus the cache tends to be jammed with old objects that have once been popular. For

| **Request** | A | B | C | B | D | E | F | C | A | E | G |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Ordered** | A | B | C | B | D | E | F | C | A | E | G |
| **contents** | | A | B | C | B | D | E | F | C | A | E |
| **of the** | | | A | A | C | B | D | E | F | C | A |
| **LRU** | | | | | A | C | B | D | E | F | C |
| **cache** | | | | | | A | C | B | D | D | F |

Table 4.2: Sample Session of a Small LRU Cache

this reason it is seldom used in practice.

Even less useful for caching is the *First In, First Out (FIFO)* strategy. Here the value of the items is not measured at all, they are simply replaced in the same order they were added. Similarly unsuitable is *RAND*, which randomly selects items for replacement. These strategies can be utilized as a lower bound for the effectiveness of replacement strategies [cf. Wessels 2001, 45].

A very simple strategy is called *SIZE*. It replaces the largest item. In Web applications, its performance is usually better than LRU when it comes to document hits, but it is very poor when the byte hit rate is measured.

A more complex replacement strategy is called *Lowest Relative Value (LRV)*. It assigns each cached document a value depending on size, number of past requests and the time of the last request. It is thus a combination of LRU, LFU, and SIZE, aimed at increasing document as well as byte hit rate. This strategy receives a very good cache hit rate, but has a high computation demand [cf. Palpanas 1998, 11].

Another variation, which might be called *LRV-TTL*, calculates the value based on the time of last request, retrieval time, time to live (TTL—the given expiration date), and size [cf. Reddy and Fletcher 1998, sec. 2.3].

## 4.2 Variations of the LRU and LFU Strategies

*LRU-k* is an extension of the LRU algorithm. While LRU measures the value based on the very last access, LRU-*k* uses the *k*-th last access to do this. Thus

in LRU-2 the value is based on the second most recent access. LRU-1 is equivalent to LRU [cf. Tomkins 1997, 36f].

*LRU-Threshold* only caches documents up to a certain size, which increases the cache hit rate while decreasing the byte hit rate.

*Log(Size)+LRU* combines the SIZE and LRU strategies by evicting the least recently used of all documents whose $\log(size)$ is highest [cf. Cao and Irani 1997, sec. 2.2].

*LFU with Dynamic Aging (LFUDA)* combines LFU with age considerations to avoid jamming the cache with once popular items. It thus effectively combines LFU and LRU. Recently requested documents get a bonus compared to other ones. This bonus is called the *cache age factor*. It is added to to reference count of an object inserted in or retrieved from the cache. The cache age factor is incremented each time an object is removed from the cache [Dilley et al. 1999, sec. 2].

## 4.3 GreedyDual-Size (GDS)

*GreedyDual-Size (GDS)* is another algorithm based on cost and size. It was specially designed to take some peculiarities of Web caching into account: Web objects differ enormously in size and even objects with similar sizes can have very different loading times, depending on the server they reside on.

To take this into account, GDS associates a value $H$ with each page $p$ in the cache. When a page is first stored in the cache, its $H$ value is set to $cost/size$, where *cost* is the cost of fetching it and *size* is the size of the document. The cost is set to a metric that should be minimized, e.g., to the downloading latency to reduce average latency.

When the cache is full, the page with the lowest $H$ value, $min_H$, is removed, while the $H$ value of all cached pages is reduced by $min_H$. When a cached page is requested, its $H$ value is reset to $cost/size$. The algorithm thus incorporates the time of last access as well as as the cost of retrieving a document and its size.

Actually reducing each $H$ value each time a replacement is made would be very inefficient, but can be avoided by keeping an inflation value $L$ and increasing all future settings of $H$ by $L$ [cf. Cao and Irani 1997, sec. 1, 4].

## 4.4 Improved Removal Procedure and Least Specific Cost

Almost all caching strategies have one thing in common: each cached object is assigned a value, and the least valuable objects are replaced. The strategies differ in how they determine the value: in the case of LRU, the value is the time of the last request; for LFU it is the number of requests to an object, etc.

Hosseini-Khayat [1998, 91–93] proposes an refinement usable for each of these algorithms, called *Improved Least-value Removal Procedure (ILVR)*. Usually, when there is not enough space left in the cache to add a new item, these algorithms repeatedly remove the least valuable object until the new item fits in. Removing the last of these items may release that much space that the removal of some of the other, previously removed items, might have been unnecessary.

The ILVR procedure tackles this problem by assembling a list $L'$ of all items that would otherwise be removed. $L'$ is sorted by value. The algorithm marks items in $L'$ for removal by iterating through the list starting at the least valuable item and adding the size of all unmarked items until the sum is equal to or greater than *deficit*, the amount of space that needs to be freed. Then the last visited item is marked for removal and *deficit* is decreased by the size of this item.

This procedure is repeated until *deficit* is reduced to zero or less. All marked items are deleted. Thus some of the items that would otherwise have been deleted are spared, if possible.

Another improvement proposed by Hosseini-Khayat [1998, 91–93] is to take the *specific cost* into account. The *cost* of an item is the performance penalty occurring when a requested object is not found in the cache. The *specific cost* is the cost divided by the size of the item.

Any value-based algorithm can be adapted as follows: the least valuable items are assembled in a list $L'$ until the size of the list reaches some threshold value $\theta B$. The items to delete are found by an ILVR procedure on $L'$, but the list is sorted by specific cost instead of value. Thus the items deleted are those with the least specific cost out of the items with little value.

# 5 Analysis of the Caching Solution

*Not included in this version.*

# 6 Design, Implementation, and Integration of the Caching Solution

*Not included in this version.*

# 7 Evaluation of the Caching Solution

*According to the latest official figures,*
*43% of all statistics are totally worthless.*
— *From the* fortune *database*

## 7.1 Measuring the Performance of Caching

Performance measuring can serve several goals:

1. Figuring out whether and how useful a feature is (e.g., caching or prefetching), by measuring the performance with and without this feature, thus exploring the benefits and costs of the feature.
2. Finding out which configuration is better suited for a setting, by comparing different configurations.
3. Finding out which strategy (cache replacement resp. prefetching strategy) performs better in a setting. This is similar to testing different configurations, but requires additional work, because strategies have to be implemented before they can be compared.

The evaluation done in this chapter for caching and in chapter 12 for prefetching covers the first two goals. The metrics used here could serve for comparing different strategies as well, but implementing further strategies is beyond the scope of this work.

These are the most important metrics to measure the performance of a cache [cf. Wessels 2001, 192–194, 209f]:

**Cache hit rate:** The percentage of requests that are served from the cache.

**Byte hit rate:** Measures the percentage of transferred bytes instead of requests.

**Response time:** The time taken by the cache to respond to a request. The response times for cache hits and for cache misses should be very different, so it is important to measure them both.

**Perceived retrieval rate (PRR):** This metric combines byte hit rate and response time into a single value, by dividing the number of bytes delivered through the time taken for delivery [cf. Reddy and Fletcher 1998, sec. 3].

Especially the response time and the PRR will depend on the overall load of the cache. The behavior under extreme load can be tested by increasing the *throughput*, i.e., the number of requests per second the cache has to handle.

## 7.2 How and What to Evaluate

Two approaches for benchmarking cache performance exist: either the benchmarking software simulates Web traffic by generating requests for different documents or it re-uses real Web traffic by replaying trace log files.

*Web Polygraph*[1] is a highly configurable tool for simulating traffic, while *Blast*[2] and *WebJamma*[3] replay log files. All of these solutions are free software (open source) [cf. Wessels 2001, 197–202].

The first approach usually aims at simulating normal Web traffic based on HTML and graphics files. The traffic caused by a voice gateway consists mainly of VoiceXML, audio, and grammar files; its characteristics supposedly are quite different from HTML-based traffic. Adapting the simulators to voice context without losing realism would be a large task, requiring careful studies to get the characteristics right.

[1] http://www.web-polygraph.org/
[2] http://www.cache.dfn.de/DFN-Cache/Development/blast.html
[3] http://www.cs.vt.edu/~nrg/webjamma.html

Therefore the trace-based approach is chosen in this work. The considered tools are not flexible enough for testing both the caching and prefetching functionality: While Blast and WebJamma can replay requests from a trace file, they neither consider the time spans passed between requests nor the concept of sessions. This makes them insufficient for testing the performance of prefetching in a realistic way.

Therefore it was necessary to create a new test driver that supports the desired functionality. A configurable number of sessions take place in parallel; different configurations for the caching and prefetching toolkit can be specified for comparing the results.

The test driver uses the `CacheMetrics` for measuring and employs the performance criteria listed above. Average response time and perceived retrieval rate are calculated for four different states of fetch requests:

- Caches misses,
- Cache hits that were replaced by a new version after validating stale,
- Hits validated to be fresh,
- Un-expired hits.

## 7.3 Evaluation Results

For these tests, 150 user sessions containing 18750 traces were replayed. The traces referred to 1418 URLs whose combined size was 47.9 MB. For the results shown in this section, 20 sessions were replayed in parallel. The size of the used cache was 32 MB, unless specified otherwise.

Figure 7.1 shows the cache and byte hit rates for this setting. Figure 7.2 shows the PRR of the different states of requests.[4] Without caching, the PRR of all requests would be similar to the PRR of the cache misses.

The cache hit rates for caches of various sizes are shown in figure 7.3. Doubling the cache size from 8 to 16 MB clearly increases the number of hits. Increasing the cache size to 32 MB offers still offers small improvements; a further

[4] There are not enough replaced hits to give representative results, therefore they are not shown.

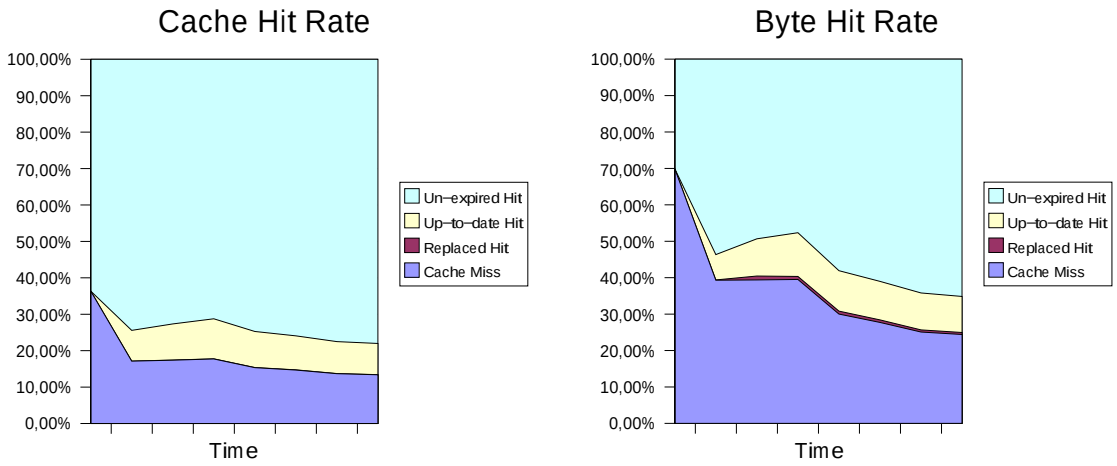


Figure 7.1: Default Setting: Cache and Byte Hit Rates

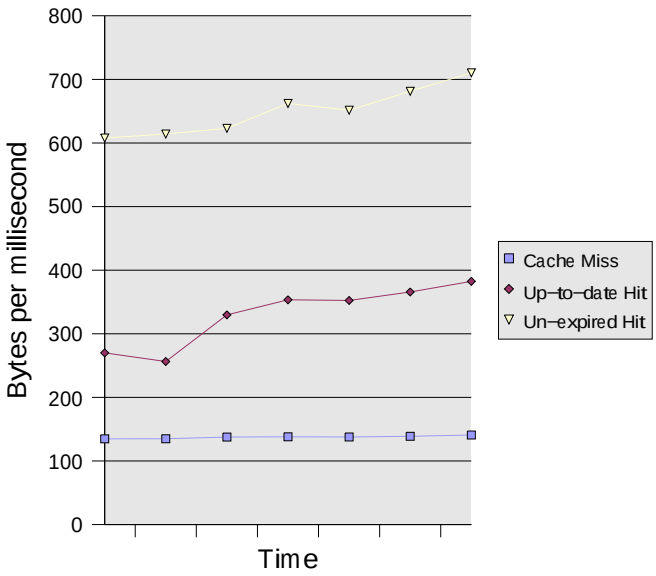


Figure 7.2: Default Setting: PRR

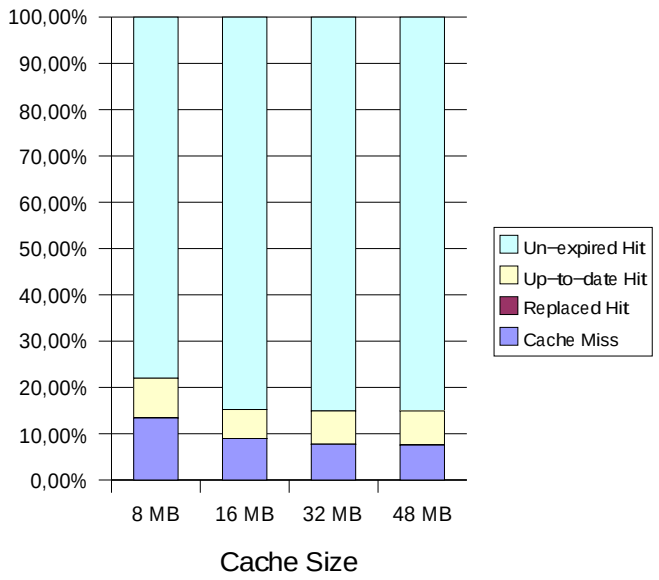


Figure 7.3: Various Cache Sizes: Cache Hit Rate

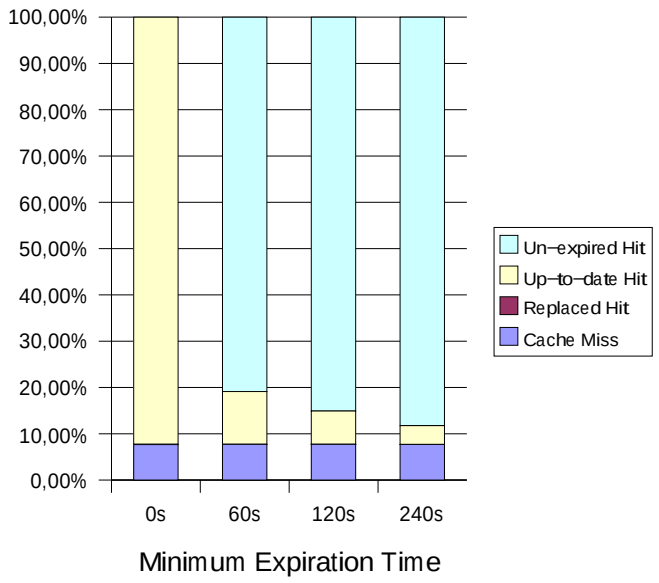


Figure 7.4: Various Minimum Expiration Times: Cache Hit Rate

increase to 48 MB (the size of all fetches documents) does not lead to observably better results. Obviously the most frequently requested documents fit in 16 or even 8 MB.

Figure 7.4 shows how changes in the minimum expiration time affect the cache hit rate.

# Part II

# Prefetching

# 8 Prefetching—An Overview

*We can predict everything, except the future.*

— *From the* fortune *database*

## 8.1 Types of Prefetching

In a Web context, the following types of prefetching can be distinguished [cf. Palpanas 1998, 19f]:

In *client-based* prefetching, the prediction model is built at the client side, using a navigational pattern of the individual(s) using this client. Thus the model covers the behavior of a single or a few users across different Web servers.

In *server-based* prefetching, on the other hand, the model is built at the server, thus covering the usage pattern of all users accessing this specific server. This approach offers the advantage of collecting much more data concerning individual Web pages, thus enabling more exact predictions.

In *combined client and server* prefetching, both approaches are merged by the exchange of prediction data between server and client.

The model can also be built on a proxy (*proxy-based* prefetching), thus covering different users as well as different servers. The proxy can prefetch documents in its own cache; it can also give hints to the client about documents that may be worth prefetching.

Due to the characteristics of a Web application platform, the approach chosen in this thesis is comparable to proxy-based prefetching. The main difference is that the clients are a direct part of the application platform.

The different types of prefetching are shown in figures 8.1 and 8.2.

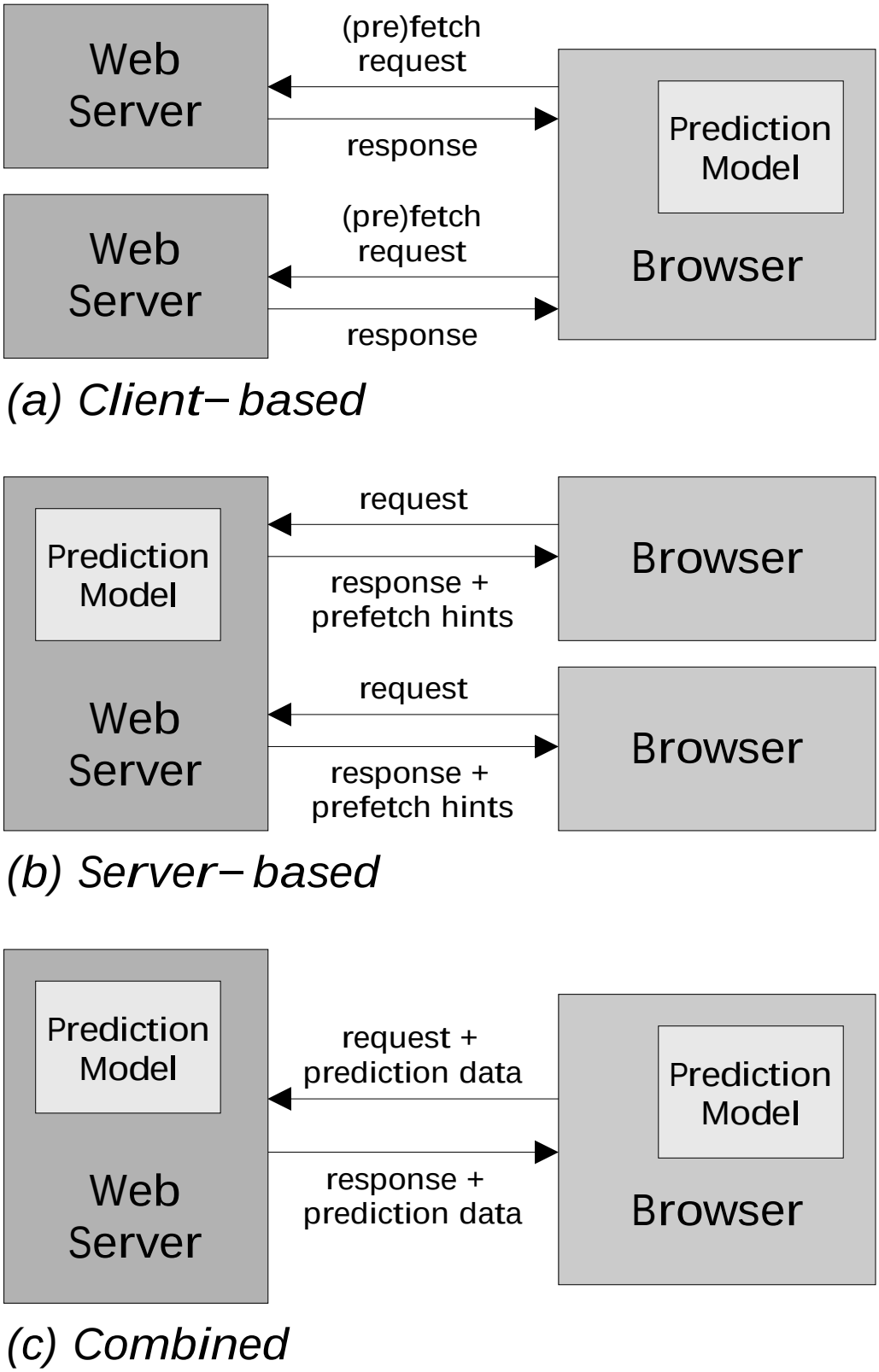


Figure 8.1: Types of Client- and Server-Based Prefetching

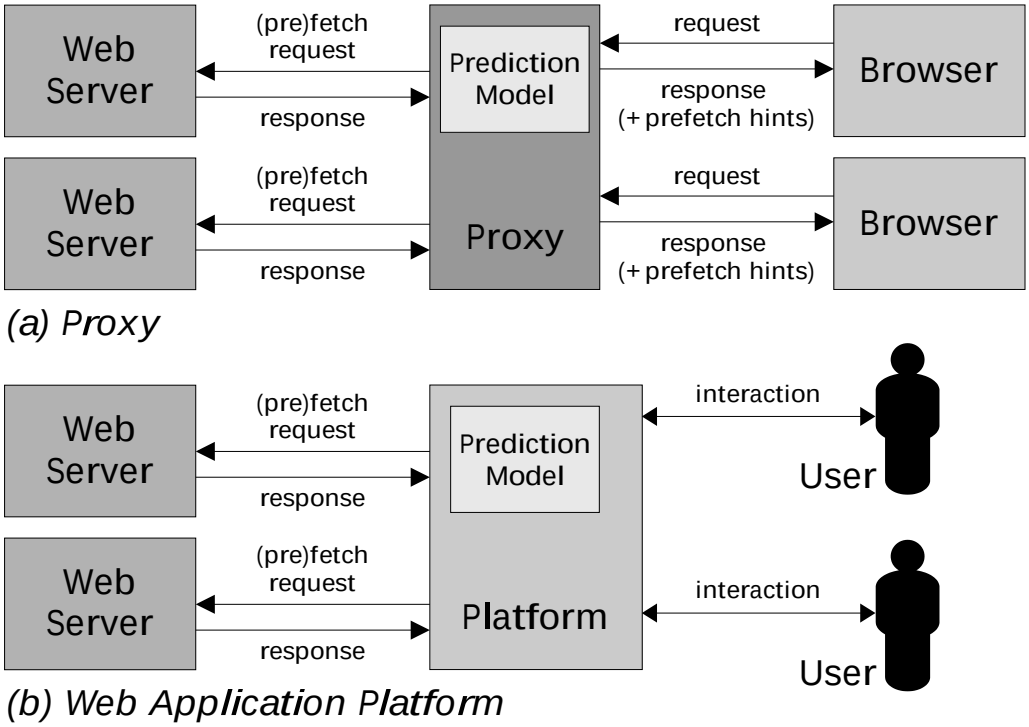


Figure 8.2: Prefetching Based on Proxy resp. Web Application Platform

## 8.2 Prefetching Issues and Limitations

Prefetching techniques are always at danger of crossing the borderline from making things better to making things worse. Prefetching aims at reducing access times to documents by increasing the cache hit rate (cf. sec. 7.1).

But overly aggressive prefetching will often fetch documents that turn out to be unnecessary and may force documents that are more useful (e.g, that will be requested sooner) out of the cache, thus lowering the actual cache hit rate. Prefetching a document only makes sense if its value[1] is higher than the value of the cached documents it replaces.

Also, additional Web connections opened for speculative prefetching may decrease the bandwidth available for necessary downloads, thus increasing the time required to load documents that were not cached or prefetched.

Prefetching is of little or no use when clients spend very little time on a site, requesting only one or two documents, or when access patterns vary enor-

[1] Which depends on the caching strategy—cf. sec. 2.1.

mously. However, this concern applies more to traditional Web applications than to voice applications. In voice applications choices of action are often more limited, making extreme variations less likely. Users cannot start their session at any document, but only at the predefined entrance points mapped to phone numbers, thus typically several steps will be necessary to reach a destination point.

A special problem of prefetching in a Web context are the huge memory requirements of the model, caused by the large number of documents and pathways between them. A way of handling this consists in specifying an upper bound of memory used for the model. When the model reaches the bound, it may either be freezed and used the way it is without further updates; it may be discarded and rebuilt with new data; or these approaches can be combined (freeze the model when it fills half of the memory and start building a new model, while the old model continues to be used for prediction).

Jacobson and Cao [1998, sec. 5.2.1] use an aging process to remove outdated parts of the history structure. The algorithm marks each element in the history structure with a time stamp which is set to the time this element is updated. Periodically it walks through the structure and removes all elements that have not been changed for a certain time.

A similar approach consists in using a caching strategy (e.g., LRU) for removing the least valuable parts of the model before memory requirements get too high. This also prevents outdated information (e.g., requests for pages that no longer exist) to clog up the model.

Memory requirements can also be lowered by storing the data in an efficient form, e.g., by mapping each URL to an integer (in a reversible way) [cf. Palpanas 1998, 35].

Crovella and Barford [1998, sec. I] explore the effects of prefetching on network performance. Simple prefetching approaches ("hoarding", issuing lots of prefetch requests at the same time) increase the burstiness of sources and thus can lead to queues and delays during such peaks in network load.

On the other hand, more refined prefetching can lead to network traffic that is more evenly distributed than without prefetching. This is reached by distributing prefetch requests over time and using periods of low load for prefetching

mechanisms. Thus network performance can actually be increased compared to the situation without prefetching, where periods of idleness are wasted.

# 9 Prefetching Strategies

*And if California slides into the ocean,*
*Like the mystics and statistics say it will.*
*I predict this motel will be standing,*
*Until I've paid my bill.*

*— Warren Zevon, "Desperados Under the Eaves"*

The purpose of prefetching strategies is to predict which documents are likely to be requested next in a session, depending on the documents requested so far. There are two main approaches to do this:

*Semantic prefetching strategies* analyze the content of documents or metadata attached to them to decide which documents will probably be requested. Typical input values of semantic strategies are

- Hyperlinks contained in the document;
- Keywords attached to or extracted from the document.

*Statistic prefetching strategies* base their decisions on statistics built from previous sessions. Considered input values include:

- First-order transition probabilities between a document and the next one;
- Higher-order transition probabilities, depending on several previous documents;
- Time passed between requests.

Table 9.1 shows which input values are utilized by the prefetching strategies presented in this chapter as well as by the *PPM-Time* strategy, introduced in the next one.

| | Links in document | Keywords | 1st-order transition prob. | Higher-order trans. prob. | Time between requests |
|---|---|---|---|---|---|
| **IPS** | x | | | | |
| **Keyword-based** | x | x | | | |
| **Dependency Graphs** | | | x | | |
| **1st-order Markov** | | | x | | x |
| **PPM** | | | x | x | |
| **PPM-Time** | | | x | x | x |
| **LxS** | | | x | x | |

Table 9.1: Input Values of Prefetching Strategies

## 9.1 Semantic Prefetching Strategies

### 9.1.1 Simple Strategies

A naive way of prefetching is called *informed prefetching* [Patterson et al. 1995; Tomkins et al. 1997]. This requires the system to know which requests will take place in the near future and to disclose this knowledge to allow prefetching these requests. While this is useful for batch applications, it cannot be used in interactive applications, where the user chooses what will happen next [cf. Palpanas 1998, 8].

Prefetching has been more a topic for research than for actual deployment, but it is used in some commercial client-side products. These *Web accelerators* use prefetching to reduce latencies in Web browsers [cf. Davison 2001].

However, the algorithms employed are usually very simplistic, such as using the idle time of the Web connection to prefetch all links of the page currently viewed. While this reduces latencies, it also results in a huge increase in network

traffic [cf. Palpanas 1998, 14f].

One such simple approach, called *interactive prefetching scheme (IPS)*, analyzes HTML pages and prefetches some or all of the linked pages. This strategy is based on a paper by Chinen and Yamaguchi [1997] and implemented in the *WWW Collector* proxy server [Wcol 1997].

### 9.1.2 Keyword-based Strategies

The *WebWatcher* system uses interests specified by a user to guess which hyperlinks in the Web page currently viewed this user is most likely to follow. Each user reveals their interests in several keywords. Every page is annotated with the keywords of all the users that have viewed it. The system compares the annotations on the hyperlinks with the keywords of the user to find the closest match [cf. Aumann et al. 1998, 9].

GOTO Software [2001] offers a client-side system that prefetches pages linked from the current page by comparing the link text with areas of interest (i.e., keywords) stated by the user. A similar approach is chosen by Naviscope Software [2001]—by default the software prefetches links that contains the word "next," but it can be configured to look for other keywords.

## 9.2 Statistic Prefetching Strategies

### 9.2.1 Dependency Graphs

A simple prefetching algorithm using dependency graphs is proposed by Padmanabhan and Mogul [1996, sec. 3]. The graph is constructed on a Web server. Each file on the server is a node of the graph. If a client requests a document $B$ during some *lookahead window time* $w$ after it requested a document $A$, an arc leads from $A$ to $B$. Multiple requests of the same document during $w$ by a client are counted only once. The weight of the arc is the number of accesses to $B$ after $A$ divided by the number of accesses to $A$.

A document is considered worth of prefetching if its weight on the arc from the document currently viewed is higher than a *prefetch threshold* $p$. In this

case, the server sends the client a prefetching hint. The decision whether to actually prefetch this document is left to the client. This is a simple variation of the Prediction-by-Partial-Match approach (cf. sec. 9.2.3), using a past history of length 1.

### 9.2.2 First-Order Markov Models

Kraiss and Weikum [1998, 141–149, 154, 160] use a continuous-time Markov chain (CTMC) model to measure prefetching probabilities. They not only measure the transition probabilities between documents requested in the same session, but also the time passed between the requests. Utilizing Markov models, they thus estimate the number of accesses to a document within a certain time span (details of their calculations not covered here can be found in their original paper).

They use only first-order Markov chains, i.e., they only consider the current state (the current document), not the session history (previous documents). For each document $d_i$, $p_{ij}$ is the probability that $d_j$ will be the next document fetched in this session. $H_i$ denoted the average time the session stays on document $d_i$, called *mean residence time*; $p_{ij}(t)$ is the probability of going from state $i$ to state $j$ in time $t$.

Instead of using the individual mean residence time of each state, they calculate a new uniformized CTMC which uses the same mean residence time for each state by allowing a state to return to itself ($d_i$ is followed by $d_i$). $\bar{p}_{ij}(t)$ marks the probability of going from state $i$ to state $j$ in time $t$ using this uniformized model, it is equal to $p_{ij}(t)$ using the original model.

Using this uniformized CTMC, the probability of reaching state $j$ after $m$ steps starting in state $i$ is calculated by the formula

$$\bar{p}_{ij}^{(m)} = \sum_{k=1}^{N} \bar{p}_{ik}^{(m-1)} \bar{p}_{kj} \qquad \text{with } \bar{p}_{ij}^{(0)} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

The sum of the products of the probability that $m$ steps are performed in time $t$ with the $m$-step transition probability for all values of $m$ gives the time-

dependent transition probabilities $p_{ij}(t)$:

$$p_{ij}(t) = \sum_{m=0}^{\infty} e^{-vt} \frac{(vt)^m}{m!} \times \bar{p}_{ij}^{(m)}, \qquad \text{for all } i, j \text{ and } t > 0.$$

This value expresses the probability that a session now residing on document $d_i$ will reside on $d_j$ at time $t$.

To measure the weight of a document up to time $t$, the visits to this document that are already finished at time $t$ must also be considered, yielding the following formula:

$$E_{ij}(t) = \frac{1}{v} \times \sum_{n=1}^{\infty} \left( e^{-vt} \frac{(vt)^n}{n!} \times \sum_{m=0}^{n-1} \bar{p}_{ij}^{(m)} \right).$$

Including all accesses to the document still in progress at time $t$ leads to this formula:

$$E = \sum_{j=1}^{N} v \times E_{ij}(t) \times \bar{p}_{jk}$$

$E$ denotes the number of accesses to the document $d_k$ in the lookahead time horizon $t$, i.e., the "worthiness" of prefetching this document and keeping it in the cache. This model can be extended to cover multiple parallel sessions. The documents with the highest worthiness (weight) not yet cached should be prefetched while the cached documents with the lowest weight can be removed from the cache to make room for worthier documents—if there are no worthier documents, no prefetching takes place.

The context explored in this study involves long response times where a gain of several seconds can be reached by intelligent prefetching decisions, making the large computational overhead feasible. In the typical Web context response times tend to be much faster, so more lightweight models should be considered.

### 9.2.3 Higher-Order Markov Models: Prediction by Partial Match

Palpanas [1998, 17–25, 32–34] proposes an algorithm called *Prediction by Partial Match (PPM)*. This prediction model is adapted from data compression algorithms, where predictions of the next elements of sequences are used to choose effective compression schemes. Based on a probability distribution of the data,

frequent sequences are encoded using few bits. The less frequently sequences occur, the more bits are used to encode them. Such a probability distribution is useful to prefetching as well, where the pages that are most likely to be required should be prefetched.

The compression algorithms that turn out to give best results use *context modeling* to calculate a prediction model. For text compression, the probability is calculated using higher-order Markov models, i.e., the probability of a character is calculated depending on the last $n$ characters preceding it. $n$ is called the *order* of the model. The predictive precision of the model increases with $n$, but so does the amount of data that must be stored in the prediction model. Also, for larger $n$'s (models of higher order), most contexts are seen seldom or never. In these cases, the model cannot be used reliably.

A solution to this problem is to keep probability models of different orders and switch to the next lowest order if no data or not enough data is available in the higher order model. Even better is *blending* the different order models, e.g., combining their predictions into a single probability.

This algorithm can easily be adapted to Web prefetching. Instead of sequences of characters, sequences of Web page requests are measured. Thus a probability model of the way users navigate in the Web is build.

For building the model, an *order-m* prefetcher contains $m + 1$ *Markov predictors* of order 0 to $m$. An *order-k Markov predictor* calculates the probability $p(P|P_k P_{k-1} \dots P_1)$, i.e., the conditional probability $p$ that a page $P$ is accessed after sequential accesses to pages $P_1, P_2, \dots, P_k$. Thus an order-$k$ Markov predictor represents an context of length $k$.

The context 0 does not capture a conditional probability, but the unconditional probability that a document is fetched. Considering the huge number of documents on a Web site, this probability is of little or no real use and may be omitted.

Predictions can be made in different variations: either only events of the highest available order are considered, or predictions from several orders can be combined (blended). Another possibility would be to report the results of several orders, assigning a certain level of trust to each of them.

Fetches in previous sessions:

| | | | |
|---|---|---|---|
| A | B | C | M |
| A | B | C | M |
| A | B | C | N |
| A | B | C | O |
| A | B | D | M |
| A | B | D | P |
| E | B | C | M |
| F | B | C | O |

Resulting prediction model:

| Session History | | | Hint | Prob. |
|---|---|---|---|---|
| A | B | C | M | 50% |
| | | | O | 25% |
| | | | N | 25% |
| A | B | D | M | 50% |
| | | | P | 50% |
| E | B | C | M | 100% |
| F | B | C | O | 100% |

| History | | Hint | |
|---|---|---|---|
| B | C | M | 50% |
| | | O | 33% |
| | | N | 17% |
| B | D | M | 50% |
| | | P | 50% |

Figure 9.1: Sample Prediction Model of the PPM Strategy

The prefetched documents are put in the same cache as normally fetched pages, thus replacing the least valuable documents. Palpanas [1998] employs an LRU cache, but the algorithm would work with other replacement strategies without changes.

This prefetching algorithm could be enhanced by enabling predictions for a longer time span, thus predicting several sequential requests. Another extension would be to make the algorithm adapt to server load, by reducing the amount of prefetching when load is high [cf. Palpanas 1998, 58f].

Another study using a PPM-based algorithm found a decrease in perceived latency by 23.4% [Fan et al. 1999, 1, 5–8]. They use a variation of the algorithm that can be configured to predict several steps in the future instead of just one (the number of sequential accesses predicted is called the *search depth* $l$). Doc-

uments are prefetched only if their probability of being requested surpasses a threshold value $t$.

Their algorithm reached an prediction accuracy of 40% (with $n, l, t = 2, 4, 0.125$) to 73% $(1, 1, 0.5)$, depending on the selected parameters, while causing an increase of 1% to 15% in traffic.

Figure 9.1 shows a simple partial prediction model based on several sessions. Markov predictors of order three and two are used in this example.

### 9.2.4 A Data Mining Approach: LxS

The decision which of the proposed prefetching algorithms to employ crucially depends on this question: do future requests depend on the current document only or must previous requests be taken into account? In the first case, a first-order Markov chain is sufficient to describe the event sequences accurately. Otherwise higher-order Markov chains must be used to keep a past history of several documents.

This question was studied by Aumann et al. [1998, 1–7, 9]. Their conclusion was that a first-order Markov model is not sufficient in the domain of Web access patterns, resulting in a 20% loss of accuracy compared to a data mining algorithm based on past histories.

Their data mining algorithm, called *LxS (Length times Support)*, is trained offline on a set of training sequences before making predictions. A *sequence* $S$ is a tuple of events: $S = (e_1, \ldots, e_k) \in E^*$. The prediction of an algorithm is correct for a sequence $S = (e_1, \ldots, e_k, e_{k+1})$ if it gives the prediction $e_{k+1}$ on the subsequence $S' = (e_1, \ldots, e_k)$. The algorithm receives a set of training sequences, $X = \{S_1, \ldots, S_n\}$.

The *support* of a subsequence $S' = (g_1, \ldots, g_m)$ in a sequence $S = (e_1, \ldots, e_k)$ is the number of times $S'$ appears in $S$ as a subsequence:

$$sup(S, S') = \left|\{i : g_j = e_{i+j}, \forall j = 1, \ldots, m\}\right|.$$

The support of $S'$ in the training set $X$ is the number of times $S'$ appears in all sequences in $X$:

$$sup_X(S') = \sum_{S \in X} sup(S, S').$$

A subsequence is *frequent* if its support is at least $\sigma$, where $\sigma$ is an application-dependent threshold value. The algorithm computes the support of all frequent sequences, or—if the number of frequent sequences is too large due to a small value of $\sigma$—of all frequent sequences that are not longer than a maximum length $\tau$.

The algorithm can predict the next event $e$ given a sequence $S = (e_1, \ldots, e_m)$ by calculating all possible *extensions* of the sequence $S' = (e_1, \ldots, e_m, e)$ for all values of $e$. For the sequence $S$ and each event $e$, the $(e, i)$-*extension*$(S)$ denotes the extension of the last $i$ events of $S$ with the event $e$, for $i = 1, \ldots, m$:

$$(e, i)\text{-}extension(S) = (e_{m-i+1}, \ldots, e_m, e).$$

The *weight* of an $(e, i)$-*extension*$(S)$, S', is proportional to both the length of $S'$ and the support:

$$weight(S') = \begin{cases} sup(S') \times i & \text{if } S' \text{ is frequent,} \\ 0 & \text{otherwise.} \end{cases}$$

Then the *strength* of a prediction $e$ is computed by adding the weights of all extensions ending in $e$:

$$strength(e) = \sum_{i=1}^{m-1} weight((e, i)\text{-}extension(S)).$$

The extension that has the highest strength is selected as prediction:

$$predict(S') = e \qquad s.t. \qquad strength(e) = \max\{strength(g) : g \in E\}.$$

When no frequent sequence matches the suffix of the sequence, no prediction is made. On the data set used to evaluate the algorithm, it made a prediction in 61.9% of all cases (called the *coverage* of the algorithm).

The algorithm gave the best results when the threshold value $\sigma = 5$ and a maximum length $\tau = 5$ were used. Using these values the algorithm reached an accuracy of 41.8% compared to 33.6% of a first-order Markov algorithm.

The authors of the study propose to create a hybrid algorithm, using the first-order Markov algorithm in the cases not covered by their LxS algorithm, thus reaching a coverage of 100% without losing the predictive power of their algorithm.

# 10 Analysis of the Prefetching Solution

*Not included in this version.*

# 11 Design, Implementation, and Integration of the Prefetching Solution

*Not included in this version.*

# 12 Evaluation of the Prefetching Solution

## 12.1 Measuring the Performance of Prefetching

To judge the success of a prefetching system and to tune the parameters used, the performance of the system must be measured. The following criteria can be used to do this [cf. Palpanas 1998, 38f]:

**Usefulness of predictions/prefetches:** The percentage of fetched pages that had already been predicted resp. prefetched.

**Accuracy of predictions/prefetches:** The percentage of predicted resp. prefetched pages that were later actually requested by the user.

**Practical accuracy of predictions:** The probability that one of the received predictions was correct.[1]

**Coverage:** The percentage of actual fetches which were preceded by predictions.

**Network traffic increase:** The volume of network traffic with prefetching enabled divided by the volume of traffic without prefetching—one of the costs of prefetching.

While the system should aim to maximize the first four metrics, the last one should of course be minimized. These criteria comprise partially conflicting objectives, e.g., while prefetching more pages should increase the usefulness of

[1] When several predictions are received for the same document, only one of them can be correct. Therefore an accuracy of 100% is impossible to reach, but a practical accuracy of 100% could be reached if the actually fetched document was always among the predicted ones.

predictions, it will cause higher network traffic and typically also decrease the accuracy of predictions.

## 12.2 How and What to Evaluate

The test driver used to evaluate caching (cf. sec. 7.2) is also used to measure the performance of prefetching. The criteria described above are used.

The overall coverage (called *combined coverage*) is measured in two steps, so both the prefetching implementation and the prediction model can be judged:

**Process coverage:** The percentage of actual fetches which were preceded by a call for predictions, even if no predictions were found. This measures the efficiency of the prefetching implementation—for all other actual fetches the prefetcher was too slow.

**Model coverage:** The percentage of "process covered" actual fetches for which at least one prediction was received. This measures the scope of the prediction model.

**Combined coverage:** The product of these two steps:

$$\textit{combined coverage} = \textit{process coverage} \times \textit{model coverage},$$

i.e., the percentage of actual fetches for which at least one prediction was received.

## 12.3 Evaluation Results

### 12.3.1 Default Setting

The setting used to test the cache was also used to test the prefetcher (cf. sec. 7.3). Figure 12.1 shows the coverage reached in this default setting. While the model coverage reaches almost 80%, the process coverage moves in quite lower ranges. This depends on the load, i.e., the number of parallel sessions (cf. sec. 12.3.3).

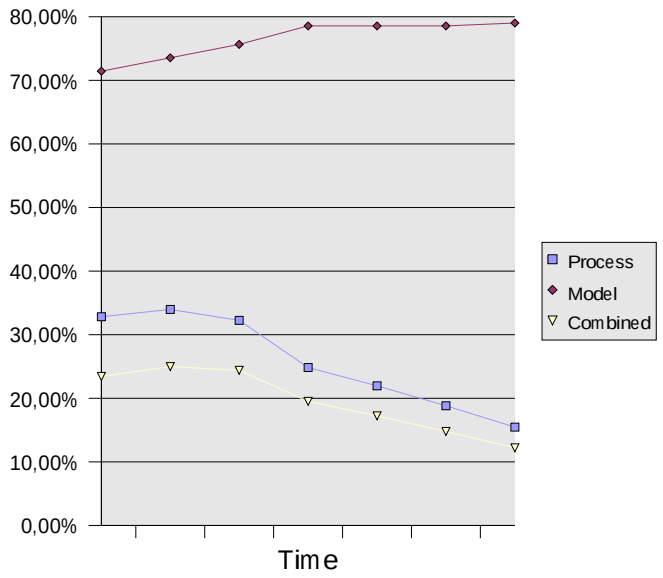


Figure 12.1: Prefetching Default: Coverage

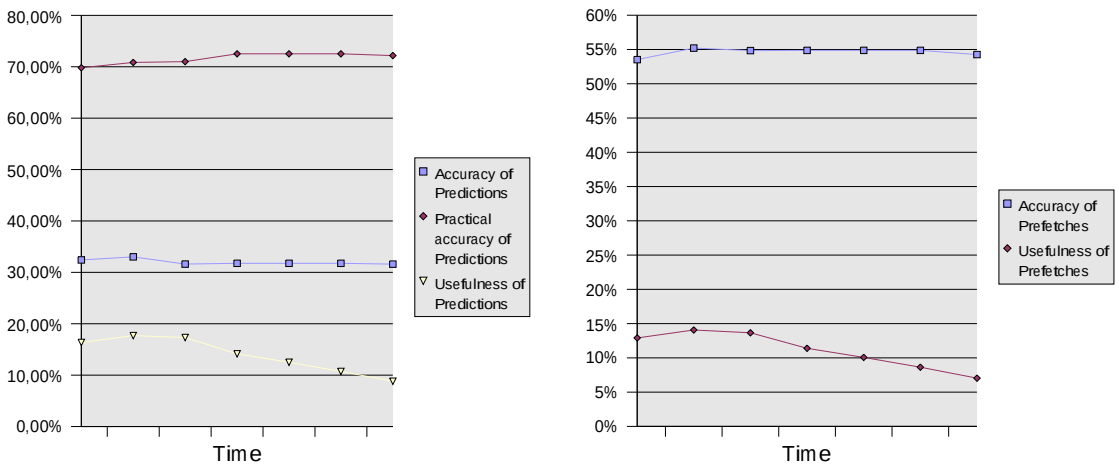


Figure 12.2: Prefetching Default: Accuracy and Usefulness

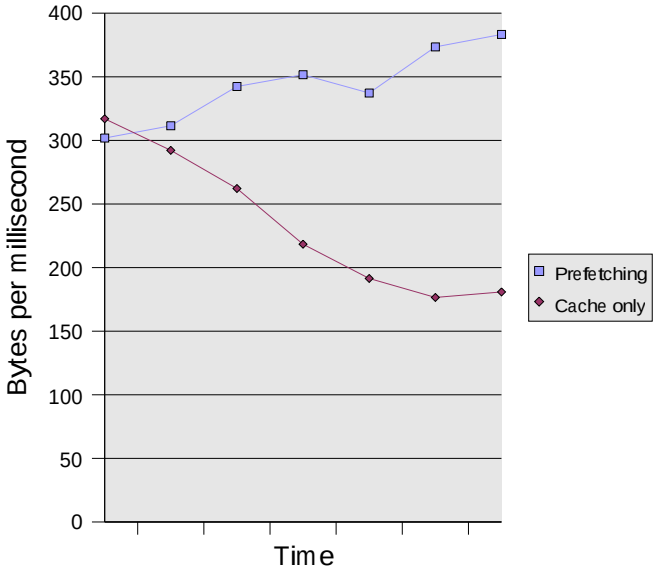


Figure 12.3: PRR With and Without Prefetching

The accuracy and usefulness reached by prefetching are shown in figure 12.2. The usefulness is lower than the combined coverage reached, as must be the case. The overall PRR increase gained by prefetching is shown in figure 12.3.

Typically, prefetching is supposed to lead to additional network traffic, but in the setting tested here the network traffic is actually lowered by prefetching (fig. 12.4). This surprising result is explained by an implementation detail: the `PrefetchingFetcher` avoids parallel fetches of the same document, which are allowed by the `CachingFetcher` to reduce synchronization overhead.

Figure 12.4 also shows the network traffic reduction reached by this feature alone (without prefetching). While enabling prefetching increases the traffic compared to this setting, the traffic is still lower than in the original setting.

### 12.3.2 Configuration Variations

In this section, several prefetching parameters are varied. Figure 12.5 shows how these configuration variations affect the PRR.

Figure 12.6 shows the usage of session histories of length three, four (the default), and five. While increasing the length of the session history leads to a

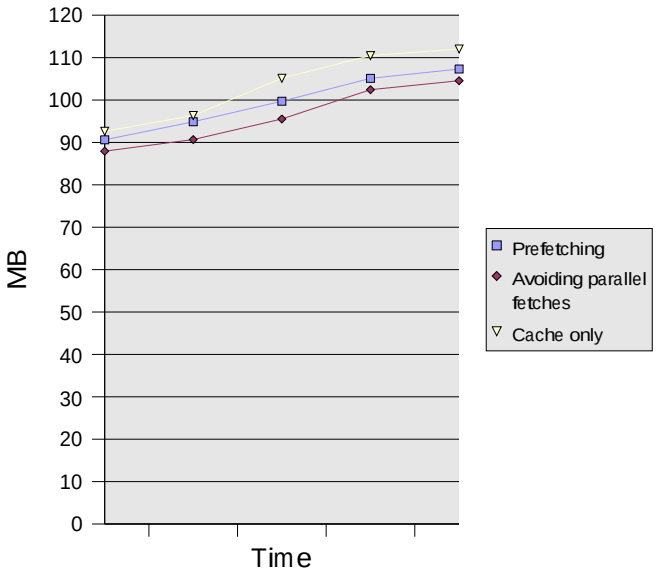


Figure 12.4: Network Traffic

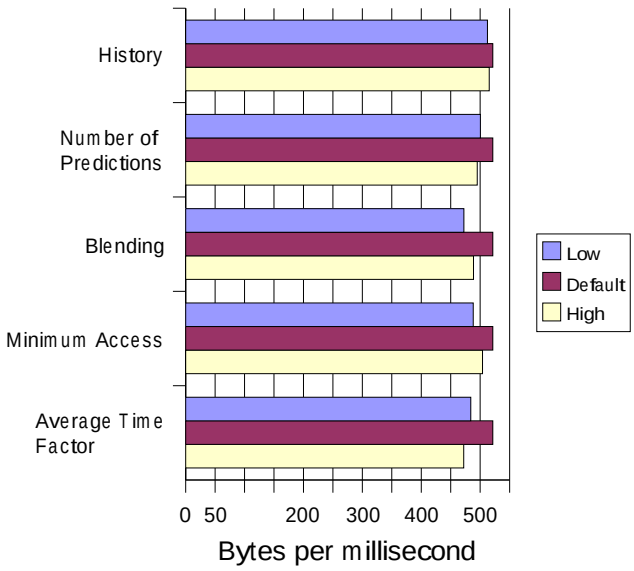


Figure 12.5: Various Settings: PRR

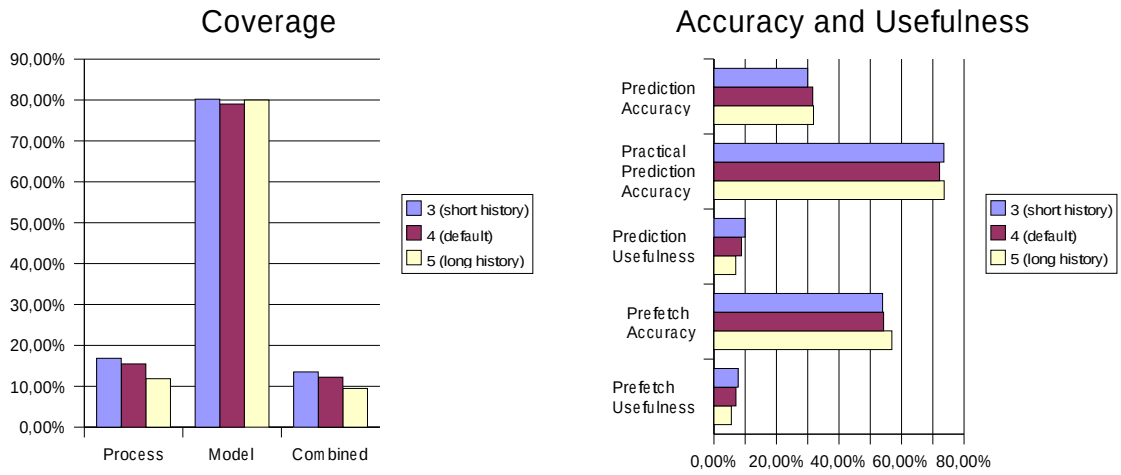


Figure 12.6: Various History Lengths

higher accuracy, it decreases the process coverage, due to the higher computation effort. In result, neither the shorter nor the longer history can reach the PRR values of the default length (cf. fig. 12.5).

How the number of predictions made affects the results is shown in figure 12.7. The accuracy of individual predictions falls considerable when this number is increased, while the practical prediction accuracy does only slightly rise. A medium value turns out to be best for the PRR. A similar effects arises when blending predictions of different orders—blending more orders also means receiving more predictions (fig. 12.8).

Figure 12.9 shows different values of the `prefetching.minimumaccesses` parameter, i.e., the number of accesses a document must have had to be considered worth of prefetching. The lower this value, the higher is the coverage reached by the prediction model, as documents can be predicted earlier. On the other hand, the process coverage sinks when lowering this value—more documents to consider means more computation effort. As the result, the combined coverage stays almost the same. The prefetch accuracy is slightly increased with a higher value, while the usefulness is reduced. The PRR is best served by a medium value (cf. fig. 12.5).

Concerning the decision of when to prefetch, a medium value also turns out to be best (fig. 12.10). Using a higher value for the `prefetch-`

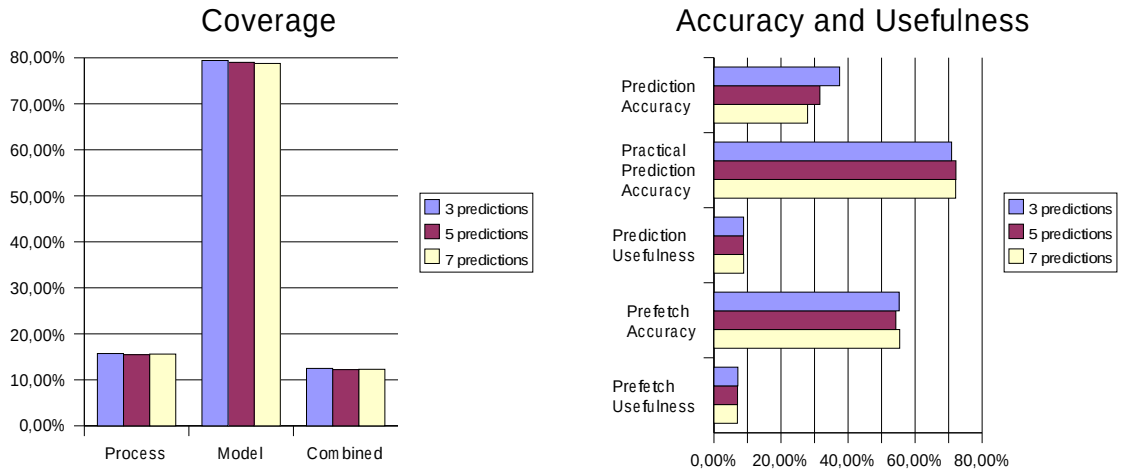


Figure 12.7: Variations in the Number of Predictions

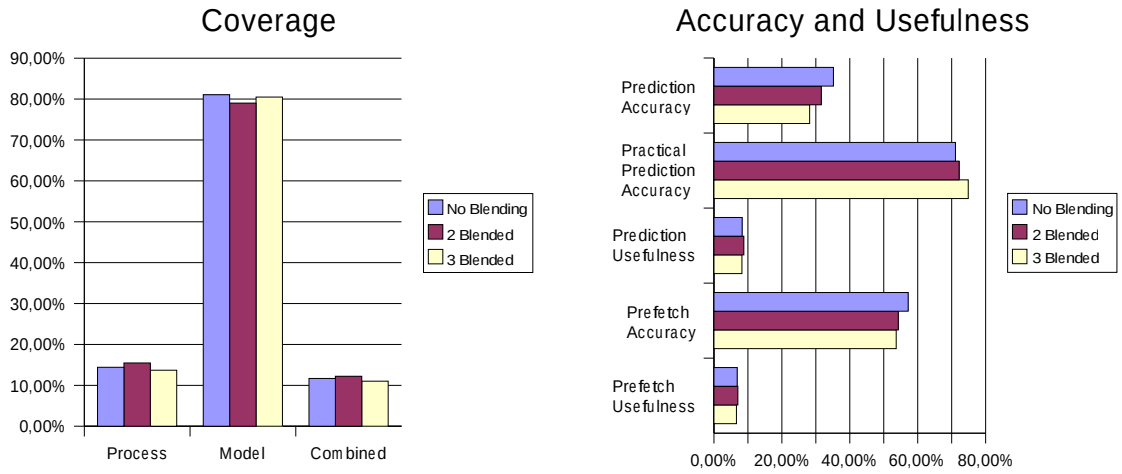


Figure 12.8: Blending Variations

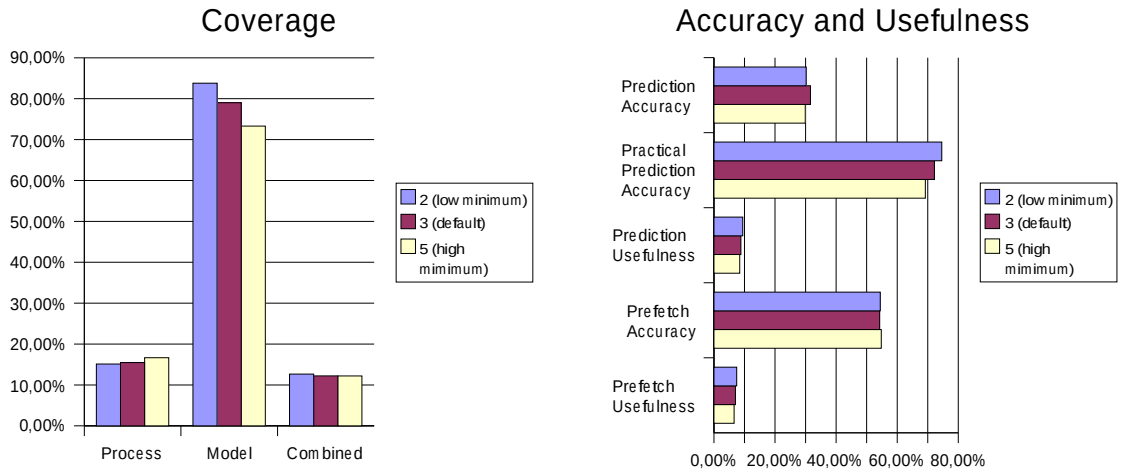


Figure 12.9: Minimum Access Variations

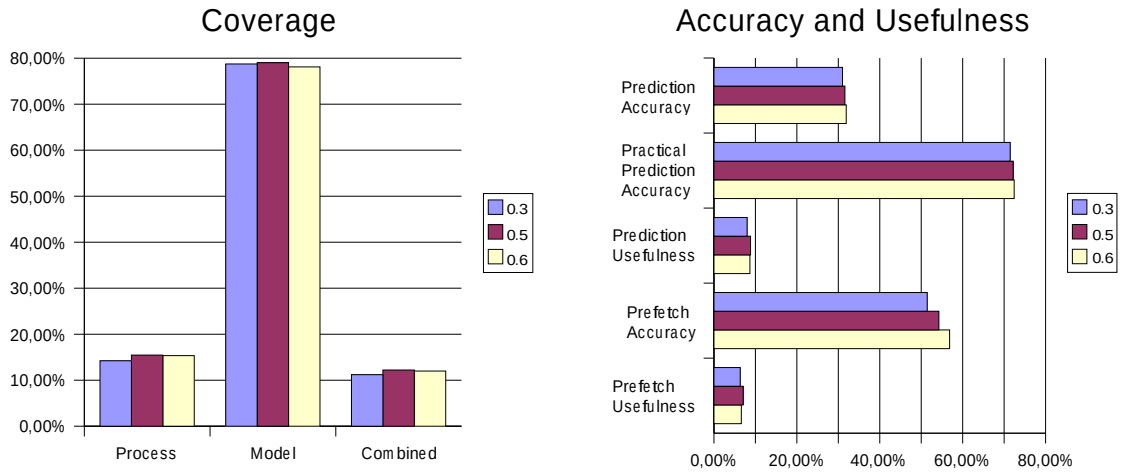


Figure 12.10: Variations of the Average Time Factor

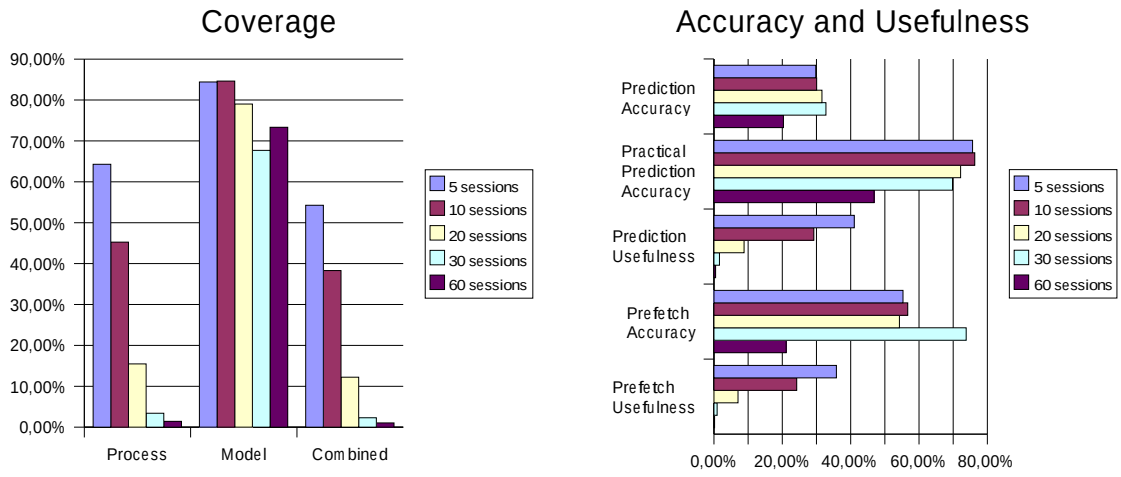


Figure 12.11: Load Variations

`ing.averagetime.factor`, i.e., waiting longer before starting a prefetch, leads to a higher accuracy. However, waiting too long reduces the usefulness.

### 12.3.3 Load Variations

Figure 12.11 shows how enormously the process coverage—and thus not only the combined coverage, but also the usefulness—depends on the number of parallel sessions, i.e., the load of the system.

The PRR increase reached by prefetching for different amounts of load is shown in figure 12.12. While prefetching always offers an advantage compared to the cache-only case, this advantage becomes considerably smaller for a huge number of sessions.

## 12.4 Design Change Caused by the Evaluation

*Not included in this version.*

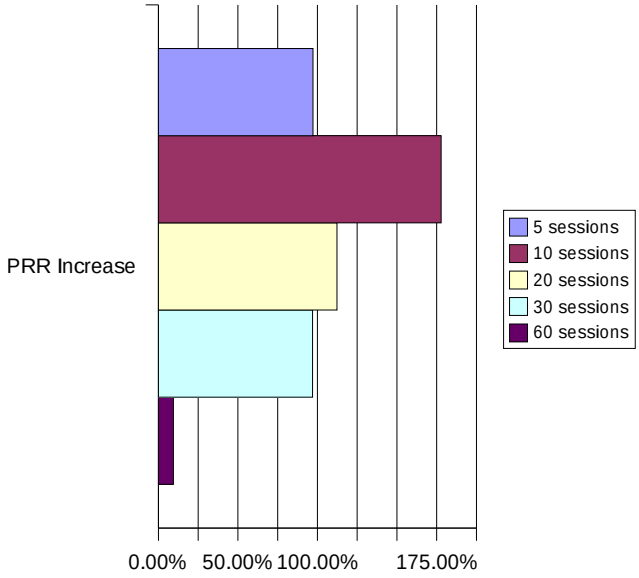


Figure 12.12: Load Variations: PRR Increase

# 13 Conclusion

*We attend and amend*
*And contend and defend*
*Without a conclusion in sight.*
*— Leslie Lipson, "The Committee"*

## 13.1 Résumé

I have designed, implemented, and documented a toolkit for caching documents retrieved from URLs as well as any other content. Both file and memory caches are used; the toolkit allows the integration of generic parsers and can store documents in parsed state. The expiration and validation rules of HTTP 1.1 are followed.

To reach maximum flexibility, the architecture of the toolkit is pluggable, so different replacement strategies can be integrated. The LRU (Least Recently Used) strategy is provided as a reference implementation.

The toolkit is integrated with the MUNDWERK voice gateway, a Web application platform for voice applications based on the VoiceXML markup language. The attributes and properties defined by VoiceXML (both version 1.0 and the drafted version 2.0) for controlling caching and prefetching are handled. But the usage of the toolkit is not limited to this context—support for other settings and standards can be added as part of other integrations. No changes to the basic toolkit are required.

The toolkit is extended to support prefetching in a transparent way. The implemented prefetching strategy is based on the PPM (Prediction by Partial

Match) strategy, which utilizes higher-order Markov models. It is extended to consider the time spans between requests. A blending algorithm is used to combine the results of Markov predictors of different orders.

The prefetching strategy is pluggable as well, so the architecture of the toolkit permits complementing or replacing this approach by other strategies. Prefetching is handled asynchronously, using primarily system resources that are not needed for other purposes and would be wasted otherwise.

The evaluation shows the effectiveness of this solution—in test runs, prefetching did increase the PRR (perceived retrieval rate) by up to 175%, without causing a significant increase in network traffic.

Since January 2002 the toolkit has been deployed as part of a production system. This makes it probably unique, because refined prefetching strategies so far have been mainly a research topic, while deployed systems typically use much simpler prefetching approaches, if at all.

## 13.2 Extensions and Further Work

An obvious field of extensions is the integration of the toolkit. New ways of usage can be added to the existing integration with the MUNDWERK voice gateway; or the toolkit can be integrated with other Web application platforms.

One extension of the existing integration is already planned: using a file cache for storing audio prompts synthesized by a TTS engine, so text already synthesized can be re-used. This extension is scheduled for realization in the near future.

Some possible extensions of the toolkit have already been discussed in this work: other caching resp. prefetching strategies could be implemented and evaluated in addition to the reference implementations provided in this work (cf. sec. 7.1); the search depth of the prefetching algorithm could be extended to include several next documents instead of just one; the general fetch statistics could be complemented by personalized ones for covering specific areas, like the first few documents of a session.

Currently the configuration of the toolkit is done manually. This could be partially automated by developing meta-heuristics for dynamically adapting

specific configuration parameters, depending on the performance metrics received. In the case of prefetching, parameters that might be adapted this way include the length of the session history, the number of predictions received, and the time span during which prefetch requests are combined.

## 13.3 Lessons Learned

The purpose of this final section is not to list further findings proved by this work, just to record some of the more important things I noticed.

Statistic and heuristic approaches can be useful where strict formalization and guaranteed correctness are impossible to reach. To not allow wrong decisions would make prefetching impossible in most cases. Informed prefetching can work only in a small part of the area covered by statistic prefetching strategies; it cannot work at all in interactive settings where the user decides what happens next.

Statistic or heuristic approaches do make some predictions which turn out to be wrong. But by allowing errors, they can work—and work well—in situations inaccessible to algorithms that postulate certainty. Just as in "real life," it can be advantageous to renounce absolute security.

Often, simple approaches can be almost as useful as extremely refined algorithms. The latter might not even be made to work as well in practice and might depend on a very special setting, requiring complex adaptations to fit little changes in the setting.

An example of this are the error-handling mechanisms discussed in this work. The approach of link analysis is a complex algorithm which in theory should reach a high level of exactness, but which would be very hard (if not impossible) to implement in practice without leaks that would degrade its results probably below the level of the simple blocking mechanism employed instead.

Additionally, link analysis would require constant adaptation to reflect each change in the content formats it needs to evaluate (VoiceXML, different grammar formats, scripting languages etc.). On the other hand, the blocking mechanism employed instead is sloppy and will necessarily let pass through some erroneous fetches, but it is easy to implement and completely independent of

the transmitted content. Thus it doesn't require any changes to reflect modifications and can be employed in completely other contexts as well.

In other cases, a complex solution might work as well or slightly better than a simple one, but may carry a huge overhead that is not justified by the comparatively small differences in the results. An example of this is the decision whether to use complex replacement strategies like GDS or the simple LRU strategy.

But, of course, simpler is not always better. Sometimes, especially for core issues, complexity does make sense. The PPM strategy is one of the most complex prefetching strategies, but its usage is justified due to its superior results.

Part of the art lies in deciding when complexity is appropriate. There are no formulas to solve this question; in each case, the individual requirements must be considered carefully. It is especially important to resist the temptation of implementing things because they can be implemented. After all, a solution should not function only in principle, but should work reliably and for the long term in practice.

# Bibliography

Aumann, Yonatan; Oren Etzioni; Ronen Feldman; Mike Perkowitz; and Tomer Shmiel (1998). Predicting Event Sequences: Data Mining for Prefetching Webpages. Submitted to KDD'98, Mar. 1998. Online (access date: 2001-10-01): http://citeseer.nj.nec.com/aumann98predicting.html.

Cao, Pei and Sandy Irani (1997). Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pp. 193–206. Online (access date: 2001-10-12): http://citeseer.nj.nec.com/cao97costaware.html.

Chinen, Ken-ichi and Suguru Yamaguchi (1997). An Interactive Prefetching Proxy Server for Improvement of WWW Latency. In *Proceedings of the Seventh Annual Conference of the Internet Society (INET'97)*. Online (access date: 2001-10-17): http://www.isoc.org/INET97/proceedings/A1/A1_3.HTM.

Crovella, Mark and Paul Barford (1998). The Network Effects of Prefetching. In *Proceedings of IEEE Infocom '98*. Online (access date: 2001-10-04): http://bbcr.uwaterloo.ca/~brecht/courses/756/readings/prefetching/net-effects-INFOCOM98.ps.

Davison, Brian D. (2001). *Improving Browser Caches with Extensions and Personal Proxies*. Online (access date: 2001-10-18): http://www.web-caching.com/personal-caches.html.

Dilley, John; Martin Arlitt; and Stéphane Perret (1999). *Enhancement and Validation of Squid's Cache Replacement Policy*. Tech. Rep. HPL-1999-69, HP Lab-

oratories, Palo Alto, CA. Online (access date: 2001-10-12):
http://www.hpl.hp.com/techreports/1999/HPL-1999-69.html.

Edgar, Bob (2001). *The VoiceXML Handbook*. CMP, New York.

Fan, Li; Pei Cao; and Quinn Jacobson (1999). Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance. In *Measurement and Modeling of Computer Systems*, pp. 178–187. Online (access date: 2001-10-01):
http://citeseer.nj.nec.com/fan99web.html.

GOTO Software (2001). *WebEarly 3*. Online (access date: 2001-10-18):
http://www.goto-software.com/us/WE3/WEstart.htm.

Hosseini-Khayat, Saied (1998). Replacement Algorithms for Object Caching. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, pp. 90–97. Online (access date: 2001-10-01):
http://www.acm.org/pubs/citations/proceedings/sac/330560/p90-hosseini-khayat/.

Jacobson, Quinn and Pei Cao (1998). *Potential and Limits of Web Prefetching Between Low-Bandwidth Clients and Proxies*. Tech. Rep. CS-TR-1998-1372, University of Wisconsin-Madison. Online (access date: 2001-10-01):
http://citeseer.nj.nec.com/jacobson98potential.html.

Kraiss, Achim and Gerhard Weikum (1998). Integrated Document Caching and Prefetching in Storage Hierarchies Based on Markov-Chain Predictions. *VLDB Journal: Very Large Data Bases*, 7(3):141–162. Online (access date: 2001-10-01):
http://citeseer.nj.nec.com/kraiss98integrated.html.

Naviscope Software (2001). *Naviscope—Powerful Browser Acceleration & Enhancement*. Online (access date: 2001-10-18):
http://www.naviscope.com/.

Padmanabhan, Venkata N. and Jeffrey C. Mogul (1996). Using Predictive Prefetching to Improve World Wide Web Latency. In *Proceedings of the SIGCOMM '96 Conference*. Online (access date: 2001-10-01):
http://citeseer.nj.nec.com/padmanabhan96using.html.

Palpanas, Themistoklis (1998). *Web Prefetching Using Partial Match Prediction*. Tech. Rep. CSRG-376, Departement of Computer Science, University of Toronto. Online (access date: 2001-10-01):
http://citeseer.nj.nec.com/palpanas98web.html.

Patterson, R. Hugo; Garth A. Gibson; Eka Ginting; Daniel Stodolsky; and Jim Zelenka (1995). Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 79–95. ACM Press, Copper Mountain, CO. Online (access date: 2001-10-09):
http://citeseer.nj.nec.com/patterson95informed.html.

Reddy, Mike and Graham P. Fletcher (1998). An Adaptive Mechanism for Web Browser Cache Management. *IEEE Internet Computing*, 2(1). Online (access date: 2001-10-01):
http://www.comp.glam.ac.uk/pages/staff/mreddy/ICPaper/.

Tomkins, Andrew (1997). *Practical and Theoretical Issues in Prefetching and Caching*. Ph.D. thesis, Carnegie Mellon University. Online (access date: 2001-10-09):
http://citeseer.nj.nec.com/tomkins97practical.html.

Tomkins, Andrew; R. Hugo Patterson; and Garth A. Gibson (1997). Informed Multi-Process Prefetching and Caching. In *Measurement and Modeling of Computer Systems*, pp. 100–114. Online (access date: 2001-10-09):
http://citeseer.nj.nec.com/tomkins97informed.html.

VoiceXML Forum (2002). *Homepage*. Online (access date: 2002-02-21):
http://www.voicexml.org/.

W3C (2001a). Speech Synthesis Markup Language Specification for the Speech Interface Framework. W3C Working Draft, 3 January 2001. Online (access date: 2002-01-16):
http://www.w3.org/TR/speech-synthesis.

W3C (2001b). Voice Extensible Markup Language (VoiceXML) Version 2.0. W3C Working Draft, 23 October 2001. Online (access date: 2001-10-26):
http://www.w3.org/TR/voicexml20/.

Wcol (1997). *WWW Collector—The Prefetching Proxy Server for WWW*. Online (access date: 2001-10-17):
http://infonet.aist-nara.ac.jp/products/wcol/.

Wessels, Duane (1995). Intelligent Caching for WWW Objects. In *INET '95 Proceedings*. Online (access date: 2001-10-01):
http://www.isoc.org/HMP/PAPER/139/.

Wessels, Duane (2001). *Web Caching*. O'Reilly, Sebastopol, CA.

XML Cover Pages (2002). *VoiceXML Forum (Voice Extensible Markup Language Forum)*. Online (access date: 2002-02-21):
http://xml.coverpages.org/vxml.html.

# Appendix

# A Implementation Issues

*Not included in this version.*

# B Zusammenfassung in deutscher Sprache

In meiner Diplomarbeit habe ich ein Toolkit für das Cachen von Dokumenten aus dem Internet und anderen Inhalten entworfen und implementiert. Sowohl Hauptspeicher wie auch Dateisystem werden zum Speichern genutzt. Durch die Integration von externen Parsern können Dokumente in geparstem Zustand gespeichert werden. Die im HTTP-Protokoll (Version 1.1) festgelegten Richtlinien zu Gültigkeit und Validierung werden eingehalten.

Die Architektur des Toolkits ist modular, so dass verschiedene Ersetzungsstrategien integriert werden können. Als Referenzimplementierung kommt die LRU-Strategie (Least Recently Used) zum Einsatz.

Das Toolkit ist in das MUNDWERK Voice Gateway integriert, eine Plattform für sprachbasierte WWW-Anwendungen, die auf der Textauszeichnungssprache VoiceXML basieren. Dabei werden die vom VoiceXML-Standard festgelegten Attribute und Eigenschaften ausgewertet, die das Caching und Prefetching beeinflussen. Das Toolkit kann in andere Umgebungen integriert werden, ohne Änderungen an seiner Architektur zu erfordern.

Eine Erweiterung des Toolkits erlaubt das Prefetching von Dokumenten. Es findet auf transparente Weise statt, ohne äußerlich sichtbare Änderungen. Die verwendete Prefetching-Strategie basiert auf dem PPM-Verfahren (Prediction by Partial Match), das Markov-Modelle höherer Ordnung einsetzt. Zusätzlich wird der zeitliche Abstand zwischen dem Laden von Dokumenten berücksichtigt. Ein Blending-Algorithmus kombiniert die Vorhersagen von Markov-Prediktoren verschiedener Ordnung.

Die modulare Architektur macht es möglich, diesen Ansatz durch andere Strategien zu ergänzen oder zu ersetzen. Das Prefetching wird asynchron durchgeführt, so dass Systemressourcen zum Einsatz kommen, die nicht für andere Zwecke benötigt werden.

In der Evaluation steigerte das Prefetching die PRR (Zahl der per Millisekunde übertragenen Bytes) um bis 175 %, ohne die Netzwerkbelastung wesentlich zu erhöhen. Während komplexe Prefetching-Strategien bis jetzt hauptsächlich ein Forschungsthema sind, kommen sie mit diesem Toolkit in einem Produktionssystem zum Einsatz.