Adaptive fragment designs to reduce the latency of web caching in content aggregation systems

A dissertation presented as partial fulfilment of the requirements for the degree of Ph.D. in the subject of Computer Science

Presented by:
Carlos Guerrero Tomé

Supervised by:
Dr. Carlos Juiz García

Palma, Spain. 2012

Universitat de les Illes Balears
Departament de Ciències Matemàtiques i Informàtica

Universitat de les Illes Balears

A mi madre y a Judit

Agradecimientos

Como buen aficionado a la montaña, quiero utilizar un símil que me explicaron una vez mientras que hacía el Camino de Santiago, pero esta vez, adaptado al proceso de realización de mi tesis doctoral. Cuando sales a la montaña con un grupo de gente, tienes una ruta fijada y marcada, con un inicio claro pero un fin, que no todas las veces se consigue. Dicha ruta, se puede ver alterada por las dificultades que te encuentras en el camino (meteorología, terreno, etc.). Además, cuando haces esa ruta, hay ratos que estas junto a una gente del grupo que te acompaña, otras veces con otros, la gente que te ayuda o a la que ayudas para superar ciertos obstáculos también va cambiando. Pero en mayor o menor medida, todos ayudan y cada uno aporta algo en la consecución del objetivo.

Una tesis, y un proceso de investigación, es igual. Sabes cuales son las condiciones con las que empiezas la investigación y conoces tu objetivo, pero no tienes claro como vas a llegar hasta él y el camino final suele ser diferente del planificado. Te encuentras con dificultades que te hacen retroceder y probar otros caminos. La gente que te rodea te ayuda y te da apoyo en la medida de sus posibilidades, pero todos ellos son indispensables. Desde estas líneas quiero agradecer la ayuda que he recibido de toda la gente que me ha apoyado y acompañado durante los años de realización de mi tesis.

En primer lugar, quiero agradecer a Carlos Juiz su ayuda, apoyo, y su papel de *guía* en la realización y consecución de esta tesis. Gracias a él, he conseguido madurar como investigador, docente y profesional. Él me ayudó en el inicio, me enseñó todo lo que había que saber para empezar en este mundo, y me orientó en las dificultades, cuando no sabía por donde seguir. Ha sido un excelente *jefe* y un gran amigo.

En segundo lugar, quiero agradecer a Ramon Puigjaner toda la ayuda que me ha prestado. Desde que ya era un estudiante de grado, me acogió en su grupo de investigación y me proporcionó la oportunidad de conocer un poco mejor lo que supone la vida de un investigador. En cierta medida, gracias a él se produjo el primer paso de lo que es mi vida de investigador. Desde aquel momento, su ayuda y apoyo han estado presentes.

También quiero dar las gracias a las dos universidades y grupos de investigación que me acogieron y con los que compartí el trabajo de mi tesis. En primer lugar a la *Universität Wien* y a todos los miembros del grupo *Institut für Distributed and Multimedia Systems* y en especial al Prof. Günter Haring y al Prof. Helmut Hlavacs que me invitaron a su universidad en los inicios de mi tesis. También a la *Glyndŵr University* y a todos los miembros del grupo *CAIR- Center for Applied Internet Research* y en especial al Prof. Vic Grout y al Dr. Rich Picking por su hospitalidad en la estancia que sirvió para cerrar este trabajo de investigación.

No quiero olvidarme de mis compañeros de grupo de investigación, algunos que han continuado en el mundo de la investigación y otros que han decidido seguir otros caminos: Isaac Lera, Mehdi Khouja, Jaume Vicens y Pere Pau Sancho. También a mi compañero de despacho Xavi Varona y a amigos del departamento como Toni Jaume y otros muchos que forman una lista interminable y que no puedo citar uno a uno. Y aquellos cuyo trabajo ha sido aprovechado en esta tesis, como es el caso de Joan Espina. Todos ellos han aportado un pequeño grano de arena ayudándome en muchas ocasiones.

Agradecer tambin al *Departament de Matemàtiques i Informàtica* en general, y en particular a los miembros de la dirección actual y pasada, toda la ayuda y apoyo que dan a los doctorandos. También a la propia *Universitat de les Illes Balears*, por considerar un objetivo prioritario que todos sus profesores sean doctores.

Los amigos también tienen una gran importancia en un proceso tan largo como es el de una tesis. Ellos ayudan en los momento más duro haciéndote olvidar. Son varios los grupos de amigos (los de la carrera, los de la uni, los de la montaña, los de AdJ, etc.) y los que son amigos sin más calificativos: Elena, Cati, los Simarro-Fernández, Rober, Pedro, Juan, Berto, Sergio, Juan Antonio, etc.

La familia también supone un apoyo fundamental en la vida de cada uno. Yo tengo que agradecer a mis padres que desde pequeño hayan permitido que estudiara, inculcándome desde mis primeros años de colegio la importancia de estudiar, formarse, y tener una preparación. No todo el mundo puede disponer de una oportunidad como la que ellos me han brindado, ayudándome en todo momento a finalizar mis estudios y continuar con mi carrera investigadora.

Y finalmente, un agradecimiento muy especial a mi pareja. No sólo por haber aguantado los malos ratos en los que el trabajo no sale y haber aportado su pequeño granito de arena en la resolución de los problemas, sino también por haberme ayudado a disfrutar de los ratos libres, de olvidar el trabajo y disfrutar de la vida. ¡Muchas gracias Judit!

A todos ellos... *¡Buen camino!*

Abstract

The performance of the web caches has been reduced in Web 2.0 applications and, particularly, in content aggregation systems because of the high customization of the web pages and the increase in the number of content updates. These two facts result in low re-usability among the responses of the user requests. This problem can be solved by reducing the granularity of the contents stored and managed in the web cache. Thus, instead of caching the whole web pages, it is better to cache fragments of the web pages.

If the number of fragments of a web page is very high, the overhead of the assembly of fragments is also very high in comparison with the total latency time of the web page. It is important to balance the number of fragments in order to maximize the hit ratio improvements and minimize the assembling times.

The main objective of this thesis is to propose a system that adapts the content fragments of the web pages in order to reduce the user-perceived latency. This performance improvement is achieved by increasing the hit ratio and reducing the assembling times. The algorithm in charge of the adaptations is based on the values of the content characterization parameters of the web pages (structure and size) and in the user behaviour (update and request rates). The fragment designs are adapted as the content changes occur.

This thesis describes a general framework to deploy, in a content aggregation system, a solution based on the adaptation of the page fragments. The framework defines the interfaces, the changes in the web application tiers, and the new modules and interfaces to be created. The definition of the framework keeps open the way to implement the adaptive core, and it can be based on a wide range of techniques.

We have implemented the adaptive core of the system using decision trees. These decision trees have been obtained as the result of a knowledge discovery process. In this process, performance and characterization data have been mined. The data have been obtained from the emulation of a synthetic web content page model. This data mining process is done in an off-line training phase in order to reduce the resource usage. This last feature is very important to generate the lowest overhead in the implementation of the adaptive core.

The proposed framework and the particular implementation of the adaptive core have been evaluated in a set of experiments using real web content models. The evaluation is done in terms of user-perceived latency, cache hit ratios, and CPU consumption.

Resumen

El rendimiento de las caches se ha visto reducido en las aplicaciones basadas en Web 2.0 y en los sistemas de agregación de contenido. Esto se debe a los altos grados de personalización de las páginas web y a que las actualizaciones de los contenidos son más frecuentes. Estos dos aspectos provocan que el grado de reutilización entre las respuestas de las peticiones de los usuarios sea muy bajo. Este problema se puede resolver reduciendo la granularidad de los contenidos que son almacenados y gestionados de forma independiente por la cache web. En lugar de cachear las páginas web completas, es mejor cachear fragmentos de dichas páginas.

Si el número de fragmentos de una página web es muy alto, esta experimenta grandes sobrecargas por el tiempo consumido en la unión de los distintos fragmentos. Es importante equilibrar el número de fragmentos para que se obtenga el mayor porcentaje de aciertos de la cache junto al menor tiempo posible de unión de los fragmentos.

El objetivo más importante de la tesis es proponer un sistema que adapte los fragmentos de los contenidos de las páginas web para conseguir reducir la latencia observada por los usuarios. Esta mejora se basa en incrementar los aciertos de la caché y minimizar los tiempos de unión de los fragmentos. El algoritmo encargado de adaptar estos diseños de fragmentos basa sus decisiones en los valores de los parámetros de caracterización de los contenidos (estructura y tamaño) y en el comportamiento de los usuarios (velocidad de peticiones y actualizaciones). Los diseños de los fragmentos se van adaptando a medida que se producen los cambios en los contenidos.

Esta tesis describe un *framework* general para implementar una solución basada en la adaptación de los fragmentos de las páginas en sistemas de agregación de contenidos. El *framework* define las interfaces, los cambios en los niveles de la aplicación web y los nuevos módulos e interfaces a desarrollar. Los detalles de la implementación del algoritmo que se encarga de adaptar los fragmentos, el núcleo adaptativo, no son definidos por el *framework*. De esta forma, el núcleo del sistema puede ser desarrollado mediante el uso de un gran número de técnicas.

Nosotros hemos llevado a cabo una implementación particular de este núcleo mediante el uso de árboles de decisión. Estos árboles han sido obtenidos como resultado de un proceso de extracción de conocimiento. En este proceso, se ha utilizado datos sobre el rendimiento y las caracterésticas del contenido. Estos datos han sido obtenidos de una emulación donde se ha utilizado un modelo sintético de las páginas de contenido. El proceso de *data mining* se ha llevado a cabo en un fase de *training* llevada a cabo previamente a la ejecución del sistema. De esta forma, evitamos que, en tiempo de ejecución, se produzcan sobrecargas generadas por el proceso de extracción de la información. Es importante que la implementación del núcleo adaptativo no genere sobrecargas importantes sobre el sistema web.

El *framewok* y la implementación particular del núcleo, utilizando *data mining*, han sido evaluadas mediante un conjunto de experimentos donde se han utilizado modelos obtenidos de webs reales. La evaluación se basa en el estudio del tiempo de latencia observada por el usuario, los aciertos de la cache, y el consumo de CPU.

Contents

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Outline 3

Part I Background

2 Content aggregation systems 7
- 2.1 Introduction 7
- 2.2 Mashup and content aggregation architecture 8
- 2.3 Performance issues 11
  - 2.3.1 Fragment-based caching approaches 12
  - 2.3.2 Adapting a fragment-based caching solution to content aggregation systems 14
- 2.4 Web performance and data mining 16
- 2.5 Summary 18

Part II Research contributions

3 Modelling content aggregation systems and performance improvement 21
- 3.1 Introduction 21
- 3.2 Research methodology proposed to address the problem of web caching in content aggregation systems 22
- 3.3 Adaptive content fragment framework 23
- 3.4 Study of the influence of the fragment design on the performance 25
- 3.5 Analysis of the adaptive core inputs 27
- 3.6 Correlation analysis between performance and content characteristics 29

3.6.1 Models for the experiment . . . 30
3.6.2 Tools of the experiment . . . 32
3.6.3 Experimental results . . . 32
3.7 Model for content aggregation systems . . . 36
3.8 Formal definition of the problem . . . 40
3.9 Summary . . . 40

4 Definition of the core for the adaptation of content fragment designs . . . 43
4.1 Introduction . . . 43
4.2 Adaptive core definition . . . 44
4.3 Alternatives to implement the adaptive core . . . 45
4.3.1 Ontologies . . . 45
4.3.2 Genetic algorithms . . . 46
4.3.3 Data mining . . . 48
4.4 Adaptive core deployment via knowledge discovery . . . 50
4.4.1 Target data definition . . . 50
4.4.2 Preprocessing and data properties analysis . . . 55
4.4.3 Transformation of the target data: data cleaning, and instance representation . . . 57
4.4.4 Data mining algorithm selection . . . 59
4.4.5 Evaluation and coverage study . . . 60
4.5 Summary . . . 61

5 COFRADIAS: COntent FRagment ADaptation In web Aggregation Systems . . . 63
5.1 Introduction . . . 63
5.2 Architecture of COFRADIAS framework . . . 64
5.2.1 Communication among COFRADIAS modules . . . 65
5.2.2 Format of the interchanged data . . . 67
5.2.3 Content aggregation systems modifications . . . 69
5.3 Real system implementation: DRUPAL-COFRADIAS extension . . . 71
5.4 Summary . . . 75

Part III Validation

6 Experimentation design . . . 79
6.1 Introduction . . . 79
6.2 Validation of the research contributions . . . 79
6.3 Definition of the experiments . . . 81
6.3.1 Hardware architecture . . . 81
6.3.2 Algorithm for the creation of the content fragment designs . . . 81

6.3.3 Content page models ... 82
6.3.4 User behaviour ... 84
6.3.5 Changes in the values of the independent attributes ... 85
6.3.6 Experiments identification ... 86
6.4 Design of the experiments ... 86
6.5 Definition of the metrics to be monitored ... 89
6.6 Test-bed architecture ... 90
6.7 Experiment execution ... 91
6.8 Summary ... 92

7 Cache and user performance analysis ... 95
7.1 Introduction ... 95
7.2 Results of the data mining process ... 96
7.3 Performance results ... 99
7.3.1 First experiment set ... 101
7.3.2 Second experiment set ... 109
7.3.3 Third experiment set ... 117
7.3.4 Fourth experiment set ... 123
7.3.5 Fifth experiment set ... 126
7.4 Summary ... 130

8 System overhead analysis ... 139
8.1 Introduction ... 139
8.2 Overhead analysis of the fourth experiment set ... 140
8.2.1 Analysis of the workload generated by the adaptive core based on the use of decision trees ... 142
8.2.2 Analysis of the workload generated by *split* scenario ... 145
8.2.3 Comparison between *entire/∅* and the *split* scenario ... 145
8.3 Overhead analysis of the fifth experiment set ... 145
8.3.1 Analysis of the workload generated by the adaptive core based on the use of decision trees ... 148
8.3.2 Analysis of the workload generated by MACE ... 150
8.3.3 Comparison between the adaptive core based on decision trees and MACE ... 152
8.4 Summary ... 157

Part IV Conclusions

9 Conclusions and open problems ... 161
9.1 Thesis summary ... 161
9.2 Contributions ... 162
9.3 Future work and open problems ... 165

Part V Appendixes and references

A Decision trees specification . . . 169
A.1 Distributed hardware architecture . . . 169
A.1.1 Entire/∅ . . . 169
A.1.2 Entire/updateRate . . . 172
A.1.3 Entire/requestRate . . . 172
A.1.4 Entire/size . . . 173
A.1.5 Entire/childrenNumber . . . 173
A.1.6 Entire/fatherNumber . . . 174
A.1.7 Ratio/∅ . . . 174
A.1.8 Ratio/updateRate . . . 176
A.1.9 Ratio/requestRate . . . 176
A.1.10 Ratio/size . . . 177
A.1.11 Ratio/childrenNumber . . . 178
A.1.12 Ratio/fatherNumber . . . 179
A.1.13 Difference/∅ . . . 180
A.1.14 Difference/updateRate . . . 180
A.1.15 Difference/requestRate . . . 181
A.1.16 Difference/size . . . 181
A.1.17 Difference/childrenNumber . . . 182
A.1.18 Difference/fatherNumber . . . 182
A.1.19 Distance/∅ . . . 183
A.1.20 Distance/updateRate . . . 183
A.1.21 Distance/requestRate . . . 183
A.1.22 Distance/size . . . 184
A.1.23 Distance/childrenNumber . . . 184
A.1.24 Distance/fatherNumber . . . 184
A.2 Centralized hardware architecture . . . 185
A.2.1 Entire/∅ . . . 185
A.2.2 Entire/updateRate . . . 186
A.2.3 Entire/requestRate . . . 187
A.2.4 Entire/size . . . 188
A.2.5 Entire/childrenNumber . . . 188
A.2.6 Entire/fatherNumber . . . 189
A.2.7 Ratio/∅ . . . 189
A.2.8 Ratio/updateRate . . . 191
A.2.9 Ratio/requestRate . . . 192
A.2.10 Ratio/size . . . 193
A.2.11 Ratio/childrenNumber . . . 194
A.2.12 Ratio/fatherNumber . . . 194
A.2.13 Differences/∅ . . . 194
A.2.14 Differences/updateRate . . . 195
A.2.15 Differences/requestRate . . . 196

A.2.16 Differences/size 196
A.2.17 Differences/childrenNumber 197
A.2.18 Differences/fatherNumber 197
A.2.19 Distance/$\emptyset$ 197
A.2.20 Distance/updateRate 198
A.2.21 Distance/requestRate 198
A.2.22 Distance/size 199
A.2.23 Distance/childrenNumber 199
A.2.24 Distance/fatherNumber 200

B Synthetic content page model features 201

References 209

1 Introduction

This thesis demonstrates that the user-perceived latency of a content aggregation web system which uses web caching can be reduced by creating and adapting fragments of the content of the web pages. By the creation of these fragments, the metrics of the web cache are improved (hit ratio). However, overhead times, corresponding to the assemblies of the content fragments, also appear. Our main objective is to find the content fragments that balance the cache improvement with the assembling losses. Thus, we have researched the next issues: the definition of a general architecture for content aggregation systems in which the content fragments can be adapted as the content changes; the study of the most suitable inputs for the algorithm in charge of adapting the fragments; the definition of the guidelines to use data mining techniques and emulation in an off-line phase to extract the knowledge required to adapt the content fragments of the web pages; and, finally, the implementation of a real web system in which decision trees are used to represent the knowledge extracted in the training phase and to implement the classification algorithm of the adaptive core of the framework.

1.1 Motivation

Web caching has been widely used and many research works have been addressed to improve it. But the emergence of Web 2.0 has resulted in new problems and new challenges in the field of web caching. Web 2.0 applications are characterized by the high customization level and the high update rates of the content. These new features have caused that the performance of web caches has been considerably reduced due to the lack of re-utilization of the contents stored in the cache.

It is well-known that the solution of the problem of content re-usability of web caches is to reduce the granularity of the cacheable units, storing fragments of the content instead of whole web pages. This granularity reduction generates an increase of the web cache hit ratio. But the assembling process adds overhead times to the user-perceived latency. If the number of fragments reaches high values, the increases of the hit ratio will be counteracted by the losses of the assembling process.

Content aggregation systems are Web 2.0 systems in which the users are allowed to create their own web pages by the aggregation of small pieces of content gathered from content providers. If each content aggregation is stored in the cache as independent fragments, the overhead time generated by the assemblies will be very long. It is important to find a suitable fragment design which balances the benefits of having a high number of small fragments with the benefits of having few fragments of big size.

The hypothesis of this research work is that the user-perceived latency of a content aggregation system can be reduced by adapting the content fragments that are managed and stored independently by the web cache. The process of adapting the content fragments is based on classifying the aggregation relationship of a pair of content aggregation elements, by indicating if both contents are assembled in the web server, creating a unique content fragment, or they are assembled in the web cache, creating two independent content fragments. These classification decisions should be done using inputs which are easily gathered from the system. The classification algorithm, in charge of deciding the assembly point of each pair of content elements, should not add important overheads to the system. The knowledge to implement the classification algorithm should be obtained in an off-line training phase. And, finally, the extracted knowledge should be expressed with some type of structure or representation that can be used by the classification algorithm without generating a high overhead.

The main objectives of the thesis, which emerge from the hypothesis enunciated in the previous paragraph, are:

- To establish a methodology, based on phases, to address the problem of adapting the designs of the fragments to reduce the user-perceived latency in content aggregation systems.
- To find a group of parameters of the content elements, fragment elements and web pages which can be used as input of the classification process of the aggregation relationships.
- To find a method to train the classification algorithm in an off-line phase. The training should be done using performance and content characterization data extracted from the execution of a synthetic content page model in the content aggregation system. To find a suitable representation for the extracted knowledge.
- To define general guidelines for the creation of the synthetic content page model to be emulated.
- To define a general framework to adapt the fragment designs of the web pages, by determining the inputs and outputs of the adaptive core, the interfaces with the tier of the content aggregation systems, and the changes in the tiers and in the interfaces between tiers.

- To implement an example of the proposed framework as an extension of some commercial web application.
- To evaluate the validity of the proposed framework by the execution of experiments in the developed tool, using web content model extracted from real web sites.
- To compare our solution with similar proposals and with traditional cache schemes.

1.2 Outline

Hence, we have divided this document into five main parts: *Background*, *Research contributions*, *Validation*, *Conclusions* and *Appendixes and references*. The first part helps the reader to understand the related work, the differences between other works and our own research and its motivations. The second and third parts include the contributions, ideas and results of our research. Finally, the two last parts present the conclusions, future works, references and complementary information.

- **Part I: Background** In the first part, we have included a general view of content aggregation systems. We have explained their architecture and given details about web cache performance limitations. We have included two surveys, one about fragment-based caching approaches and another one about the use of data mining to improve the web performance.
- **Part II: Research contributions** The second part of this work has three chapters. Chapter 3 includes a description of a model for content aggregation systems and the study of the relationship between the performance of the web cache, for a given fragment design, and the characterization parameters of the content elements of the fragments. In the next chapter, Chapter 4, we explain the implementation of the core to adapt the fragment designs, the algorithm in charge of determining the content fragments. An exploration of the benefits and drawbacks of three possible implementations of the core is also done in the chapter. Finally, we present, in Chapter 5, the details of a general framework to adapt the content fragments in order to improve the user-perceived latency. The details about the integration in a content aggregation system are also done. The work done in the development of an example of this framework in a real web application is also presented.
- **Part III: Validation** The third part of the dissertation explains all the details about the design and execution of the experiments which evaluate the validity of our contributions. The first of the three chapters is about the design of the experiments (Chapter 6). The next one, Chapter 7 includes the analysis of the cache performance and latency results of the experiments. Finally, Chapter 8 is devoted to study the overhead of our solution by the analysis of the server workload generated by the software of our framework.

- **Part IV: Conclusions** The chapter of this part contains the conclusions of our dissertation, open problems and further research work.
- **Part V: Appendixes and references** The last part includes the appendixes and the bibliography. The appendixes are devoted to present all the decision trees that have been used in the evaluation of the research, the details of the framework algorithm which we have compared our results with, the details of the crawled real web sites and the setup to create the synthetic content page model.

Part I

Background

2 Content aggregation systems

Wit is the sudden marriage of ideas which before their union were not perceived to have any relation.
—Mark Twain—

In this chapter, the general architecture of content aggregation web applications (or mashups) is introduced and the performance limitations on this type of systems are explained, more precisely, the problems that these new applications generate in traditional web caching techniques.

The chapter also includes the summary of the most significant techniques of web caching. The significance of the techniques is based on the relation and similarity to our approach. These reviews explain that some of the solutions for the performance limitations can be solved by reducing the size of cacheable objects.

2.1 Introduction

In order to start the explanation of our research work, the background in the field of web caching and current web system should be explained. Therefore, we are going to introduce the details of web caching architectures in traditional and current web systems.

Current web systems have new performance limitation, mainly web caching techniques, because of the increase of the content update rates and the customization level of the web pages. These two facts generate a reduction in the cache effectiveness. Important research works should be done in this direction.

Research efforts in web performance and web caching have been done since web applications appeared. New paradigms of web usage and new web data flows have been created during the last years. These changes have caused that an important number of performance techniques got useless. We have addressed our research work to create a framework in which traditional performance tools are adapted to the requirements of the new web paradigm, commonly known as Web 2.0.

This chapter is devoted to explain, in a conceptual way, the architecture of content aggregation systems, and how the performance changes for each of these architectures.

The chapter is organized as follows: Section 2.2 outlines the general architecture of content aggregation systems. Section 2.3 is about the performance issues for web caching in this type of applications. Section 2.4 includes the reviews of some significant, previous and related research works, focussing on the use of data mining techniques in order to improve the performance of web caching.

2.2 Mashup and content aggregation architecture

Content Aggregation Systems (CAS) are applications in which users are able to create their own web pages by the aggregation of contents. These applications differ from other Content Management Systems (CMS) in that users do not create the content, they only set up the web pages by aggregating content from public services and content sources (Web Services, RSS,...). Therefore, the web pages retrieve content from distributed sources and assemble it in a single web page.

CAS applications do not only differ from other Web 2.0 applications in the way that the users use them. The features of CAS result in that web pages are updated more frequently and they are more customizable. Web pages have high update rates because the content is retrieved from different sources and the web page needs to be updated every time a single source changes. The web pages of these systems also have high customizable degree, because each user sets up his own pages. The pages created by different users cannot be re-used between them.

There are a lot of web applications that fit in the definition of Content Aggregation Systems: social networks, web blogs, feed aggregation tools,... We have focused our study on the next types: news sites (newspapers web sites), customized dashboards publishing platforms (PageFlakes), and personal start-pages or web portal (Yahoo! Pipes, iGoogle, Netvibes).

The well-known three-tier architecture of web environments is also used for Content Aggregation Systems. This common schema is usually extended with other tiers in order to improve the system, for example, in terms of performance. We are specifically interested in the case of extending the basic scheme by the use of a web cache tier. Therefore, we consider a four-tier architecture: data sources, web application, web cache and user presentation (Figures 2.1 and 2.2). A deeper explanation of each tier is done in the next paragraphs.

CAS systems retrieve content from data sources and assemble them to create the final user web page. The data sources are content web services which usually correspond to remote servers, geographically distributed. The web application retrieves independent content from these servers. We called content elements (CE) to these independent and indivisible contents.

HTTP requests are used to retrieve content elements. We call them content requests in order to distinguish them from other request types (user requests, template requests, etc.). The web server and the web cache are the only elements which can create HTTP requests to retrieve single content elements.

CAS systems use local databases to store some data. They need to store the information of the users of the system, the user web page setups —which contents are aggregated in which pages—, the templates of the web pages and, in some rare cases, content elements. The web application server is in charge of the process of retrieving templates from local databases and the content elements from remote content sources.

The content elements are assembled in some of the tiers of the architecture in order to create the user web pages. The CAS system is called mashup when the assemblies take place in the user clients or browsers. However, when they take place in some of the tier of the server side, the system is called portlet. For the last one, the web application server and the web cache proxy are two usual assembly points. Depending where they take place, the cache would be able to manage and to store whole and indivisible web pages, or otherwise, content elements.

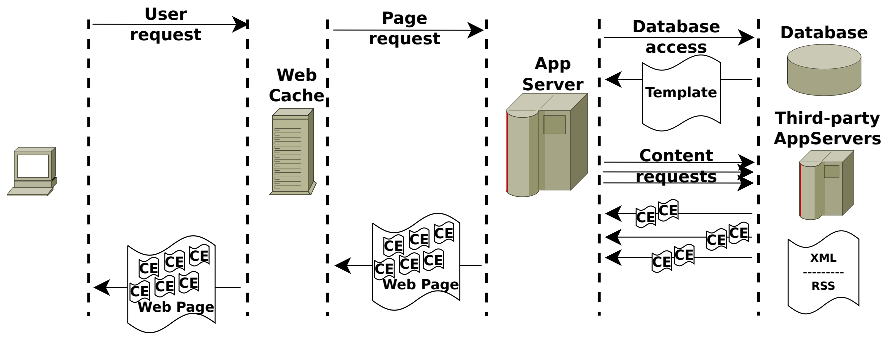


Fig. 2.1. Architecture of a content aggregation system in which the content assemblies take place in the application server.

On the one hand, when the assembling process is done in the web proxy cache, the cache is able to store content elements (CE) independently. Consequently, when the proxy creates a web page to respond a user request, it only requests to the web server the content elements that are not locally stored —because they have been invalidated, they have not been requested before or the template of the user has been modified—, the other ones are obtained from the cache local store (Figure 2.2).

On the other hand, when the assemblies take place in the web application server, if the web page is invalidated, the cache needs to request the whole web page to the server (Figure 2.1). This invalidation takes place when a single content element, or the template, changes. Architectures with different assembly points have advantages and drawbacks and they are explained in Section 2.3.

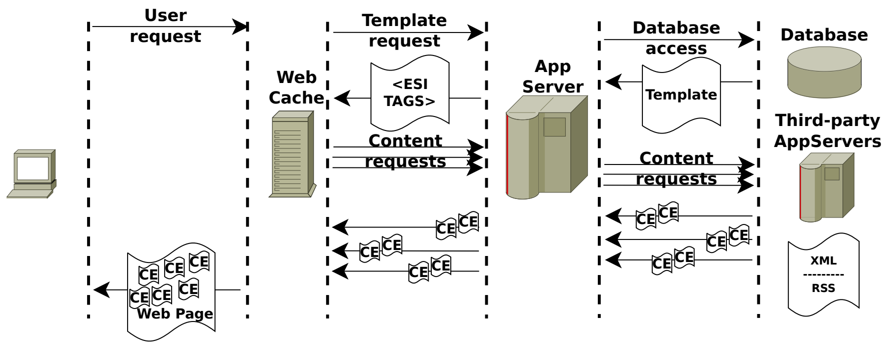


Fig. 2.2. Architecture of a content aggregation system in which the content assemblies take place in the cache server.

When the assembling process is done in the web application server, the web proxy cache is in charge of managing user requests. When a user request arrives to the proxy cache, this last checks if a temporally copy of the web page is stored in the cache: if it is stored, the answer to the user request is generated immediately; on the contrary, the proxy cache forwards a page request to the web server, and this is in charge of retrieving the template of the web page and the individual content elements in order to create the whole web page (Figure 2.1).

When the assembling process is done in the web proxy cache, this receives the user request and it forwards a template request to the web server, which retrieves the template data from the local database and sends it to the web cache. The web cache analyses the template and generates the single content requests, which are sent to the web server to be retrieved from remote sources. The web server responds with the contents which are finally assembled in the web cache (Figure 2.2).

In the case of assembling the content elements in the proxy cache, the web server needs to tell the cache which content elements are part of a web page and how they are identified and requested. ESI (Edge Side Includes) is the standard *de facto* which defines how content elements are included in a general template. ESI is based on the use of special tags that are interpreted by the cache proxy and translated into the appropriate action. There is a tag for inclusion of content elements (`<esi:include src=""/>`). The content elements are identified by an URL (Uniform Resource Locator) and the cache requests them, to the web server, using this URL. Listing 2.1 shows an example in which the ESI tags include four content elements in a HTML web page template. The ESI tags indicate the URLs which identify the four content elements and that are used to request them.

The next section (Section 2.3) has the details of the performance issues of both assembling scenarios, assembling the content elements either in the application server or in the cache proxy. We have shown that the performance of

Listing 2.1. Example of ESI tags included in a HTML file.

```
<body>
 <h1> title 1 </h1>
  <esi:include src="http://www.domain.com/cnt_elem1.html"/>
  <esi:include src="http://www.domain.com/cnt_elem2.html"/>
 <h1> title 2 </h1>
  <esi:include src="http://www.domain.com/cnt_elem3.html"/>
  <esi:include src="http://www.domain.com/cnt_elem4.html"/>
</body>
```

the web cache depends on the place in which the content elements are assembled [36]. One of the contributions of this thesis is to create an architecture with an intermediate assembling scenario, where some of the content elements are assembled in the application server and other ones in the cache, in order to achieve a higher performance than in the case of the two basic assembling scenarios.

2.3 Performance issues

The performance of a web cache is based on the re-usability of previous requests. The web cache temporally and locally stores the response of the web application server for a given page request. If this same request arrives before the page changes, the cache responds with the local copy. In this way, it is reduced the workload over the web server and the user-perceived latency.

The most common performance metrics for web cache performance are hit ratio, byte hit ratio, and latency (or response time). A hit occurs when a page request is responded with the local copy in the web cache. The hit ratio is the percentage of hits by the total number of page requests. Byte hit ratio considers the percentage between the size of all the hits and the size of all the requests. The user-perceived latency is the time between the user requests a web page and the response for the request is received. This time is usually shorter with high hit ratios, but not always as we explain in Section 2.3.2.

Web responses are re-used, in order to increase the hit ratio, in two cases: when a user requests, at least, twice the same page; when different users request the same page. The first case occurs when the page content has not been updated between two requests. Therefore, low content update rates improve hit ratios. The second case occurs when the same web page is available for different users. Therefore, customizable web systems do not help to improve the cache performance.

CAS are systems in which the web pages are very customized —each user sets up his own web page— and in which the page content updates are very usual —the page content is generated from a set of contents, so the update frequency is the sum of all the single update frequencies—. Web caches are not useful in this type of web systems, because it is very unlikely to re-used web responses.

Systems in which the contents have high update rates and high customizable level reduce the performance of web caching techniques. This is the main drawback to apply caching in current web systems and the main motivation of our research. Almost all the types of current web applications have high levels of update rates and user customization. Different solutions have been used to counteract these limitations in web systems. We have focused our background study mainly on the techniques based on the creation of fragments of web contents, but we have also taken into account others type of solutions in our background study.

2.3.1 Fragment-based caching approaches

Web caching reduces its performance in systems with high update rates and user customization levels. It is well-known that a solution for this problem is to reduce the minimum cacheable unit [65, 60, 91, 92]. The cache is able to manage fragments of the web pages instead of complete web pages. We have called them as fragment-based web caches.

The process of fragmenting the web pages to improve the performance can be applied on all the tiers of the web architecture. These tiers correspond to the data, application or presentation. In any of them, the objective is to reduce the cost of generating dynamic web pages, mainly in user-customized scenarios.

The fragment-based techniques for the presentation tiers usually work by fragmenting the HTML code in different parts which can be assembled in different places: the server, the edges of the content delivery networks (CDNs) or, even, in the user systems. We differentiate two groups: the fragment-based techniques that analyse the HTML to detect fragments, and the techniques that modify the web applications to manage fragments.

In the first set of techniques for the presentation tier, Ramaswamy *et al.* [72, 73] proposed a technique to detect fragments directly in the HTML of the web pages. It is a scheme to automatically detect and flag fragments that are cost-effective cache units in web sites serving dynamic content. The criteria to detect the fragments are the content sharing level among multiple documents and the differences in lifetime and customization. For them, any part of the document is a candidate to be a fragment.

Another approximation to deal with the high rates of changes in web systems is to work with base documents and managing the differences between documents or time shots of a document. In [69], Psounis proposed to combine a cacheable, previous snapshot of a document, called base-file, with a small difference-file, called delta, to generate the current snapshot of the web document. Instead of splitting the document in parts which could be updated independently without having influence on the others parts. This proposal always uses the same base-file, and the delta will be updated in the future.

The main part of research contributions for the presentation tier are focussed in the second set of techniques. For example, Khaing and Thein, in [53], proposed a framework in which the pages are created by aggregation of fragments. The fragments do not have to be detected, because the web application manages the pages as a set of content fragments. The fragment-based publishing system analyses the information sharing behaviour, customization characteristics, and the changes occurring to them over time.

Another framework for publishing web pages with content fragments was proposed in [16] by Challenger *et al.*. The system allows to designers to modify inclusion relationships among Web pages and fragments. The study also includes algorithms for consistency analysis, and for efficiently detection and update of web pages affected after changes in the content.

Welicki and Sanjuan proposed the creation of distributed caching framework for web-based applications [85]. In this approach, the fragments have descriptive metadata that are used for synchronization and eviction purposes, and they are not accessed directly by the users. The system has a caching front-end to interact with the storing cache server.

In [71], Rabinovich *et al.* proposed a fragment-based solution in which the fragments are assembled in the client-side. They addressed an issue of assessing whether and which pages should use fragmentation on a Web site. This is based on an analytical study where different attributes (content sizes and request rates) of the web pages are taken into account. By estimating these attributes, the web administrator should discover if the use of fragments improve the performance. Nevertheless, the fragmentation of the web pages is done by the programmers by changing the application server.

The techniques for the application tier are mainly focussed on code caching instead of content caching. These techniques apply the concept of reducing the minimum cacheable unit, but they split the web application code instead of the generated HTML code. Suresha and Haritsa, in [79], proposed a system with the integration of the fragment and code caching. The results of the different code fragments are cached separately. The code fragments, with cached results, are bypassed when a new request triggers. If the result is not cached, the code fragment is executed. They applied the technique using scripting languages. They created code block which could be executed independently.

Finally, the techniques corresponding to the data tier include, for example, Ullrich *et al.*'s contribution in [83]. They propose a pipeline based on a Model-View-Controller architecture which overcomes caching in a customization problems. They deal with the problem by using small XML documents with content elements of the web pages. They propose three points at which caching take place: after the fetching XML files with the contents; after the XSLT-transformation process; or the complete result of the user request.

Another approach with different caching points is proposed by Datta *et al.* in [21]. They proposed an approach for caching granular proxy-based dynamic content that combines the benefits of front-end and back-end caching, while suffering the drawbacks of neither. Their solution caches dynamic content fragments in the proxy cache, but the layout information will be determined, on demand, from the source site infrastructure.

Our proposed solution differs from the above ones in the way of dealing with the problem. Previous proposals are focussed in the definition of frameworks for the creation of web content by the assembly of fragments of the pages, or on splitting web pages in fragments by the analysis of the HTML code. Our proposal is focused on adapting the fragments of the web page (content fragment design), in an on-line way, as the characteristics of the content elements are changing. Since our solution is going to be applied in content aggregation system, the content elements are our initial minimum cacheable units. The fragments are created by assembling sets of these content elements. Thus, we do not need to create any algorithm to split the HTML code, it only need to decide when two content elements are assembled or not.

2.3.2 Adapting a fragment-based caching solution to content aggregation systems

We have approached the problem to get the answer for the next question: how many fragments should the content be split into in order to achieve an optimal performance, measured in terms of user-perceived latency? This amount of fragments depends on their characteristics. Content fragments which are part of several web pages with low update frequencies are good candidates to be split. But web pages, with a huge number of fragments, experiment long overhead times in assembling fragments, and connection and protocol times for the fragment requests. These problems are deeply developed at the end of this section.

Systems with fragment-based web caches have to deal with the problem of splitting, identifying and requesting the content fragments. The problem of identifying and requesting the fragments can be easily solved by the use of ESI. The problem of splitting the content of a page in several fragments depends on the type of the web application.

In the case of CAS systems, the splitting process of the web page content can be done almost directly. Each content element of the system is considered as a split fragment which will be assembled in the web cache. This fits perfectly with the CAS architecture in which the content elements are assembled in the web proxy cache.

The overhead times have influence on the latency. These overhead times are related to the connections, transmission, content processing, etc. In the case of CAS systems, there are a lot of number of content elements, and consequently, of page fragments. Thus, the overhead times have an important influence on the total latency. When a web page has a high number of fragments, as the web pages of a CAS system, the cache hit ratios are improved, but the latency gets longer due to the assembly and connection overhead times of each fragment.

These overhead times are the reason for which latency is not always improved by a better hit ratio.

$$\begin{aligned} UPL = {} & fragmentNumber* \\ & *[(hitRatio * storage_{ST}) + \\ & + (1 - hitRatio) * (server_{ST} + connectionTime) + \\ & + joinTime + \\ & + parseTime] \end{aligned} \tag{2.1}$$

The system response (UPL, user-perceived latency) is composed of several attributes. Equation (2.1) approximates the user-perceived latency [87, 55, 63]. The latency is the sum of latencies of the single fragments. Each of these fragment latencies is composed by: the time to parse the content, in order to find ESI tags; the time to join or assemble the content with other contents to create the whole page; and, finally, the time to retrieve the content — the service time of a storage access when a hit occurs, or the service time of connection, transmission and server response, when it is missed in the cache—.

The user-perceived latency depends on the hit and miss ratios, the number of fragments in a page, and the cost of assembling and parsing the templates and content fragments. Depending on the values of these attributes, it would be better to improve the hit ratio by the use of several content fragments, or minimize the overhead times (connection, assembling and parsing times) using a small number of fragments, or even, only a single fragment corresponding to the whole web page.

Basic CAS systems have only two trivial choices: (a) a huge number of fragment elements, where each content element corresponds to a fragment element (CAS with assemblies in the web cache proxy); (b) only one fragment element that is the whole web page (CAS with assemblies in the web application server). We have analysed the latency in these two schemes, and we have seen that the shortest latency is achieved sometimes by one of the schemes and other times by the other one, depending on the analysed web page [36].

In this study, we also proved that the choices are not only two —the whole page assembled in the application server or in the proxy cache—, instead of that, the problem could be addressed for each content element individually. Thus, for a given web page, the highest performance corresponds to the design in which some of the content elements are assembled in the application server and other ones in the proxy server. Therefore, an open problem emerges from this study: which contents should be assembled in the proxy cache to improve the user-perceived latency?, and moreover, which contents are improving this latency when they are assembled in the application server?

This dissertation contributes with a framework in which a core tool decides the content elements that are assembled in the application server and those in the proxy cache. The process is based on the classification of the aggregation relationship into two states: one to indicate that the assembly takes place in the application server and another one in the case of the proxy cache. The aim of this classification is to improve the performance by reducing the latency.

From other point of view, the objective of our dissertation is to define the fragments of the pages to be interchanged between the web server and the web cache. In current CAS systems, these fragments are the single content elements. We propose to pre-assemble some of them in the web server. From our knowledge, there is only another approach that addresses the problem in the same way. This is the MACE framework, presented by Hassan *et al.* in [49, 51].

The MACE framework is an approach that analyses the cost and benefits of caching data in various stages of different mashups and selectively stores data that is most effective in improving system scalability. They model mashup applications as a set of operations represented by a tree. Each web page of the system is represented by a different tree. They consider that the system can be improved by defining a caching point in the tree structure. Thus, the partial result of the execution of the web page is cached independently, and other web pages (tree structures) can take profit of the cached result. The main differences with our approach are:

- The web pages structure is represented as a graph in our approach instead of a tree as in MACE approach.
- The number of caching points in a web page is 1 in MACE approach. In our approach there is not limit in the number of different caching points.
- The algorithm to decide the caching point is implemented as a cost function in MACE approach. We use decision trees which have been mined from synthetic data.
- The inputs of the MACE algorithm are the hit ratio, the request rate, the tree-depth of the element and the cost of processing an operation, which can be expressed as a latency, utilization or size. Our inputs are quite similar, but not the same: request rate, update rate, size, number of father elements and number of child elements.

The MACE framework is the most indicated to compare the improvement of our approach with, because both approaches address the problem in the same way.

2.4 Web performance and data mining

Our proposed framework creates an intermediate scheme between the two basic ones for CAS systems. This new scheme would create pages with a smaller number of content fragments than the one in which each content element is a content fragment. Thus, the performance of the cache, in terms of hit ratios, would be lower, but the performance, in terms of latency, would be higher. The framework will pre-assemble some content elements in the application server, creating fragment elements, and these resulting fragment elements will be finally assembled in the cache proxy in order to create the whole web page, *i.e.*, the fragment elements are a set of one or more content elements pre-assembled in the application server. The details about the architecture of the framework are given in Section 3.3.

The element in charge on deciding the fragment elements is the adaptive core. The details about how this adaptive core classifies the fragment elements are given in Chapter 4. Several approaches have been considered to implement the adaptive core, but we have finally adapted a solution based on knowledge discovery and data mining. We have mined data about the performance in order to predict it in the future. This prediction helps us to decide the fragment elements design.

Data mining techniques have been widely applied in the field of web performance engineering, and more precisely, in the field of web caching. Some of them are based on the mining of historical data of user behaviour. This is usually called as Web Usage Mining and there are a lot of techniques based on it [68].

Bonchi *et al.* proposed to extend the LRU (Least Recently Used) caching algorithm by using decision trees and association rules. These structures are created by a process of data mining over server logs [8]. In [89, 90], Yang *et al.* also used data mining to extend the eviction algorithm of the cache, but, in this case, they extended the GDSF (Greedy-Dual-Size-Frequency) one. The data mining process is also done with data from the request logs and a predictive model is obtained. The predictive model is used to calculate the probability of requesting the same web page twice.

Instead of using traditional web mining algorithms, other authors proposed their own algorithms to extract knowledge from the historical user data. For example, Fue *et al.* defined the *access-orientation* between two URLs, which indicates the frequency of visiting one URL after the other [28]. The prediction algorithm uses this metric to decide the cached web pages with less probability to be accessed in the future, and consequently, the most suitable to be evicted from the cache.

Huang and Hsu [52] proposed their own mining algorithm. In this case, the result of the algorithm is a rule table. These rules will be applied to future requests in order to assist prefetching and caching to decide which web pages are evicted and which ones are prefetched. The algorithms generate frequent sequences of paths. The frequent sequences are used to create the rules to indicate the most probable requests after a given one.

The inventors of patent [74], Ramos *et al.*, provided a method for populating a web cache, either in real-time or batch mode, with data mining techniques. These techniques evaluate workloads, discover query patterns of consumers, and anticipate the needed data. These decisions are made dynamically in anticipation to the requests of the consumers.

Other authors mix the data from the user access logs and from the website structure repository. Makkar *et al.* use this to overcome the limitation of path completion. They apply Petri Nets to extract web site structure in order to complete the paths, improve the prediction and decrease the web latency [61].

Apart from data mining techniques to create rules or decision trees, they are used to create clusters or groups of web pages in the field of web caching. Pallis *et al.* published in [67] an algorithm based on graph clustering, which is used to identify clusters of correlated web pages. Kumar *et al.* [56] deal with the caching problem by discovering patterns in user object requests. They exploit the patterns of users by making caching decisions for a specific time interval based on the history of observed requests for the same interval. Their approach also includes a dynamic portion in order to handle deviations from normal usage patterns.

Finally, some approaches mix several techniques. For example, Sulaiman *et al.* tackle the problem of caching as a classification problem. They employed Classification and Regression Trees (CART), Multivariate Adaptive Regression Splines (MARS), Random Forest (RF) and TreeNet (TN) for classification on Web caching [78].

All the previous approaches try to improve the eviction algorithms or anticipate the introduction of web pages in the web cache by mining historical user patterns. We also apply data mining to the field of web caching, but our approach differs from these techniques in the aim and in the data used in the data mining process. Our aim is to create fragment elements designs (the parts of the web page). The knowledge to achieve this goal is obtained by mining performance data, instead of user patterns.

2.5 Summary

This chapter included the background of this Ph.D. dissertation. This background covers the content aggregation systems, and the details of their architectures. These schemes are differentiated by the place in which the content elements are assembled: either in the application server or in the proxy server. Both schemes have their own benefits and drawbacks from a performance point of view. If the cache is able to manage the content elements independently, the hit ratio of the cache is improved, but the overhead time is longer. On the contrary, the overhead time is reduced when the content elements are assembled in the application server, but the hit ratio of the cache decreases significantly.

This dissertation includes the proposal of a framework to counteract the limitations of cache performance in environments with high update rates and web pages with a high level of user customization. The framework bases the contributions on adapting dynamically the fragment elements design using knowledge extracted, by the use of data mining, from synthetic data.

This chapter also completed the background of our work with some related research works. We have reviewed some significant studies in two different fields: web caching and data mining; and fragment-based web caches.

Part II

Research contributions

3

Modelling content aggregation systems and performance improvement

Think globally, act locally.
—Patrick Geddes—

In this chapter, we explain the most important issues of our proposed solution for the performance degradation in content aggregation systems. We also study the influence of content characteristics on the performance. This study is addressed to define a set of parameters or attributes which will be used as inputs of the adaptive core of our proposed architecture. As a result of this preliminary study, we will obtain a group of possible parameters. Further experiments will be done to validate their final suitability.

The chapter also includes details about how to model the web pages and the content aggregations in a CAS system. A formal definition of the model is given. Finally, a formal description of the main problem is also explained.

We propose a new architecture for content aggregations systems in which the web pages are split into fragments. These fragments are adapted as changes in the contents occur. The design of the fragments is addressed to improve the performance, more precisely, the user-perceived latency.

3.1 Introduction

The latency of a content aggregation system (CAS) can be improved by changing the content fragment design, *i.e.*, the definition of which content elements are assembled in the web application, and which ones are assembled in the proxy cache. This is the main hypothesis which our dissertation is based on. A framework to solve these performance issues is presented in the first section of this chapter (Section 3.3). We have created this framework to counteract performance losses in content aggregation web applications.

In Section 3.2, we explain the details of the phases to conduct the different problems addressed in our research. The specification of these phases is part of the definition of a methodology to to cope with problems related to content aggregation systems and web caching. The definition of the methodology is also a contribution of our research work. Further research works may be solved by using the same methodology and following the same phases. Therefore, our contributions are not only related to the techniques, solutions, etc. to solve our problem, they are also related to the definition of a methodology to address this type of problems.

One important issue, to validate our proposal, is to prove that different content fragment designs have influence on the performance. In Section 3.4, we explain the results of an experiment which proves that changes in the content fragmentation design affect the latency.

Once the initial hypothesis is proved, we shall identify which parameters or attributes can be used to determine ideal fragment designs. From our knowledge, there is not any previous study about which parameters are able to predict the fragmentation design. Therefore, we have selected usual parameters from cache techniques and methods. We have studied if these parameters show any correlation between their values and the difference of the latencies when the content elements of a web page are assembled in the web application and when they are assembled in the proxy cache. The experiments and the results are explained in Section 3.5 and 3.6.

Finally, a formal description of the main problem of the dissertation (Section 3.8) and a formal modelling annotation for the contents of a content aggregation system (Section 3.7) are presented. We propose to use a model based on an extended DAG (Directed Acyclic Graph) to represent characterization parameters of content elements and to represent information about the assembly point of each aggregation. The adaptive core determines this last information using the structure and the characterization data from the DAG.

3.2 Research methodology proposed to address the problem of web caching in content aggregation systems

We have addressed the research of this dissertation in successive phases. These phases are shown in Figure 3.1 and their descriptions are:

(i) *Definition of the problem.* The problem of our research has been already stated. Our goal is to reduce the user-observed latency in content aggregation systems. The use of web caching in this type of system has experimented an important degradation. This is due to the increase in the customization of the web pages and the higher update rates of the contents.

(ii) *Proposed solution.* Our approach is based on the concept that the latency of the system can be reduced by changing the content fragments. These adapted fragments could be managed and stored independently in the web cache. The assembly points of each pair of aggregated element define these content fragments. This feature should be validated before the next phases.

(iii) *Design of the framework for the approach.* The definition of the framework should describe the changes in the tiers and the interfaces of traditional content aggregation systems and the new interfaces. These new interfaces will integrate the tiers of the CAS system and the adaptive core.

(iv) *Exploration of the parameters available as input of the approach.* The available parameters of web pages (content elements, user behaviour, system metrics, etc.) should be considered as possible inputs of the algorithm in charge of defining the fragments of the web pages. We should validate if it exists any relationship among the values of these parameters and the assembly point in which the system shows a shorter user-perceived latency. We would select, as inputs of the core of the framework, the parameters that show correlation between these two metrics.

(v) *Selection of the technique to implement the core of the framework.* We need to decide which technique is used to implement the algorithm in charge of adapting the fragment designs. This algorithm should accomplish some requirements: it should generate a low overhead; it should use inputs that can be monitored easily from the system; and, finally, it should make the changes in the fragment design as soon as possible a change occurs in the content elements, web pages or user behaviour. The inputs of the core are the parameters selected in the previous phase, and the outputs are the fragment designs.

(vi) *Evaluation of the benefits of the approach.* The benefits of the techniques used to implement the core should be evaluated with the execution of a set of experiments. This experiment should be designed to validate the use of the core, and an enough number of replicas of the experiments should be executed in order to obtain reliable values. The evaluation of the results of the experiments validate, or not, the contributions of the research.

3.3 Adaptive content fragment framework

One important contribution of this dissertation is the design of a new framework to improve the performance in content aggregation systems. This performance improvement is based on a reduction of latencies. Traditional content aggregation systems are able to assemble the contents either in the web server application or in the web proxy cache. Our previous studies have shown that the performance can be improved by pre-assembling some of the content elements in the web application server and assembling the complete page in the web proxy cache [36, 34].

We call content fragment (CF) to the fragment of a web page that has been pre-assembled in the application server. Thus, a content fragment is a set of one or more content elements. We call fragment request to the HTTP request that the proxy cache sends to the application server in order to retrieve the content of these fragments. Content fragments are completely transparent to the user, and they are internal re-organizations of the content assemblies which help to improve the performance.

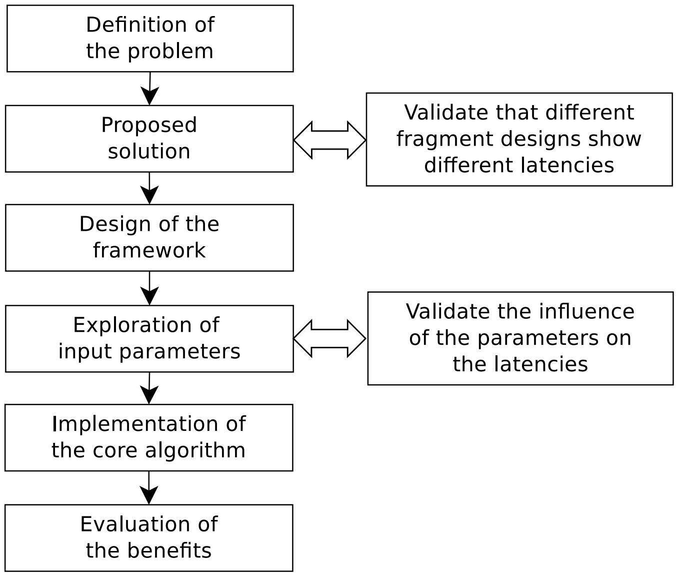

Definition of the problem
Proposed solution
Validate that different fragment designs show different latencies
Design of the framework
Exploration of input parameters
Validate the influence of the parameters on the latencies
Implementation of the core algorithm
Evaluation of the benefits

Fig. 3.1. Research methodology of the dissertation.

The proposed framework is able to adapt the fragment elements in order to improve the performance. This fragment adaptation is achieved by changing the content elements which are pre-assembled in the application server, *i.e.*, the content elements included in a content fragment. We call content fragment design to each of the possible distributions of the content elements in content fragments. The performance improvement is based on the concept of creating content fragment designs that balance the hit ratio improvement of the cache and the losses generated by the fragment overhead times. The system has to deal with two classification criteria: small content fragments improve cache hit ratio; big content fragments reduce overhead times.

A general view of the architecture of the adaptive content fragment framework is shown in Figure 3.2. The main differences with the two previous schemes (Figures 2.1 and 2.2) are that the content elements (CE) are pre-assembled in the application server creating content fragments (CF). Finally, the content fragments are assembled in the proxy cache.

The framework provides a new element in the architecture: an adaptive core. The adaptive core uses data about the characteristics of the content elements in order to decide the content elements included in a content fragment. It means that the system adapts the fragments in which a web page is divided. The result is delivered to the application server, which is able to create the content fragments. The details about the inputs of the adaptive system and the models used to interchange data with the content aggregation system are given in Sections 3.5 and 3.7.

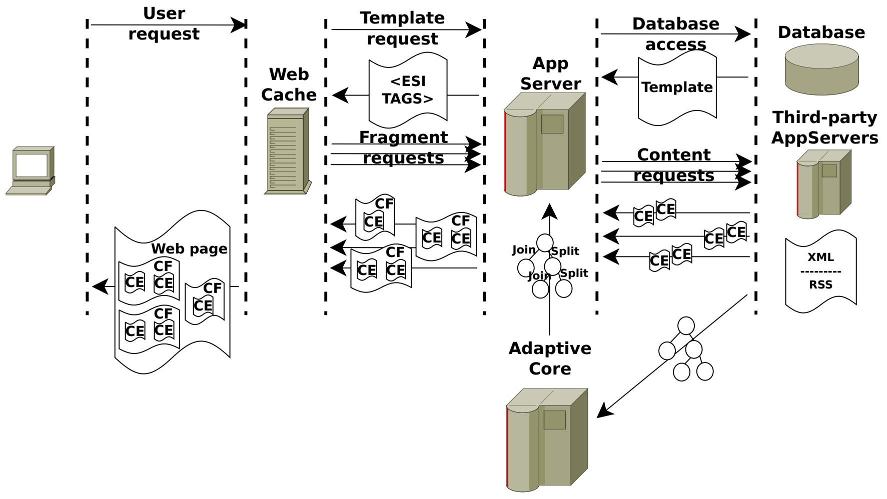


Fig. 3.2. Architecture for adaptive content fragment framework.

There is an important number of open problems in order to create the proposed system. The performance research involves a deeper analysis of the performance of content aggregation architecture. Thus, we need experimental results which prove that changes on the content fragments have influence on the performance perceived by the users (Section 3.4).

We use the results of this previous work to study which parameters or characteristics of the content elements, fragment elements and web pages have influence on the performance (Section 3.6). We use these parameters, and the relation between them, to define the inputs of the adaptive system.

We analyse different techniques for the implementation of the adaptive core (Chapter 4). The overhead generated by the core should be the lowest possible, so that, we analyse the overhead and the hardware resources consumption for each alternative, together with the improvement achieved for each of them.

Once the inputs and the implementation of the adaptive core are defined, the changes in the system architecture are also analysed. Thus, we define the interface of the CAS system and the guidelines to identify and manage the adaptable content fragments (Chapter 5).

3.4 Study of the influence of the fragment design on the performance

The work developed in this dissertation is based on the concept that different content fragment designs experiment different latency times, and the best solution is not any of the two basic schemes for content aggregations systems: assembling all the content in the applications server or assembling all the contents in the proxy cache.

We have developed a benchmarking experiment to show that, for some web pages, the latency is shorter when the content elements are assembled in the proxy cache, and, for other pages, it is shorter when they are assembled in the application server. We have used a content aggregation system with a real content model. The CAS system is a specific development for the web site of the *Fundació Universitat Empresa de les Illes Balears-FUEIB*. These results have been published and explained in [36].

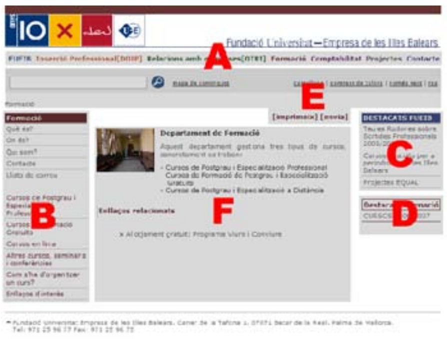


Fig. 3.3. Content designs for the benchmarked web site.

The web pages are created using a template and six content elements. These content elements correspond to several remote contents, a navigation menu, and several banners. The structure of the web page and the content elements is in Figure 3.3. The content elements are labelled with capital letters. Some of them (A, E and C) are shared between all the pages: the menus and the banners. The total number of web pages is 60. Some other content elements are shared with a subset of web pages (B and D): highlighted contents that are retrieved from remote servers. There is a total number of 6 subsets with 10 pages each. Finally, one of the content elements is exclusive for each page (F): content that is aggregated from other systems.

We have performed two experiments. In the first one, all the content elements have been assembled in the application server. In the other one, they have been assembled in the proxy cache. In this way, we are able to compare the latency of the web pages among the two cases. If we find pages with shorter latencies in both assembly points —some pages with shorter times when the contents are assembled in the proxy cache, and other ones with shorter times and joints in the application server—, we will show that the best assembly point is not always located at the same architecture element (the cache or the web server).

Both tests have been executed 10 times. 100 concurrent users have been emulated in each execution, and each user has done 100 requests. The requests have been uniformly distributed over the 60 web pages. The users have been emulated with *Apache JMeter* [46]. *JMeter* allows us to gather information about the latency of each web page. We have summarized these data in three groups:

(a) Pages in which the shortest latency is obtained when the content elements are assembled in the web cache proxy;
(b) Pages in which the shortest latency is observed when the content elements are assembled in the application server;
(c) Pages in which the difference between the latency of both assembly points is less than 50ms (1% of the maximum latency), which can be considered almost negligible for our purposes.

In Table 3.1, we present the number of pages in each group and the average of latencies for all the pages in a group in both benchmark tests.

Table 3.1. Summarized latencies for the benchmark to study the assembly points with the highest performance.

Page group	Number of pages	Mean Latency (ms) A.P.= cache proxy	Mean Latency (ms) A.P.= web server
a	39	2576	4673
b	21	3850	2281
c	10	2740	2734

The results of this benchmarking test validate our initial hypothesis: the ideal content fragment design, from a performance point of view, is different for each web page. Moreover, this optimal design does not have to correspond to one of the basic assembling schemes because the shortest latencies are obtained, in some web pages, by assembling the content elements in the cache, and, in other web pages, by assembling them in the web server. The parameters or attributes which these two cases are depending on are analysed in the next two sections.

3.5 Analysis of the adaptive core inputs

The content fragment design is adapted while web pages and content elements are changing. Therefore, the adaptive core needs to be executed every time a content of a web page changes. These changes are very usual, especially in content aggregation systems and Web 2.0 applications. Thus, the workload generated by the algorithm execution should be very low. But another important requirement should be met, the inputs of the algorithm should be easily gathered and they should be available immediately after a change occurs.

The first option for the inputs of the adaptive core is to use the feedback from the performance metrics. This feedback indicates if the current fragment design experiences an ideal performance or not. The problem is that performance cannot be calculated in a short period of time, because a single sample of a performance metric is not valid. A great number of samples should be gathered in order to get reliable averages. Furthermore, the system has high update rates, so it is very probable that content will be changed again before mean reliable values can be calculated. Therefore, the adaptive core does not use performance metrics as an input value.

The algorithm of the adaptive core needs to decide if a content element should be previously assembled in the web application tier. This decision results in a classification of the aggregation relationships in two possible states. This state indicates the assembly point of an aggregation. The details of the data model to represent this information are given in Section 3.7.

If the number of content aggregations is represented by n, the total number of fragment designs is calculated by the formula of the variation, $V'(2, n) = 2^n$. The complexity of such problem is $O(2^n)$ and the data complexity is $O(n)$. The solution space complexity needs to be simplified because it is too high to be executed in an on-line process. In order to create a faster and simpler algorithm, only two content elements directly related are going to be taken into account to classify a given aggregation. In this way, the complexity of the problem has been reduced to $O(n)$, and the amount of data used in each classification is limited to two sets of node parameters, data complexity is $O(2)$.

We need to find content metrics that could be used to predict the best content fragment design, from a performance point of view. This problem is not trivial and, from our knowledge, previous research work has not been done. The number and the type of parameters of a content element are very high. The analysis and the selection of a set of content parameters to be used to predict the content fragment design are part of our research contributions.

Metrics and parameters, such as size, hit ratios, request rates, etc., have often been used in well-known cache techniques. Their suitability has been proved in many research studies and publications [2, 17]. Although these techniques are related to the field of the web caches, their basis of applicability are not common to our research problem. In traditional web cache techniques, these parameters are used in cache eviction techniques. We need to study if there is a relationship between the content metrics and the suitability of assembling the content elements on the server or on the cache, *i.e.* the content fragment design.

The complete lack of studies has caused that any set of metrics related to caching techniques could be very useful for us. Therefore, we have studied traditional web cache inputs (size, latency, request rate, update rate, number of children, number of fathers) to predict the ideal content fragment design, from a performance point of view. In the next section, we explain the details about the experiments that have been used to select and to reject some of the parameters.

3.6 Correlation analysis between performance and content characteristics

In this section, we explain the experiments to prove if a set of seven content characterization parameters —content size, content hit ratio, content request rate, number of pages aggregated by a content element, number of pages where a content element is aggregated, content service time, and depth level of the aggregation— can be used to decide a content fragment design for a given web page, in order to improve the performance or user-perceived latencies.

These experiments are only addressed to study the suitability of the parameters to predict the ideal content fragment design. They are not addressed to create an algorithm, a formulation or a technique to determine the content fragmentation design. The issues and details of these other problems are explained in Chapter 4.

We have based the study on a correlation test. The correlation is studied between the content characterization parameters and the latency. Correlation tests ($\rho_{X,Y}$) involve two random variables (X and Y) and they prove if the samples of both variables follow, or not, any trend.

Thus, we have created a synthetic content model —content elements and aggregation relationships created randomly—, and we have measured, in a real system, the latency when two content elements are assembled in the application server and when they are assembled in the proxy cache. We have three different variables in our system: the characterization parameters of both content elements (the father content element[1], and the child content element[2]) and the improvement among the latencies of both assembly points. In order to simplify our study, we should reduce these data to only two variables.

In our particular case, the first random variables of the correlation test (X) are the characterization parameters. The aggregation is called i and each single characterization parameters of the father element of the aggregation is $CE_{i,cp,father}$, where *cp* indicates the name of this single characterization parameter. The notation is the same for the child element $CE_{i,cp,child}$. Therefore, we are going to perform one correlation test for each possible value of *cp* and for each of the two involved content elements $\{father, child\}$. The samples for the first variable are $x_i = CE_{i,cp,\{father,child\}}$. The second set of samples is the improvement obtained by comparing the latency results in both assembly points. This type of improvement is usually expressed as the speed-up between both cases ($y_i = \frac{UPL_{assemblingPoint(i,appServer)}}{UPL_{assemblingPoint(i,proxyCache)}}$).

We are going to make a correlation analysis for each single characterization parameter we want to study. If the analysis shows correlation, the given characterization parameter would be useful for predict the content fragment design. We study the next characterization parameters, *cp*: content size, ser-

[1] The father content element is which includes the content element of the other element.

[2] The child content element is which is included by the other one.

vice time, content request rate, content update rate, number of children of a content element, number of fathers of a content element, depth level of the aggregation —number of aggregations that are between the template element and the current aggregation—. The last one is a parameter associated with the aggregation relationship; the other ones are related to the content elements. Therefore, we have two values for all of them —the values corresponding to the father element and to the child element— and only one for the last one (depth level).

We have also considered to study the correlation using the ratios between the father characterization value and the child one. Sometimes, it is more important the relation between these values, than the values themselves. Therefore, the two initial correlation studies have been extended to a third one in which the random variable is $x_i = \frac{CE_{i,cp,father}}{CE_{i,cp,child}}$.

In order to study the correlation, we need to have samples of the two random variables. The characterization metrics are created synthetically and they determine the content page model (content elements, aggregations and web pages) of the experiment. Once we have the content page model, we will obtain the performance results. We are going to use two different tools to obtain these results: a simulator and an emulator. The simulator help us to scale the experiment to high workloads. The emulator give us real performance data, since it is a real deployment of a content aggregation system with real applications, hardware, data, etc. The subsections below include details about the creation of the page model, the experimental tools and the results analysis.

3.6.1 Models for the experiment

In order to obtain and measure performance metrics, we need to parametrize and deploy tools. This parametrization defines the user behaviour and contents of the system (web pages and content elements).

The particular values of the parameters for the user behaviour and the content characterization models are randomly generated using statistical distributions. We have used previous research results to select the most suitable statistical distribution for each parameter. There is a huge number of studies about content and user characterization for web systems. We focus on studies about Web 2.0 systems because they are the most similar to content aggregation systems. We analyse these model parameter results by grouping them in three sets (Table 3.2): parameters related to the user behaviour, to the server workload, and to the content.

The parameters for the user behaviour model are the user arrival rate, number of requests in a session, think-time and the popularity of the content. Duarte *et al.* [23] conclude the popularity of the contents is modelled by a power law with $\alpha = 0.83$ and $R^2 = 0.99$ for reads and $\alpha = 0.54$ and $R^2 = 0.99$ for updates. These authors also conclude the user arrival rates are defined by a Weibull with parameters $\alpha = 0.000469$ and $\beta = 064892$.

We only need the web service time and cache service time to model the system service times. These values are only used in the simulator, because the test-bed tool uses a real cache and a real web server. Traditionally, the exponential distribution has been considered the best one to define service times in general computer systems and, in particularly, for web systems [14].

The last group of parameters are related to content. We need to model the content size, the number of aggregated contents in another one (number of children), and the number of contents in which another one is aggregated (number of fathers). The size of content elements is modelled by a lognormal distribution [76]. Brodie *et al.* specify, in [11], the content fragment sizes of three different web pages. Two of them are newspaper web sites and the other one is a web blog. They considered a mean fragment size of 1 KB and a variance of 128 bytes. The structure of the aggregation relationship is defined by the popularity of the content elements. [15], [31] and [45] model the object popularity with a Zipf-like distribution with $\beta = 0.56$ and $R^2 = 0.97$. The number of aggregations of a content element is modelled by a Pareto distribution.

Table 3.2 summarizes the parameters we need to define to create the models and the distributions used for each of them. The parameters of each distribution and its initialization values are also presented. These values have been determined using the referenced bibliography.

Table 3.2. Statistical distributions for the characterization of a web fragment cache.

Parameter	Group	Distribution	Parameters
Popularity	User	Power Law	$\alpha = 0.83$ $R^2 = 3$
Arrival rate (ms^{-1})	User	Weibull	$\alpha = 0.000469$ $\beta = 0.64892$
Update frequency (ms^{-1})	User	Power Law	$\alpha = 0.54$ $R^2 = 3$
Thinking time (ms)	User	Pareto	$\alpha = 1000$ $m = 5000$
Web service time (ms)	System	Exponential	$\mu = 100$
Cache service time (ms)	System	Exponential	$\mu = \frac{frag\_size}{200000}$
Content size (bytes)	Structure	Lognormal	$\mu = 1024$ $\sigma = 128$
Number of aggregations	Structure	Pareto	$\alpha = 1$ $m = 20$
Object popularity	Structure	Zipf-like	$\beta = 0.56$ $R^{=}0.97$

3.6.2 Tools of the experiment

We have used two different tools to measure the performance: a test-bed and a simulator. On the one hand, the test-bed gives us more realistic results, but the capacity to scale the system and the number of users is smaller. On the other hand, we are able to scale the system as much as we need by using a simulator.

The simulator and the test-bed periodically change the assembly point of one aggregation relationship in a web page. Thus, we are able to measure the latency in two cases: when the selected aggregation is assembled in the web cache, and when it is assembled in the application server. Only one of the assembly points of the pages is changed, the other ones remain identical. Thus, we compare the performance of both cases with the characterization parameters of the two elements related by the aggregation.

The simulation tool has been developed as an event-driven simulator. The simulator generates clients which represent page requests. The simulation scheme has two main entities: the proxy cache and the web applications. This simulation model does not take into account the communication layers. We use the simulator to validate the results from a real application (the test-bed), but with high loads. The cost of deploying a simulator, to include communication layers, is very high. We also considered the use of a network simulator as NS2 or OPNET. These simulators are oriented to network scenarios, but they are not able to simulate web cache systems based on content elements and aggregation of contents. There are also specific web cache simulators [13, 62, 29, 22], but they have the same limitations.

The test-bed is completely based on the architecture presented in Figure 3.2. A Java application is developed to emulate the behaviour of the users. The application creates requests randomly, using the suitable distributions. The web proxy cache is deployed with *Oracle Application Server Cache 10g* [20]. This proxy cache is compatible with ESI tags [19]. The application server is a PHP application [58], which manages web pages and aggregation of content elements. The web pages and the aggregations are created previously to the emulator execution. Finally, the adaptive core is a simple program that is able to change the assembly point for individual aggregations. Thus, we compare the latency times among web pages in which only one assembly point changes.

3.6.3 Experimental results

We use two correlation tests to study the correlation between our two variables: Pearson correlation coefficient and Spearman's rank correlation coefficient. Pearson coefficient is the best known and it considers that there is a linear relationship between the two studied samples. Spearman coefficient is used without making any other assumptions about the particular nature of the relationship between the variables. Both coefficients give us an idea of how independent the variables are ($\rho_{X,Y} = 0$) or not ($\rho_{X,Y} = 1$ or $\rho_{X,Y} = -1$).

The experiments have been repeated 10 times. Each simulator execution covers 1.000.000 user requests. Each emulator experiment covers 100.000 requests. The total number of web pages is 10.000 for the simulation experiment and 1.000 for the emulation. The number of content elements is 300.000 and 30.000 respectively. The values for the performance functions have been calculated as the mean value for all the experiment replicas.

Before the emulation or the test-bed execution, we already have the data samples corresponding to the characterization parameters: those for the father content element of the aggregation relationship, $CE_{i,cp,father}$, and those for the child element, $CE_{i,cp,child}$. Once we have run the experiments in the emulator and the test-bed, we have the data samples of the performance values: the latency when a given aggregation is assembled in the application server ($UPL_{assemblingPoint(i,appServer)}$) and the latency when it is done in the proxy cache ($UPL_{assemblingPoint(i,proxyCache)}$). Finally, we calculate each y_i and we related them to their corresponding x_i.

The results of both correlation tests (Pearson and Spearman) are presented in Tables 3.3 and 3.4. There are many studies which give some guidelines to determine the minimum correlation value to consider significant correlation. These values depend on many aspects, for example, on the size of the sample. We have considered the correlation is significant for values above 0.3. Thus, we have labelled with an asterisk the test results in which there is correlation, from our point of view.

We are not interested in performing a deep statistical analysis of the results. We only need some clue about which parameters can be used as inputs of the adaptive core. Therefore, we use these preliminary results only to reject parameters. We are going to design further experiments to prove the suitability of the selected ones. These future experiments are based on the use of the real adaptive core and we will obtain more accurate results, for our specific solution. These experiments are explained in Part III of this dissertation.

In general terms, the simulation and the test-bed results are quite similar. The results between Pearson and Spearman tests are similar as well. These two facts help us to validate the results of this preliminary study.

The results show that there are parameters in which the correlation is very clear, *e.g.*, the request rate. The request rate parameters show correlation in the case of the father, the child and the relation between both of them.

For these three cases, father, child and ratio, correlation is also present for the size and for request rate, but it is not so clear. The results from the simulator do not show correlation (values below 0.3) in some of the cases of these two parameters. But the correlation is increased to values above 0.3 when the results from the test-bed are analysed, except in the case of Spearman correlation of the related sizes. The suitability of these parameters as inputs of our system needs to be validated with further experiments. Therefore, we are not able to reject their use, but we are not either sure about their use.

Table 3.3. Correlation analysis for the results obtained in the simulator. The samples of the Y variable are calculated as $y_i = \frac{UPL_{assembPoint(i,appServer)}}{UPL_{assembPoint(i,proxyCache)}}$

X	Pearson $\rho_{X,Y}$	Spearman $\rho_{X,Y}$
$x_i = CE_{i,size,father}$	* -0.3729	* -0.3926
$x_i = CE_{i,size,child}$	* -0.3162	* -0.3013
$x_i = \frac{CE_{i,size,father}}{CE_{i,size,child}}$	-0.2451	-0.2231
$x_i = CE_{i,updateRate,father}$	* 0.9034	* 0.8635
$x_i = CE_{i,updateRate,child}$	* 0.9102	* 0.9012
$x_i = \frac{CE_{i,updateRate,father}}{CE_{i,updateRate,child}}$	* 0.5837	* 0.5164
$x_i = CE_{i,requestRate,father}$	-0.2924	-0.2913
$x_i = CE_{i,requestRate,child}$	* -0.3027	* -0.3189
$x_i = \frac{CE_{i,requestRate,father}}{CE_{i,requestRate,child}}$	0.2468	0.2023
$x_i = CE_{i,fathersNumber,father}$	0.0045	0.0021
$x_i = CE_{i,fathersNumber,child}$	* 0.6499	* 0.6010
$x_i = \frac{CE_{i,fathersNumber,father}}{CE_{i,fathersNumber,child}}$	0.0632	0.0412
$x_i = CE_{i,childNumber,father}$	* -0.8813	* -0.9023
$x_i = CE_{i,childNumber,child}$	0.1009	0.1128
$x_i = \frac{CE_{i,childNumber,father}}{CE_{i,childNumber,child}}$	0.0095	0.0072
$x_i = CE_{i,serviceTime,father}$	-0.1034	-0.0914
$x_i = CE_{i,serviceTime,child}$	-0.1231	-0.1329
$x_i = \frac{CE_{i,serviceTime,father}}{CE_{i,serviceTime,child}}$	0.0234	0.0239
$x_i = CE_{i,depthLevel,-}$	0.0042	0.0013

Experiment results presented in Chapter 7 help us to make a final decision about that.

The number of children (aggregated contents) of the father content element, and the number of the fathers of the child content element (number of content elements where it is aggregated) also show clear correlation. These two parameters should be taken into account. The other parameters (depth level and service time) and cases (number of children of the child content element, number of fathers of the father element and their relation with other parameters) have been rejected because they do not show significant correlation.

We presented similar results in [41], but the correlation analysis was focused in a different way. The user behaviour model, the synthetically created web pages, the emulation model, the test-bed system, the executions and the sample of the results for performance, and the samples for the characterization parameters were the same that we have used for the analysis presented in this section. In contrast, the variables of the correlation test have been defined in a different way.

In the cited work, we studied three performance metrics, the hit ratio, the byte hit ratio and the latency. On the contrary, we have finally consid-

Table 3.4. Correlation analysis for the results obtained in the test-bed executions. The samples of the Y variable are calculated as $y_i = \frac{UPL_{assembPoint(i,appServer)}}{UPL_{assembPoint(i,proxyCache)}}$

X	Pearson $\rho_{X,Y}$	Spearman $\rho_{X,Y}$
$x_i = CE_{i,size,father}$	* -0.3925	* -0.3836
$x_i = CE_{i,size,child}$	* -0.3472	* -0.3211
$x_i = \frac{CE_{i,size,father}}{CE_{i,size,child}}$	* -0.3129	-0.2925
$x_i = CE_{i,updateRate,father}$	* 0.9483	* 0.9286
$x_i = CE_{i,updateRate,child}$	* 0.9262	* 0.9319
$x_i = \frac{CE_{i,updateRate,father}}{CE_{i,updateRate,child}}$	* 0.7027	* 0.7089
$x_i = CE_{i,requestRate,father}$	* -0.3638	* -0.3055
$x_i = CE_{i,requestRate,child}$	* -0.4240	* -0.3971
$x_i = \frac{CE_{i,requestRate,father}}{CE_{i,requestRate,child}}$	* 0.3425	* 0.3288
$x_i = CE_{i,fathersNumber,father}$	0.0559	0.0696
$x_i = CE_{i,fathersNumber,child}$	* 0.7753	* 0.7969
$x_i = \frac{CE_{i,fathersNumber,father}}{CE_{i,fathersNumber,child}}$	0.0632	0.0539
$x_i = CE_{i,childNumber,father}$	* -0.9516	* -0.9601
$x_i = CE_{i,childNumber,child}$	0.1593	0.1329
$x_i = \frac{CE_{i,childNumber,father}}{CE_{i,childNumber,child}}$	0.0139	0.0099
$x_i = CE_{i,serviceTime,father}$	-0.1530	-0.1499
$x_i = CE_{i,serviceTime,child}$	-0.1487	-0.1463
$x_i = \frac{CE_{i,serviceTime,father}}{CE_{i,serviceTime,child}}$	0.0916	0.0900
$x_i = CE_{i,depthLevel,-}$	0.0037	0.0105

ered only the analysis of the latency since this dissertation is addressed to improve the user-perceived latency. Another important difference is that the correlation was not studied using the speed-up of the latency between both assembly points. Independent correlation studies were done for both latencies. In this section, we have analysed the correlation with the speed-up, because we are truly interested in the improvement between both assembly points in addition to the relative performance results of both cases. Finally, the number of characterization parameters studied in the previous paper was significantly smaller because when we wrote the paper we did not consider the other ones as suitable inputs for the algorithm. After some additional experiments, we took them into consideration.

To summarize, we have studied the correlation between the characterization parameters of the content elements and the improvement of the latencies speed-up between the case when the content elements are assembled in the application server and in the proxy cache. Some of the parameters have been rejected to be used as inputs of the future adaptive core because they do not have a significant correlation. The other parameters, which have shown correlation, are used in further experiments to validate their suitability. These parameters are size, request rate and update rate and the number of children of the father element and the number of fathers of the child element.

3.7 Model for content aggregation systems

The inputs and outputs of the adaptive core of our architecture need to be represented by some type of data structure. We explain, in this section, the details of previous CAS system model representations, and how we have adapted these models to our particular case.

Content aggregation systems are applications in which users set up their own web pages by selecting content sources and putting them into groups. We have called content elements to the single and indivisible contents. Aggregations are the relationships among the contents that a user groups and associates to a web page (template). Different users are able to add or to create aggregation relationships over the same content element. Thus, content elements are aggregated in several pages.

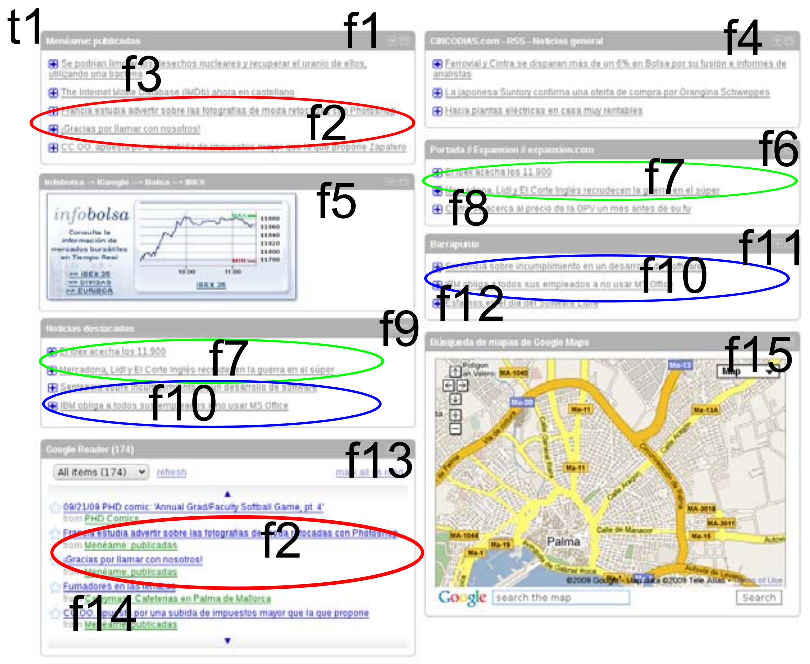
The data structure which best fits in the above explanation is a graph. The vertexes of the graph represent the content elements and their setups. On the one hand, the edges of the graph represent the aggregation relationships created by the users. By definition, content self-aggregation is not allowed, *i.e.* loops are not allowed in the aggregation relationships. On the other hand, aggregation relationships are ordered: one element aggregates the content of other. The aggregated content is considered the child vertex and the element which aggregates the content is the father vertex.

These previous features define the type of graph to represent the content model in a CAS system: a directed acyclic graph (DAG). The graph is directed because the aggregation relationship determines an order (which element includes and which is included). And it is acyclic because it is impossible to create a recursive inclusion of contents.

In Figure 3.4 we can observe an example of a web page generated by the aggregation or combination of contents from different sources. The coloured elements are aggregated more than once.

The content model (DAG) for the example above is shown in Figure 3.5. The web page is generated by the combination of the contents from eight elements (each grey box). Some of these elements are retrieved directly from content sources (f4, f5, f15), and other ones are groups of other content elements (f1={f3,f2}, f6={f8,f7}, f9={f7,f10}, f11={f10,f12}, f13={f14,f2}). The differences among the names of the nodes (use of f or t) are only to identify the template node. The source vertexes of the DAG correspond to this type of nodes.

The use of graphs is not new when we model web pages created by aggregation of contents. Challenger *et al.* [16] defined an Object Dependence Graph (ODG) to represent this type of data. An ODG is a Directed Acyclic Graph (DAG) where content elements are represented by vertexes (nodes).



Menéame: publicadas

Se podrían limpiar los desechos nucleares y recuperar el uranio de ellos, utilizando una bacteria

The Internet Movie Database (IMDb) ahora en castellano

Francia estudia advertir sobre las fotografías de moda retocadas con Photoshop

¡Gracias por llamar con nosotros!

CC.OO. apuesta por una subida de impuestos mayor que la que propone Zapatero

CINCODIAS.com - RSS - Noticias general

Ferrovial y Cintra se disparan más de un 6% en Bolsa por su fusión e informes de analistas

La japonesa Suntory confirma una oferta de compra por Orangina Schweppes

Hacia plantas eléctricas en casa muy rentables

Portada // Expansion // expansion.com

El Ibex acecha los 11.900

Mercadona, Lidl y El Corte Inglés recrudecen la guerra en el súper

Cintra se acerca al precio de la OPV un mes antes de su fu

Infobolsa --> iGoogle --> Bolsa --> IBEX

infobolsa
Consulta la información de mercados bursátiles en Tiempo Real
>> IBEX 35
>> DIVISAS
>> EURIBOR
IBEX 35

Barrapunto

Sentencia sobre incumplimiento en un desarrollo de software

IBM obliga a todos sus empleados a no usar MS Office

Estamos en el día del Software Libre

Noticias destacadas

El Ibex acecha los 11.900

Mercadona, Lidl y El Corte Inglés recrudecen la guerra en el súper

Sentencia sobre incumplimiento en un desarrollo de software

IBM obliga a todos sus empleados a no usar MS Office

Búsqueda de mapas de Google Maps

Map

Palma

©2009 Google - Map data ©2009 Tele Atlas - Terms of Use

search the map

Search

Google Reader (174)

All items (174) refresh mark all as read

09/21/09 PHD comic: 'Annual Grad/Faculty Softball Game, pt. 4'
from PHD Comics

Francia estudia advertir sobre las fotografías de moda retocadas con Photoshop
from Menéame: publicadas

¡Gracias por llamar con nosotros!
from Menéame: publicadas

Fumadores en las terrazas
from Cafeymas - Cafeterías en Palma de Mallorca

CC.OO. apuesta por una subida de impuestos mayor que la que propone
from Menéame: publicadas

Fig. 3.4. Example of content aggregation system web page (iGoogle [32]).

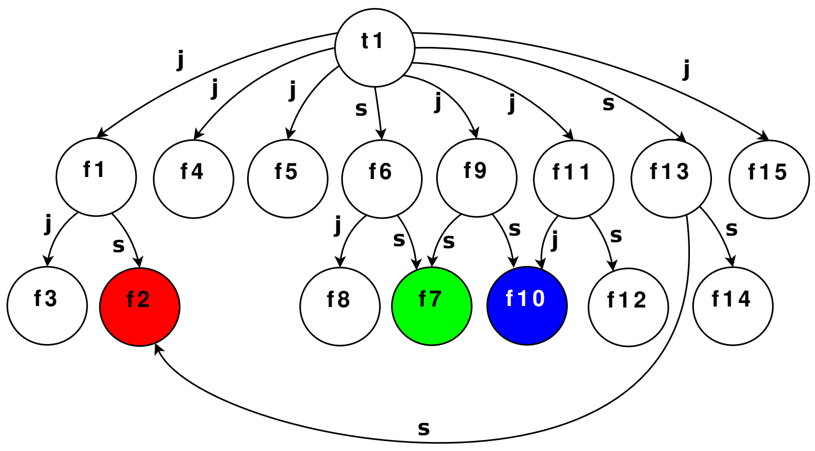


Fig. 3.5. Example of content aggregation graph structure.

An ODG model has two vertex types: P_i which represents parent vertexes (user web pages) and f_i which represents content elements. Finally, the edges of the graph represent aggregation relationships between a pair of elements. This ODG model is not enough to represent all the data that our adaptive core needs, so we extend this model representation. We have called ODGex model to this model extension.

The framework presented in this dissertation suggests pre-assembling some content elements in the application server, creating content fragments. The content fragments are finally assembled in the cache proxy, so the cache is able to manage them independently, and, to improve the hit ratio of the web cache. The model of the content aggregation web pages needs to be extended in order to represent these content fragments (ODGex).

We use labelled edges to identify which elements are pre-assembled in the application server. An example is presented in Figure 3.5. When the aggregation relationship (edges) between two content elements (nodes) is labelled as *join* (j), it means that the application server pre-assembles these two elements. On the contrary, if it is labelled as *split* (s), the elements are independently requested, received and managed by the cache proxy.

The content fragments can be obtained from the extended ODGex model getting all the connected subgraphs when only the *join* edges are taken into account. The process to conceptually distinguish the different content fragments of an ODGex is as follows:

1. Remove all the edges labelled as *split*.
2. Choose the representative nodes of each content fragment. These representative nodes are the nodes without incoming edges or with incoming edges in state *split*.
3. Iterate all the child nodes of each representative node of the content fragments.

This process is not taken in the real framework because the content fragment detection is directly done when the contents are retrieved and assembled. The details are given in Section 5.2.

In the case of the example in Figure 3.5 the fragment elements are: $\{t1$, $f1$, $f3$, $f4$, $f5$, $f9$, $f11$, $f10$, $f15\}$, $\{f2\}$, $\{f6$, $f8\}$, $\{f7\}$, $\{f10\}$, $\{f12\}$,$\{f13\}$, $\{f14\}$. The first node of each sub-set is the representative node, and the nodes of the sub-set are pre-assembled in the application server. This is an example of a fragment design for the given content model. The total number of fragment designs is the number of variations of repetitions of 2 elements (join and split) choose p (where p is the number of edges) and it is calculated as $V'(2,p) = 2^p$.

The basic ODG content model also needs to be extended in order to represent the content characterization parameters which are used to predict the fragment designs. As it is explained in Section 3.5, these characterization parameters associated to each single content element are: content size, content update rate, content request rate, number of other content elements that are included in a given one, and number of other content elements that includes in a given one. The three first parameters need to be explicitly indicated, on the contrary, the two last are implicitly in the graph structure —the number of incoming and outgoing edges of a given node—.

Figure 3.6 is an example of a content model using the extended ODG representation (ODGex). In the example, each node explicitly indicates a tuple of three values corresponding to size, update rate and request rate. The content fragments have been also plotted in the graph (dotted line).

The input of our adaptive core is an ODGex model instance with edges without labels. The adaptive core is in charge of classifying each edge in *join* or *split* state. The output of the algorithm is the ODGex model within the labelled edges.

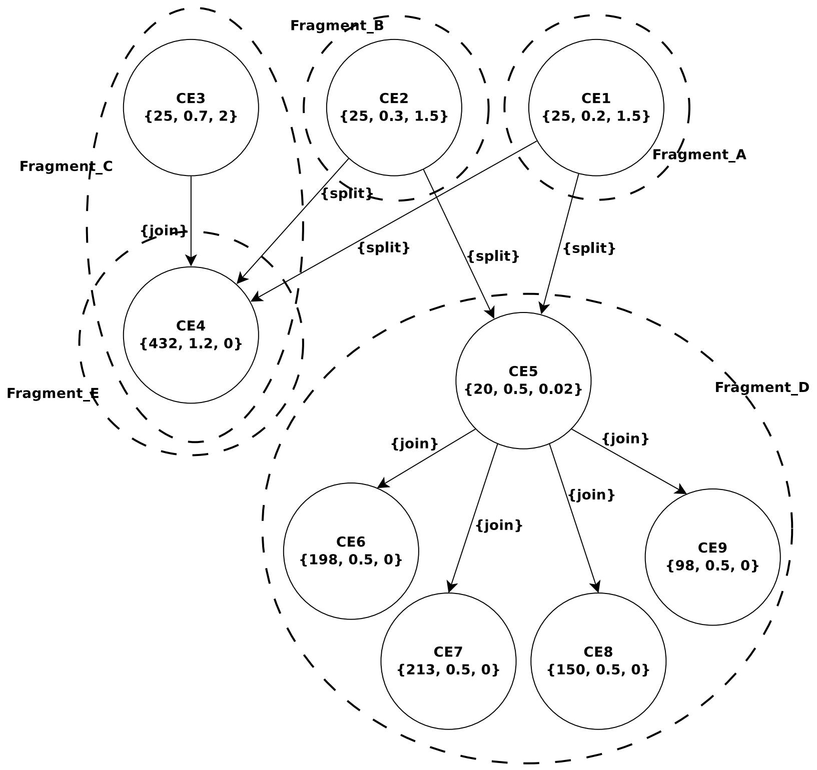


Fig. 3.6. Example of extended content aggregation graph structure.

3.8 Formal definition of the problem

In this section, we define the main problem of this dissertation more formally. We use a mathematical annotation to represent the concepts explained in previous sections.

Define CE^* to be the set of all *content elements* available in a content aggregation system. Define $cp = [cp_1, cp_2, ..., cp_n]$ as the n-tuple of *characterization parameters* for a given content element. In our case, it is a 5-tuple defined by its constituents: *size*, *hitRate*, *requestRate*, $numberFathers$, and $numberChilds$.

An *aggregation*, $a_k = (Fce_k, Cce_k, Fcp_k, Ccp_k)$, is defined by: $Fce_k \in CE^*$, the content element that aggregates the other one, the father; $Cce_k \in CE^*$, the content element that is aggregated by the other one, the child; Fcp_k, the values of the characterization parameters of the father content element; Ccp_k, the values of the characterization parameters of the child content element. A^* is the set of all the aggregation in a system.

Define AT^* as the tier of the architecture where two content elements can be assembled ($appServer, proxyCache$ for our specific system). An *assembly point* ap_k is a tuple which indicates the *application tier*, at_k, where a given *aggregation*, a_k, is assembled: $ap_k = (a_k, at_k)$. Only one assembly point is considered for an aggregation.

A *content fragmentation design*, $D = [ap_1, ap_2, ..., ap_n]$ is the sequence of all the assembly points for each aggregation of a content aggregation system. It defines the assembly point for each aggregation in the system.

The problem of improving the performance by adapting the content fragments is expressed as:

In a given content aggregation system, with a group of content elements CE^* and a group of aggregations A^*. Which is the content fragmentation design D that generates the best performance in terms of user-perceived latency?

3.9 Summary

In this chapter, we have explained the details of our approach to solve the web cache performance limitations of content aggregation systems. Our solution can improve the performance, in terms of user-perceived latency, by pre-assembling some of the content elements in the web application and creating content fragments (sets of content elements). These content fragments are managed independently by the web cache and it finally assembles them to create the web pages.

The main open problem is to define a system that determines which are ideal content element sets (content fragment design) in order to improve the latency, by balancing the improvement of the cache hit ratios and the losses of the assembling overhead times. We have proposed a framework in which an adaptive core determines these content fragment designs and they are adapted as the content of the web pages changes. Several open problems remain for next chapters:

- To discover the most suitable inputs for the adaptive core.
- To define the interfaces to intercommunicate the adaptive core and the content aggregation system.
- To adapt the content aggregation system in order to make it able to manage, identify and request content fragments.

We have explained the details of the experimentation addressed to validate some initial hypotheses of the dissertation. These hypotheses are related to the improvement of the performance by changing the content fragment design, and to the set of parameters to be used to predict ideal fragment designs.

We have benchmarked a real content aggregation system with its real web pages contents and aggregations. This benchmarking has been based on comparing the latency when all the content elements are assembled in the web server and when they are assembled in the proxy cache. The results have shown that the shortest latencies are obtained, in some cases, with assemblies in the application server, and for other cases, in the proxy cache. This validated the hypothesis that different fragment designs have different latencies, and the ideal assembly point is related to the particular web page contents and the aggregation structure and it is not the same point in all the cases.

We have also presented some experiments addressed to obtain a set of content characterization parameters to be further investigated. This future investigation will study if these parameters can be used to predict the optimum content fragmentation design. In the preliminary results, some of the initial parameters have been rejected to be further considered because they did not show significant correlations with the performance results (user-perceived latency). The results for the experiments have been obtained from two different tools: a simulator and a test-bed emulator. Both of them have used models created from statistical distributions. These statistical distributions have been selected using results from research works of other authors.

We have also defined a formal model to represent the contents of a content aggregation system (ODGex), and a formal description of the main problem of the dissertation. The content modelling is based on a DAG structure in which the content aggregations are represented by vertices (content elements) and edges (aggregation relationships). The vertex information has been extended with information about the characterization parameters in order to use the DAG as input of the adaptive core. The edges have been labelled to indicate the places where the assembly of a given aggregation takes places.

To summarize, we present a list with the main contributions of this chapter:

- Definition of a methodology to solve the problem of web caching in content aggregation systems.

- Proposal of a framework to define content fragment designs in order to reduce the user-perceived latency. These fragment designs define the assembly point of the single content elements.
- Definition of a formal model for content aggregation structure, content element characterization parameters, and content fragment designs (ODGex).
- Validation of the hypothesis that different content fragment designs generate different user-perceived latencies.
- Validation of the hypothesis that characterization parameters of the content elements have influence on determining the ideal assembly point.
- Definition of a set of characterization parameters to be used to predict the optimal content fragment design.

4

Definition of the core for the adaptation of content fragment designs

If you do what you've always done, you'll get what you always got.
—Anthony Robbins—

This chapter is about the design and development of the core module of the proposed framework. The core of the framework is in charge of determining the content fragment design in order to improve the performance. The different designs are evolving over time due to the changes in the values of the inputs. Therefore, we have called it adaptive core, because it adapts the fragment design over time.

In this chapter, it is briefly explained some techniques to develop the core. At the end of the chapter, the detailed description of the technique selected to implement the core is done.

4.1 Introduction

The main element of our framework is in charge of adapting the content fragment designs. This core makes the fragment designs evolve over time when the content element characteristics change. All this process is done during execution time. The workload generated by the core system should be very small in order not to penalize the latency of the generation of the web pages. It is a problem of balancing the improvement and the overhead generated by this solution.

The technique to implement the adaptive core remains open in the definition of the framework. Several techniques could be used to implement it. We have explored solutions based on the use of ontologies, genetic algorithms and knowledge discovery/data mining. We have evaluated their requirement accomplishment and their generated overhead. Finally, we have selected a solution based on data mining techniques because it seems to be the best alternative, from our point of view.

We present a description of the adaptive core, and its requirements and features, in Section 4.2. The next section (Section 4.3) is devoted to analyse three alternatives to develop the adaptive core. Finally, Section 4.4 includes the details of the selected alternative, a knowledge discovery based solution.

4.2 Adaptive core definition

The objective of our approach is to create the content fragment designs which generate the highest performance in the system. These content fragment designs should be adapted when the content changes.

One of the most important contribution of the dissertation is the technique to create the algorithm in charge of adapting the fragment designs. One of the requirements of this algorithm is that it should generate a low overhead over the system. Thus, the inputs of the algorithm should be calculated with low overheads. They should be also available as soon as possible after changes in the characteristics of the contents. The suitability of the inputs has been already discussed in Section 3.5. We stated that the characterization parameters of the content elements (aggregation structure, size, request rate and update rate) can be used as inputs of an adaptive system in order to predict the design which generates the highest performance in the system.

The output of the adaptive core is the fragment design for the web pages. This design is represented by the states of the aggregation relationships among the content elements. The content fragments are composed by the content elements which are directly connected using *join* labelled edges. The obtained design should show the highest performance.

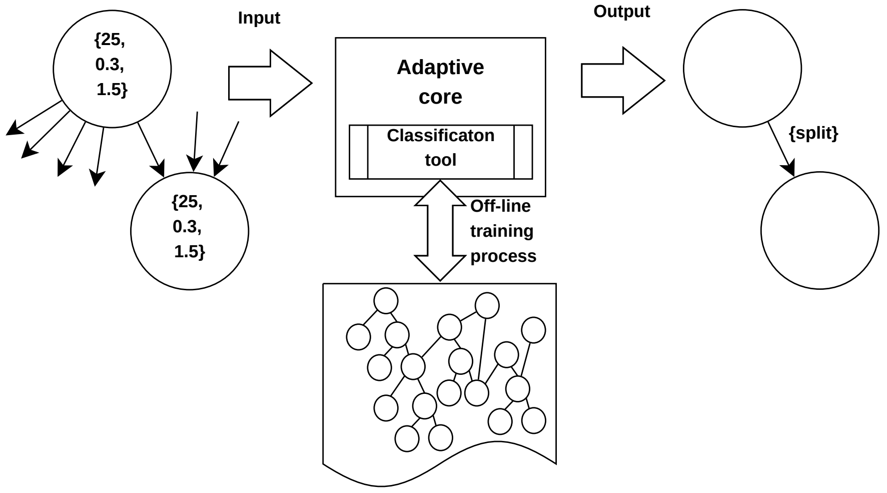


Fig. 4.1. Example of the inputs and outputs of the adaptive core.

In Figure 4.1, we can observe a brief diagram of the core algorithm. The inputs of the adaptive core are size, update rate, request rates, number of children and number of fathers of two aggregated elements. The output of the algorithm is the ODGex edge state. The algorithm adapts the fragment designs by changing the states of the aggregation relationship. The value of the state is decided by the characterization parameter values of both related content elements. These values are used by a classification algorithm, which decides the state. The execution of the algorithm generates a low overhead, and the creation of the algorithm is done previously in an off-line phase.

The details of the process to create the classification algorithm and the study of alternatives are given in the next section.

4.3 Alternatives to implement the adaptive core

The number of techniques to implement the adaptive core is very high. We have studied the applicability of some techniques which, *a priori*, seem to be suitable in our case.

We have dealt with the solution as an optimization process. This optimization solution needs to classify the ODGex edges in one of the two available states, *join* or *split*. The resulting classification should create the fragment design which generates the highest performance in the system. Two main requirements must be achieved by the solution:

- Overhead: the solution should generate as low overhead as possible. If the overhead is very high, the improvement of the system will be counteracted.
- Immediacy: the solution should be generated as soon as possible, because the contents change with high rates. If the solution takes a lot of time to be found, it will not be obtained before new content changes.

The studied solutions for the developing of the adaptive core are quite heterogeneous. More precisely, we studied the suitability of solutions based on ontologies, genetic algorithms and knowledge discovery/data mining. The details are explained in the next sections.

4.3.1 Ontologies

Ontologies are a hot topic in current research studies and they are widely used in reasoning and artificial intelligence problems. Ontologies are formal representations of knowledge, as a set of concepts within a domain, and the relationships among them. Due to their implicit logic, they are easily used for reasoning processes based on classifications.

Ontologies offer an opportunity to significantly improve knowledge management capabilities. Therefore, we considered using them as a possible implementation of our adaptive core. Problems related to resource and computational requirements made us to reject their use, but we obtained some interesting contributions that we are highlighting in this section. The work done in this phase involved: the definition of the system architecture to enable the use of ontologies; the definition of the ontologies for the different domains of the problem; and, a simple reasoning process to monitor its resource consumption.

For the integration of an ontology-based solution, we proposed an architecture which is the same to the one proposed in Section 3.3, but including some singularities of the ontological systems. Gathering modules were proposed in order to obtain the data to build the ontologies and their instances. A reasoning module was also defined in order to use the knowledge expressed in the ontologies.

The architecture is based on ontologies to model all the elements in the system (Web System Elements Knowledge Base) and the behaviour of the users (Behaviour Knowledge Base). The reasoner analyses the knowledge provided by both knowledge bases in order to define the configuration of the system that obtains the highest performance. Due to the similarities to the final architecture and because this solution was not finally considered as the right one for our system, we avoid giving the details, but they are explained in [35, 37]. The next ontologies for our specific knowledge domains were defined:

- Content aggregation system domain. This domain represents the knowledge related to the content aggregation web pages and applications. The concepts of this domain represent the content elements, the content fragments, the fragment design, the aggregation relationship, etc.
- User behaviour domain. This domain represents the knowledge related to how the users interact with the system, their content preferences, and how they configure their custom web pages.
- Web performance domain. This domain represents knowledge about how the system behaves and about the performance metrics of different hardware and software elements of the system.
- Web architecture domain. This domain represents the knowledge about the elements of the architecture (hardware and software) which take part in a content aggregation web application.

The details of the ontologies that we created for each domain are explained in [39, 38].

Finally, some tools based on ontologies were created to reason using the instances of the models. We have shown that these techniques are valid to solve our problem. However, we noticed problems regarding with resource consumptions. High demand was generated over computational and space resources. Thus, one of the most important requirements of the solution, simplicity and low overhead, was not accomplished. We decided to investigate other techniques to implement the adaptive core of the framework.

4.3.2 Genetic algorithms

Genetic algorithms (GA) are heuristic search tools [3]. They are based on the process of natural evolution. The solution search is based on changes over a population of solutions. The changes are addressed by the application of operations. These operations are inheritance, crossover, mutation and selection. A very brief explanation of the genetic evolution is:

- Two individuals (solutions) are selected from the population (group of solutions). The selection process is done randomly, but not uniformly. Each individual has associated a probability of being selected and these probabilities are assigned using a fitness function.
- These two individuals are combined with a crossover operation. The resulting solution is created by combining parts from each of the two parent solutions.
- In some cases, a small part of the resulting solution is changed randomly. This is a mutation process, and it takes place with small probabilities.
- The worst solution from the population, individual with the lowest fitness value, is replaced by the new solution.

The creation of genetic algorithms involves decisions about how to represent the solutions of the problem, how to define the genetic operations (crossover and mutation), and how to define the fitness function. The rest of the section is devoted to explain the application of these three steps in our problem.

GAs usually represent the solution of the problem as a string of 0s and 1s, but other encodings are also possible. The solution of our system is a DAG with labelled edges. The number of different labels for the edges is 2. A graph can be modelled as a matrix of NxN elements that represents the edges between vertexes [9]. This type of representation fits perfectly in GA.

Therefore, the solution of our problem can be represented using a string of size NxN, where N is the number of content elements in the CAS, and the possible values for each element are -1 (*join* state), 1 (*split* state), and 0 (no aggregation relationship between these content elements). But not all the strings are solutions in our case, only the acyclic graphs representations. Moreover, the structure of the pages, or content page model, does not change while the optimal solution is been searched. Therefore, all the possible solutions should be DAGs with the same edges. The differences between them are the states of the edges, *i.e.*, the 0-value positions of the strings remain equal.

In the initialization phase, many single solutions are randomly generated to create an initial population. The graph structure of all these solutions should be the same to the one of the content page model. All the solutions are generated using a graph mask which represent the real structure of the content page model. The edges, which correspond to aggregation relationships in the graph mask, are randomly labelled with 1 or -1. Once this is ensured, it is not necessary to check it again each time that an operation is applied. Only elements with values 1 or -1 can evolve. Conceptually, the population evolves by changing the fragment design, the values of the edge labels, not by changing the content structure, creating or removing edges. Therefore, changes in 0 elements are not considered.

Next step in the design of the system is to define the evolving operations: crossover and mutation. As we have explained in previous paragraphs, the 0 elements of the solutions are not changed by any operation. The mutation operator randomly selects one of the edges of the graph, and changes the value of the label of this edge, its state. A second alternative for the mutation operator is to change the state of all the descendant edges instead of only the selected one. Analysing the results of some experiments, we concluded that this second method is better to avoid local minimums in the solution search.

The crossover operator creates a new individual from the combination of the information of two given father elements. In our case, this results in taking some edge labels from one of the father elements, and the rest of labels from the other one. We studied two alternative crossover operators. One of them takes the definition of one father solution and replaces all the elements between two random indexes with the elements in the other father solution. This operator behaves as the most basic crossover operator in GA.

In the second crossover operator, the replacement process is based on the structure of the graph instead of the solution string. The solution of one of the father individuals is taken as the reference solution, and a node is randomly selected. The values of all the descendant edges of this node are replaced by the element values in the other father solution. Our experiments showed that this last operator achieves an optimal solution more quickly.

Finally, the definition of the fitness function should be done. This phase made us to reject the use of GA in the deployment of the adaptive core. In fact, the designing of the fitness function is the hardest work in GA. Fitness function maps the individuals of the population in an ordered set. Since the population has a high number of individuals, the fitness value should be calculated very easily and quickly. Some of our experiments were addressed to calculate the fitness value by measuring the real performance for each solution. Obviously, this is not a quick and easy method to measure the suitability of each solution. Therefore, the measurement of the performance for each solution was rejected as a fitness function.

We used the parameters explained in Section 3.5 in order to create the fitness function. Our preliminary study proved that the characterization parameters can be used to decide the state in which two fragments generate the highest performance. But this study did not conclude anything about the mathematical relationship among the parameter values and the performance values, and even less about some kind of sort function. In fact, we observed that the relation between the characterization parameters and the aggregations states is defined by patterns instead of a direct mathematical relationship. Therefore, we finally tried data mining techniques to classify the aggregation relationships.

4.3.3 Data mining

Data mining is a set of techniques to discover knowledge in large data sets. The emphasis of data mining lies on the discovery of previous unknown patterns. One of our dissertation goals is to prove that this previous patterns can be used to improve the performance in the future, *i.e.*, that a predictive analysis can be used to improve the performance of a CAS, by classifying new data instances.

Data mining is a task of a more general technique, Knowledge Discovery (KD). We have generally called data mining to all the phases involved in the KD, despite we have actually deployed a complete KD process. This is because we wanted to focus the explanation of our work in the task of mining the data but, in a more precise way, we should call it as knowledge discover instead of data mining.

A group of tasks and phases should be done when knowledge discovery techniques are applied to a specific problem [27]: (a) Selection or target data definition; (b) Preprocessing or data properties analysis; (c) Transformation of the target data: data cleaning, and instance representation; (d) Data mining algorithm selection; (e) Evaluation and coverage study (testing data set). Figure 4.2 also shows the phases of the KD process.

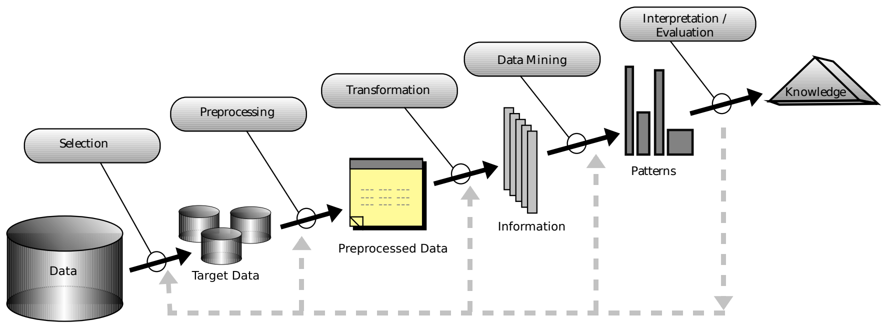


Fig. 4.2. An overview of the steps that composes the Knowledge Discovery process (source: [27]).

The target data definition consists in defining the independent attributes, the class attribute, and the available data samples. The class attribute is defined or classified by the patterns extracted using data mining. In our case, this attribute is the state of the aggregation relationship of two content elements. The independent attributes are used to decide the value of the class one. We previously showed correlation among characterization parameters of the content elements and the aggregation state, so these are our independent attributes. We created a big enough data set in a synthetically way in order to use them as the data samples. The samples were created by simulating content page models in a real web system.

The data properties analysis consists in analysing the histograms and the scatter plots, and in detecting atypical values. In our research process, we graphically analysed the data obtained from the emulation results. By this analysis, we observed patterns in the data. This fact supported the idea of using data mining techniques and knowledge discovery in our problem resolution. We created an initial solution by creating association rules straight from the observation of the data, and we obtained good results.

In the phase of the transformation of the target data, we took into account more than one data representation. Since we previously proved that we can use both entire values of the characterization parameters or mathematical relations among their values, we created four different data representation sets. These sets combine entire representations with other ones where values are represented by mathematical relationships obtained with low processor overhead.

After the transformation of the data, we need to select a data mining algorithm which generates a classification structure. Decision trees and association rules are usual implementations for process of classification [47]. We need to obtain the structure, or classification algorithm, which uses less computational resources to classify the data instances.

Finally, the evaluation and coverage study is used to check the suitability of the obtained knowledge. In our case, experiments with real systems have been done because a coverage analysis is not enough. A coverage analysis only gives information about the number of well-classified instances using the knowledge obtained in the data mining process.

Since we have finally implemented the solution to our system with data mining techniques and knowledge discovery, the details of each knowledge discovery phase have not been explained in this section. They are in the next section (Section 4.4).

4.4 Adaptive core deployment via knowledge discovery

In the initial steps of the research work, we explored different alternatives to deploy the adaptive core. Finally, we decided to implement a data mining based solution. In the previous section, some details about the knowledge discovery process have been explained.

This section is addressed to give the details of the whole process. The section is divided in subsections which correspond to each of the phases involved in a knowledge discovery process. The details of the adaptation of each of these phases to our particular problem are explained.

4.4.1 Target data definition

In the data definition phase, the independent attributes and the class ones are defined. We have previously studied suitable attributes in order to detect patterns in the performance behaviour of the system. We analysed the relation between the *best* aggregation state, from a performance point of view, and the characterization parameters of two related content elements in Section 3.5.

The objective of the adaptive core is to determine the fragment design in which the system experiments the highest performance. Fragment designs are defined by the state in which the single aggregation relationship is classified or the edge label, if we refer to the ODGex model. Thus, the output of the adaptive core is the state, or edge label, of each aggregation. Our class attribute is a state of the aggregation relationship.

Finally, the independent attributes are used to predict or classified the class attribute. As we have mentioned before, we proved that a set of characterization metrics of the content elements can predict the *best* state, from a performance point of view, of each aggregation relationship. Therefore, the independent attributes are: request rate, update rate, size, number of fathers, and number of children.

Obviously, the knowledge extracted for the data obtained in an architecture would not be the same than for a different architecture. This difference is mainly based on the hardware, the tools to deploy the system, the software, the operating system, the setup of the system, etc., *i.e.* on the system architecture. A clear consequence of this is that a new data mining process should be done when a change in the architecture occurs, in order to update the patterns that are used in the predictions.

The attributes that take part in the process are the same, either the independent ones as the class one, but the data sets (data instances) are not equal in scenarios for different architectures. Therefore, we should define a systematic process in order to obtain the training data set. The steps to obtain this data set are:

- Creation of a content model (set of content elements, content fragments and web pages) with a wide range of values for the characterization parameters of the content elements: content size, request rate, update rate, number of aggregators (fathers) and number of aggregations (children).
- Definition of a random fragment design over the page model, *i.e.*, assignation of the values of the edges of the ODGex model.
- Emulation of the content model. During the emulation, the values of some edges of the ODGex model are changed periodically.
- Gathering the performance metrics from the emulation. The performance metrics are related to the latency.
- Comparison of the latencies among the web pages in which only the value of one state is different. The state with the shortest latency is selected as the best one.
- Creation of the data instances using the best state and the characterization parameters of the two content elements involved in the analysed aggregation relationship.

Two important issues to be solved in this process are the creation of the ODGex model to be emulated and the condition to finish the emulation. These two phases are explained below.

Firstly, we focus on the creation of a synthetic data set for the training phase of the data mining process. These data are obtained from a random content model, or ODGex model. The knowledge extracted from the data set needs to represent as many patterns as possible, and not only patterns of the most usual cases. Therefore, we are interested in creating an ODGex model which covers a wide range of values for the characterization parameters, *i.e.*, we are interested in a homogeneous model —all the values have the same relative importance— and we need to use uniform distributions to create the model.

It is impossible to create a general ODGex model useful for all the cases, because the number of cases is infinite. Some value limits need to be fixed before the creation of the model. We need to have some details of the characteristics of the content elements we are going to manage in the real system.

The model is composed of three characterization metrics —size, request rate and update rate of a content element— and two structure metrics — number of aggregations (children) and number of aggregators (fathers) of a content element—. We need to create all of them in a uniformly way.

The uniform statistical distribution is the mathematical tool in order to create uniform distributed data instances of a variable. The parameters of a uniform distribution are the maximum and minimum values of the random variable. Therefore, we need to know the maximum and minimum values for the size, the request rate and the update rate of the content elements.

The creation of a uniform structure is more complex than the case of samples from a random variable. The structure that represents an ODGex is a directed acyclic graph (DAG), so we need to use some kind of algorithm to create uniform graphs. Some of the most usual algorithms to create random graphs are defined by Barabási [4], Kumar [57] and Broder [10].

These new models, which are focused in the creation of random Web-like graph models, strongly differ from the classic view of random graph models. All they are better approximations to represent real networks and graph structures. But, as we have mentioned before, we are more interested in a model to cover a wide range of values.

Classic models ($G_{n,p}$) consider a fixed number of n vertexes and each edge occurs independently of all other edges with a fixed probability p. The properties of these classic models are deeply studied and analysed [24, 25, 7]. These publications define the Erdös-Rényi model in which the process of creating a uniform random graph is defined. Since we are interested in a directed acyclic one, some modifications over the general process have been done.

Summarizing, the process in order to create the ODGex model is stated by the next steps:

1. Consider a set of n vertexes (content elements).
2. Generate randomly the samples for the size and the update rate of each vertex (but not the request rate). Uniform distributions, with their particular maximum and minimum parameters, are used to generate the samples.

3. Create a directed edge (aggregation relationship) connecting two random vertexes (content elements). The new edge has to be rejected if it connects two previously connected vertexes, or when it creates a cycle in the graph. Finally, assign randomly the label (aggregation state) to the edge.
4. Repeat the last step until the total number of edges N is reached.

The case of the request rate is particular. It is more difficult to predict the request rate for single content elements than for web pages. Thus, we randomly assign the request rate of the source vertexes of the graph using a uniform distribution. Once the creation of the ODGex model has finished, we walk the graph and we recalculate the single content element request rates of each single vertex by adding the request rates of all its fathers.

Table 4.1. Parametrization of the process to create the training data set.

Parameter	Description
min_{size}	Minimum size of a content element
max_{size}	Maximal size of a content element
min_{req}	Minimum request rate of a CAS web page
max_{req}	Maximal request rate of a CAS web page
min_{upd}	Minimum update rate of a content element
max_{upd}	Maximal update rate of a content element
$total_{ce}$	Total number of content elements in the ODGex model
$total_{ag}$	Total number of content aggregations in the ODGex model

Table 4.1 lists the setup values we need to know from the real content model in the time of creating the training data set. The six first characteristics correspond to the parameters of uniform distributions.

The other two characteristics to be used in the creation of the training data set are the total number of content elements and aggregation relationships. If we assign to the total number of vertexes n and of edges N the real values of these two parameters, we would not probably obtain an enough number of data instances to extract a proper knowledge. We need to scale the number of elements in the content in order to create an big enough training data set. The main issue of this scaling process is to keep constant the average degree or valency of the graph, by multiplying both values with the same number s ($n = s * total_{ce}; N = s * total_{ag}$).

In our experiments, the value of s was assigned arbitrarily (around $s =$ 1000) and it resulted in a high enough value. Probably, it might be smaller but experimentation is the only way that we have to determine this value. Further research work should be done in order to define some kind of formula to calculate the value of s more accurately. This formula should consider interval distances of the setup parameters related with the size, the update rate and the request rate. We consider this work is out of the bounds of our dissertation and it remains as future research work.

Secondly, we focus on the determination of the end condition of the emulation. The data instances of our training data sets are composed by a class attribute that is determined by comparing the latencies of the two possible states of an aggregation relationship. Web latencies are not deterministic and they may vary with external conditions of the system as, for example, network traffic, operating system processes, etc. The measures of the performance (user-perceived latency) of the emulator should be done in an isolated environment. But a complete isolated environment cannot be achieved. Therefore, we need more than a single measure for the latency of each web page. The main issue is to determine the number of measures that we should do to get reliable values.

In order to guarantee the reliability of the results, we have used a multiple replica method. In this type of methods, experiments are repeated several times. Each repetition is called replica. The accuracy of the results gets higher as the number of replicas is increased (n). The bias decreases as the number of samples of the replicas is increased (m).

In statistics, a confidence interval (CI) is a particular kind of interval estimation of a population parameter. We have used it to indicate the reliability of the metric we want to estimate, the average latency of the web pages. In our case, we have calculated the reliability of the average latencies of the samples of a given replica, and also the reliability of the averages latencies of the same requests (samples) among different replicas.

In the case of the average latencies for the samples of a replica, the latency of the web pages varies between requests of the same web page. This is because of the influence of the hit and miss ratios of the content fragment elements that compose a web page. Therefore, we are not able to work only with one average value of the latency. We have analysed it separately for each individual case. We have studied the average latency for each content fragment element and we have divided them into two cases: cache hits and misses. In the case of the requests among different replicas, we have calculated the single latency of each single sample or request.

In both cases, we have calculated the 95% confidence level for the average times. We have considered reliable results when the CI is smaller than $\pm 1.25\%$ of the average latency. When this was not accomplished for all the samples of the experiments, the size of the experiment —number of replicas n and number of samples in a replica m— was increased. We experimentally discovered that the size of the experiment got too big if this requirement had to be achieved by all the elements of the experiment. We relaxed the requirement so we considered to achieve this in only the 90% of the elements of the experiment. In this case, the size of the experiments resulted in manageable terms.

The transient period is the period of time in which the initial state of the experiment has influence on the results, *i.e.*, it is the time during which a given metric shows alterations in its value. The samples of this period are not considered for the statistical studies, and only those of the steady state period are used. The components of a system are not in the same state in the starting point than in the steady state. For example, in our case, the cache is completely empty when the system starts. Thus, the hit ratio of the cache is 0 when the system starts the execution. The cache is filled as the web pages are requested by users. During this period of time the hit ratio is increased until a constant value is reached. In fact, the hit ratio is the performance metric we have used to define the size of the transient period (l).

The transient period can be determined by visual analysis or by some mathematical relation —the difference of the two last values is less than a given value x—. We have used a mathematical method to determine the size l of the transient period. The metric we have analysed is the hit ratio of the cache. We consider the steady state has been reached when the value of the hit ratio remains with small alterations. Different execution replicas, using the same content page model (ODGex), have been considered in order to define the transient state. The first l samples of the future replicas have to be rejected in order to study the mean latency.

4.4.2 Preprocessing and data properties analysis

This phase is addressed to prove the relation between the variables used for the independent attributes and the class attribute. This analysis can be done by observing graphical representations of the data (histograms, scatter plots, etc.) or by the use of multivariate statistics, as linear regression or correlation. Our research work covers both types of analysis.

On the one hand, we have already presented a multivariate analysis in Section 3.6. We analysed the correlation between some characterization parameters and the state in which the highest performance is experimented. This preliminary study helped us to reject the use of some of the initial parameters and to take into account six characterization parameters (size, request rate and update rate of the father and the child content elements, number of aggregations in the father, and number of aggregators of the child). The suitability of the remaining ones needs to be tested in the validation phase. Therefore, we have created and analysed more than one target data set. Each of these sets removes one of the attributes. Thus, we can compare the results that are obtained when a given attribute is used or not.

On the other hand, the properties of the data were analysed by the use of graphical representation. There is usually a big number of independent parameters involved in data mining process. Multidimensional representation is needed to analyse graphically the data. Scatter plots are not multivariate representation, but we can use several bivariate representations, in which the multivariate data are projected into multiple two-dimensional plots. Figure 4.3 is an example of this type of representation.

The analysis of the scatter plots gave us some clues about the patterns that could be obtained from the training data. We used this information in order

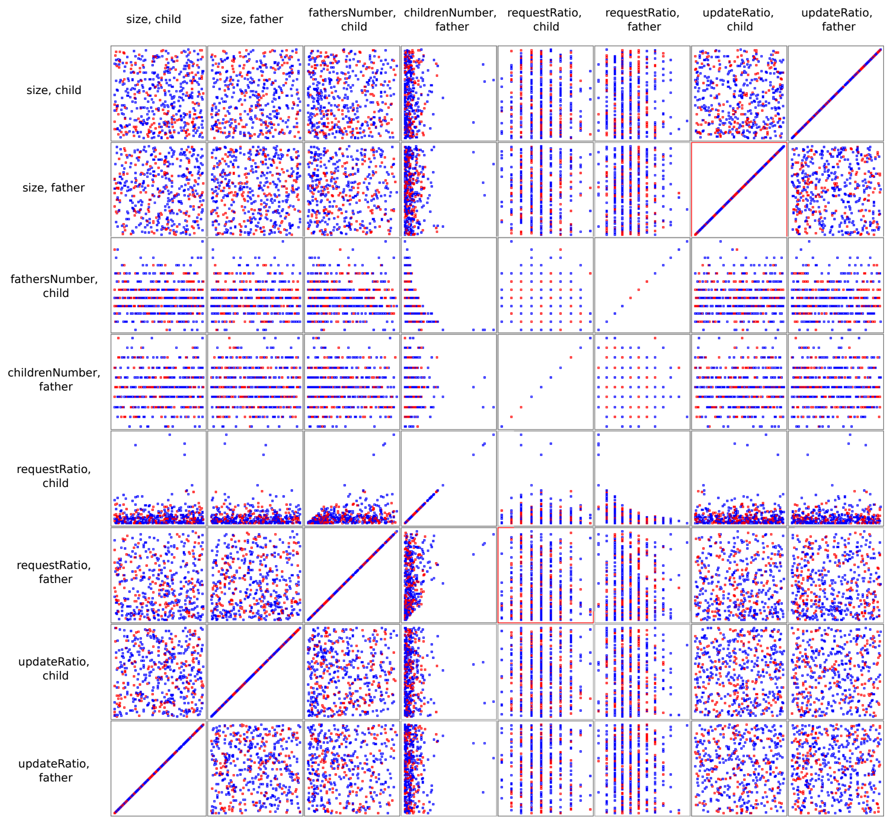


Fig. 4.3. Example of a multivariate scatter plot representation.

to create a small preliminary experiment. This initial experiment used an association rule system to classify the data instances. The rules were created by observing the scatter plots. The results were quite good despite using a small training data set and of creating the rules by only the observation of the scatter plots.

The association rule set was very simple with a very small number of rules. We have represented graphically the rules that we created in Figure 4.4. The rules are represented as a tree structure because it is possible to represent association rules as decision trees and *vice versa*. We published the details of this previous work and the results in [40]. The list of the rules is:

- If the update rates of the child and father elements are higher than the request rates, then the state is set to *join*.
- If only one of the element has its update rate higher than the request rate, then the state is set to *split*.
- If neither of the update rates is higher than the request rate:

– If the child content element has more than one aggregator (father vertices), then the state is set to *split*.
– If the previous condition is not accomplished and the number of aggregations (children vertices) of the father element is 1, then the state is set to *join*.
– If neither of both previous conditions is accomplished and the size of the child content element is bigger than 25 KB, then the state is set to *split*, but if the size is smaller than 25 KB, then the state is set to *join*.

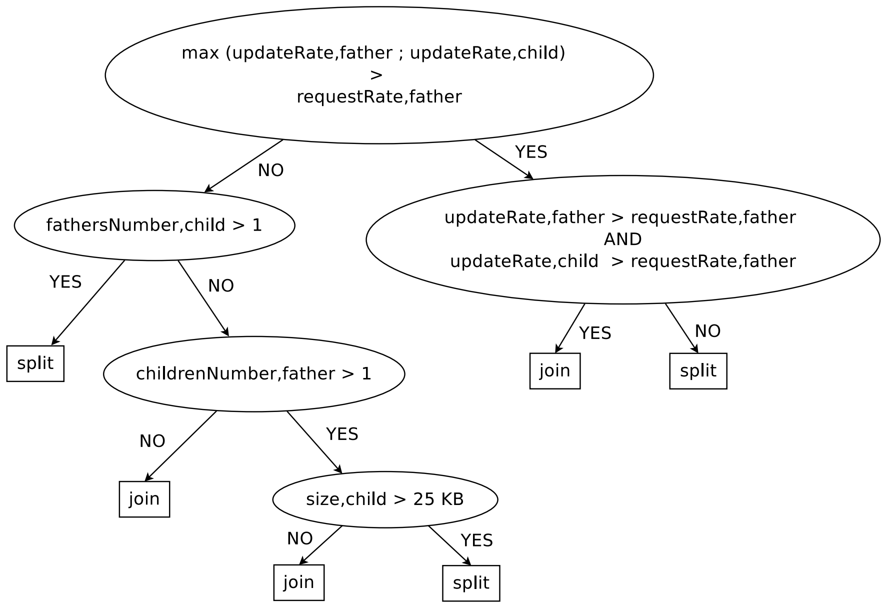

max (updateRate,father ; updateRate,child)
>
requestRate,father
NO
YES
fathersNumber,child > 1
updateRate,father > requestRate,father
AND
updateRate,child > requestRate,father
YES
NO
split
childrenNumber,father > 1
YES
NO
join
split
NO
YES
join
size,child > 25 KB
NO
YES
join
split

Fig. 4.4. Graphical representation of the association rules for the preliminary test.

4.4.3 Transformation of the target data: data cleaning, and instance representation

We have mentioned previously, in Section 3.6, that further investigation over the independent attributes should be conducted. There are two main issues to be validated: the suitability of all the independent parameters and the best way to represent them —by entire values or by related ones—. We have designed an experiment in which different instance representations are taken into account depending on these two issues.

We proved that the correlation among aggregation states and characterization parameters exists both when the values of the common attributes are expressed using entire values and when they are expressed as a mathematical relationship. The common parameters are those present in father and child content elements ($CE_{size,father}$, $CE_{size,child}$, $CE_{requestRate,father}$, $CE_{requestRate,child}$, $CE_{updateRate,father}$, $CE_{updateRate,child}$). The other parameters are the individual ones and they are only considered for one of the content elements ($CE_{childrenNumber,father}$, $CE_{fathersNumber,child}$).

The data instances are represented in four different ways, regarding with the relation of the attributes:

- *Entire*, the common parameters are expressed independently.
  $\{CE_{attribute,father}, CE_{attribute,child}\}$
- *Ratio*, the common parameters are expressed as a division.
  $\{\frac{CE_{attribute,father}}{CE_{attribute,child}}\}$
- *Difference*, the common parameters are expressed as a difference.
  $\{CE_{attribute,father} - CE_{attribute,child}\}$
- *Distance*, the common parameters are expressed as the absolute value of their differences.
  $\{|CE_{attribute,father} - CE_{attribute,child}|\}$

The individual parameters are expressed independently in the four patterns. All these attributes are the independent ones, and all of them are numerical attributes. Finally, each data instance is completed with the class attribute, corresponding to the best state, from a performance point of view. The class attribute has only two possible values, both states of an edge (*split* or *join*). Table 4.2 includes examples of the four representations.

The second issue to consider is the validation of all the attributes. This issue generates the creation of six data representations where some attributes are removed. The common attributes are removed together. Thus, there are six target data representations: one in which all the attributes are considered; three more in which the pair of attributes corresponding to the request rate, update rate and size are removed; one in which the number of fathers is removed; and, finally, one in which the number of children is removed.

Considering both discussed issues, the total number of data instance representation is 24. In order to identify the data sets, they are called as $DataSet_{relationPattern,removedAttribute}$ where *relationPattern* indicates the type of mathematical relation among common attributes (*Entire*, *Ratio*, *Difference* and *Distance*) and *removedAttribute* indicates the attribute which has been removed from the vector. Its value is $\emptyset$ if none attribute has been removed. For example, $DataSet_{ratio,updateRate}$ is the data instance representation in which common attributes are expressed as a division and the update rate attributes have been removed. $DataSet_{ratio,\emptyset}$ is the same case, but without removing any attribute.

Each of the 24 different representations for the target data is used to represent the instances of one of the 24 different training data sets. Afterwards, these data sets are mined and knowledge is extracted from them. The results are studied in an experimental environment and the performance results are

Table 4.2. Data instances representation for the training data sets.

Entire representation
$\{CE_{requestRate,father}$, $CE_{requestRate,child}$, $CE_{updateRate,father}$, $CE_{updateRate,child}$, $CE_{size,father}$, $CE_{size,child}$, $CE_{childrenNumber,father}$, $CE_{fathersNumber,child}$, StateBestPerf $\}$ Ex.: {2.1, 3.0, 1, 0.5, 324, 250, 5, 3, split}
Ratio representation
$\{CE_{requestRate,father}$ / $CE_{requestRate,child}$, $CE_{updateRate,father}$ / $CE_{updateRate,child}$, $CE_{size,father}$ / $CE_{size,child}$, $CE_{childrenNumber,father}$, $CE_{fathersNumber,child}$, StateBestPerf $\}$ Ex.: {0.7, 2, 1.296, 5, 3, split}
Difference representation
$\{CE_{requestRate,father}$ - $CE_{requestRate,child}$, $CE_{updateRate,father}$ - $CE_{updateRate,child}$, $CE_{size,father}$ - $CE_{size,child}$, $CE_{childrenNumber,father}$, $CE_{fathersNumber,child}$, StateBestPerf $\}$ Ex.: {-0.9, 0.5, 74, 5, 3, split}
Distance representation
$\{\lvert CE_{requestRate,father}$ - $CE_{requestRate,child}\rvert$, $\lvert CE_{updateRate,father}$ - $CE_{updateRate,child}\rvert$, $\lvert CE_{size,father}$ - $CE_{size,child}\rvert$, $CE_{childrenNumber,father}$, $CE_{fathersNumber,child}$, StateBestPerf $\}$ Ex.:{0.9, 0.5, 74, 5, 3, split}

compared among them. If the best results correspond to data representation in which a particular attribute has been removed, we will be able to conclude that the removed attribute does not have influence on the performance. On the contrary, all the attributes will be important if the best results are obtained in the experiments where all the attributes are used. The same analysis can be done for the experiments in which different mathematical relations have been used for the common attributes.

4.4.4 Data mining algorithm selection

We are interested in the creation of a classification algorithm which predicts the class membership of new instances based on a series of measurements in that instance. There are many methods available in order to create classification algorithms [59, 47]. But there is not a general guideline to select the best method.

The usual criterion to decide the best algorithm or method is the performance of the process (related to the time to create the classification algorithm) and the coverage of the created classification algorithm. Our problem has some particular features which make us to consider another criteria in the decision.

The performance of creating the classification algorithm is not a problem because this is an off-line task that does not have influence on the performance in running time. The main issue is that the execution of the classification process should take the shortest time, in order to reduce the overhead of the system. Finally, the coverage of the classification is important.

In order to select the best algorithm, we have analysed the performance of the classification process. We have used several training data set, created by the process explained in Section 4.4.1. We have applied several data mining algorithm over these training data sets. We have finally studied the coverage and the classification time of these classification algorithms. Both metrics have been calculated over the same data sets used as training sets.

The criterion to select the data mining algorithm is to obtain the shortest classification time and the highest coverage. We have analysed the most usual data mining algorithms for general purpose. The results for each algorithm are presented in Table 4.3.

Table 4.3. Comparative analysis among data mining algorithms.

Type	Name	Classification time (ms)	Coverage (%)
Tree	J48	3.349	90.258
Tree	NBTree	20.221	90.552
Tree	RandomForest	35.639	85.414
Tree	RepTree	3.832	86.814
Rules	DecisionTable	9.675	90.552
Rules	JRip	2.913	86.544
Rules	OneR	2.906	84.042
Rules	PART	4.830	89.600

Weka [84] has been used to create all the knowledge representation for each data mining algorithm. Weka (Waikato Environment for Knowledge Analysis) is a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand.

In Table 4.3, the mean classification time of one instance and the coverage of the classification process are presented. After the analysis of the results, we have selected the J48 algorithm as the most suitable in our case. It is in the set of the three best algorithms in both criteria. J48 is the implementation of Weka for C4.5 algorithm, which was developed by Quinlan [70]. C4.5 is probably the most popular data mining algorithm to create tree classifiers.

4.4.5 Evaluation and coverage study

One of the quality metrics of the resulting algorithm of a data mining process is the coverage. The coverage is the percentage of instances that are classified correctly. Thus, it is needed a data set in which the class attributes of the instances are previously known. This set is usually called as testing data set. The number of data instances of the testing data set is usually significantly smaller than the number of the training one. The classification algorithm is executed using the data instances of the training data set as inputs, and the classification results are compared with the correct class attribute. The number of hits and misses in the classification is evaluated and, finally, the percentage of hits, or coverage, is calculated. In some cases, a reduce number of data instances from the training data set are used to evaluate the resulting algorithm, but it is recommended not to use the same data instances in both processes.

We need a second data set every time that we create a classification algorithm. Therefore, the process explained in Section 4.4.1 needs to be done twice: once to create the training data set and another one to create the testing data set.

There is not a rule to define a minimum value for the coverage. The strength of the coverage depends on the problem. Moreover, the coverage is not a direct metric of the benefits of our solution. A correct classification does not generate a higher performance on the system in some cases. So, we consider that good coverages values are around 90%.

4.5 Summary

Our dissertation contributes with the definition of an architecture for content aggregation systems in which the content fragments are adapted as the content elements change. The details about the proposed framework (interfaces, modules, etc.) are given in the next chapter (Chapter 5). The most important element of the framework is the adaptive core. This element is in charge of determining the fragment design, the content elements which form a content fragment. The deployment and implementation of the adaptive core is independent of the definition of the framework, *i.e.*, the architecture is independent of the algorithm that creates the content fragments. In this chapter, we have studied different alternatives to implement this core.

The first part of the chapter is focused on the definition of the requirements of the adaptive core. The core needs to manage the fragment design adaptation as changes occur in the characterization parameters of the content elements. Also, the overhead that the adaptive core generates over the system should be as low as possible.

The second part of the chapter has been devoted to study the use of three techniques to implement the adaptive core: ontologies, genetic algorithms and knowledge discovery. This study helped us to select the last one as the most suitable method to implement the adaptive core. The other two alternatives presented important drawbacks. The reason to reject the use of ontologies has been the high computational requirements to implement the solution. Genetic algorithms have been rejected because of the difficulty to create a fitness function and because of the long time between an update occurs and a solution is found. In any case, we got some contributions from the results of the analysis of the two rejected alternatives.

The latest part of the chapter has been devoted to give the details of all the process for the deployment of the adaptive core using knowledge discovery. The details of the adaptation to our case have been explained. We have explained how the phases of a knowledge discovery process have been addressed to our particular case. We explain, in the next chapter, how the adaptive core is included in the framework for the adaptation of the content fragments.

To summarize, the contributions of this chapter are:

- Definition of the requirements and features to be achieved by the adaptive core.
- Study of the benefits and drawbacks of the use of ontologies to implement the adaptive core. Ontologies were rejected because of their resource consumption.
- Study of the benefits and drawbacks of the use of genetic algorithms to implement the adaptive core. They were rejected because of the effort in the definition of the fitness functions and the time taken to obtain a solution.
- Definition of ontologies domain for content aggregation system, web performance, web architecture and user behaviour.
- Definition and study of crossover and mutation operators for genetic algorithm to find an optimized fragment design.
- Definition of a general method to create content page models to be used in a emulation phase to obtain data about the relationship among the characterization parameters of the content elements and the best assembly points of each aggregation relationship.
- Definition of the data instances representations to be used as training data sets.
- Selection of the data mining algorithm based on the instances classification times and the coverage of the resulting structures.

5

COFRADIAS: COntent FRagment ADaptation In web Aggregation Systems

Cofradía *f. Gremio, compañía o unión de gentes para un fin determinado. (Guild, company or union of people for a particular purpose.)*
f. ant. Vecindario, unión de personas o pueblos congregados entre sí para participar de ciertos privilegios. (Neighborhood, union of people or congregated towns to partake of certain privileges.)
—Real Academia Española—

Content Fragment *Assembly of content elements for a performance improvement purpose.*

COFRADIAS framework is our proposal for a general framework which includes an adaptive core system in the basic scheme of a content aggregation system. COFRADIAS framework is not only the definition of the interfaces between the new adaptive core and the tiers of a traditional CAS system. It is also the design solution taken in order to adapt the tiers of a CAS system to the new type of elements: the content fragments.

In the first part of this chapter, we give details about the design of the framework and general guidelines to implement the COFRADIAS framework. We have also created a real implementation of the framework. We have developed an extension for Drupal CMS (Content Management System) in order to manage aggregation of contents and to integrate it with our adaptive core system. The details are explained in the second part of this chapter.

5.1 Introduction

In previous chapters, we have presented the problems of using web caching in CAS systems. In order to improve the performance, fragments of the pages are created and, in consequence, the hit ratios of the web cache are increased. But the process of assembling these fragments adds overhead times to the latency of the CAS system.

Therefore, we have proposed a technique to define content fragment designs that balance the hit ratio of the cache with the overhead of the assembling process. In the previous chapter (Chapter 4), we have explained the details of our technique to decide the content fragment design. This decision is based on a classification algorithm extracted from off-line data mining process. Thus, the content fragment design is adapted as the ODGex model changes.

We have obtained two main deliverables or contributions from a technological point of view. The first one refers to the definition of a general architecture of a CAS system where our core tool is integrated. The guidelines to adapt an existing CAS system to our solution are defined. These guidelines detail the data to be gathered from the CAS system, the new interfaces with the adaptive core, the changes in the existing interfaces, and the updates in the application tier. We have called COFRADIAS to the general framework that adapts the fragment designs of the web pages.

The second contribution is a proof of concept (POC) of the general framework. This POC is an extension of Drupal CMS to manage content aggregation web pages and to integrate our adaptive tool in Drupal [26]. This POC is used in the experimental phase of this Ph.D. dissertation.

5.2 Architecture of COFRADIAS framework

The architecture of COFRADIAS framework is based on the general layout of a four-tiers web content aggregation system (Figure 2.2 and 3.2). The details about the creation of the core of COFRADIAS framework have been explained in the previous chapter (Chapter 4), but this section is about the design decisions taken in our framework. These design decisions are related to the interfaces among CAS tiers and the module in charge of creating the fragmentation design. Changes in other interfaces between tiers of the CAS system are also explained. These changes are done in order to allow the management of content fragments in addition to content elements and web pages.

There are two new interfaces in COFRADIAS framework (the *Gathering Service* and *Fragment Design Service* modules) and one of the interfaces in the traditional scheme needs to be adapted to the new concept of content fragment (the *Fragment Manager* module). All these interfaces use some sort of XML structure to interchange data. The modules in charge of implementing and interacting within these interfaces are shown in Figure 5.1.

On the one hand, the new interfaces are placed between the CAS system and the adaptive core. The adaptive core is the responsible of creating the adaptive fragment design by classifying the assembly point for each aggregation relationship. On the other hand, the interface which needs to be redefined is the corresponding to the communication between the CAS system and the cache proxy, because a new element type needs to be managed and identified: the content fragment.

The output of the adaptive core is based on the values of the characterization parameters of the content elements. These values are the inputs of the system and are gathered from different tiers of the web systems. Some of the parameters are measured from the content elements (sizes and structure),

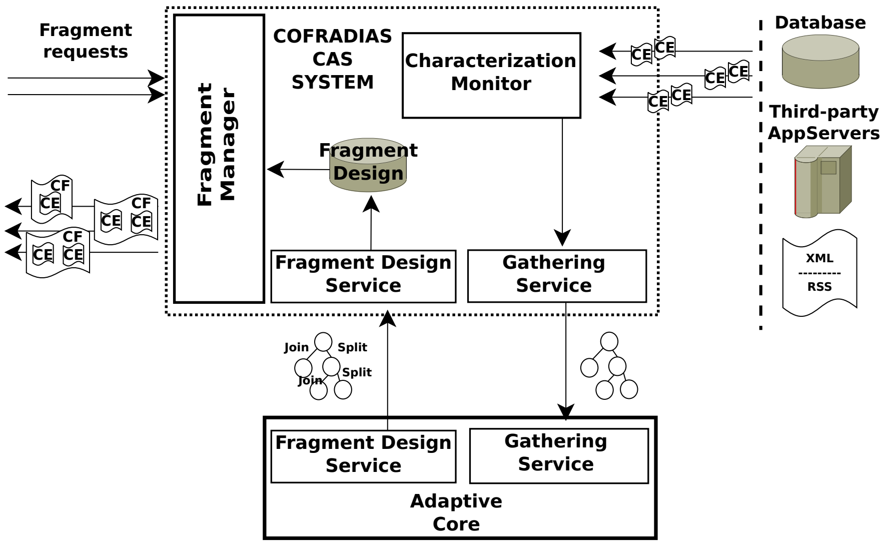


Fig. 5.1. Architecture of the adaptive content fragment framework.

other ones from the web application system or the local database (request rates) or from both sources (update rates). Modern web systems usually store this sort of performance metrics [82, 18], so there is not any drawback to gather these data.

Just some low workload processes need to be implemented to summarize the already gathered data and to transmit them to the adaptive core system. In COFRADIAS framework (Figure 5.1), the *Characterization Monitor* module is the new module in charge of gathering and summarizing all these performance data. This module is the data source for the *Gathering Service* module, which sends these data to the adaptive core.

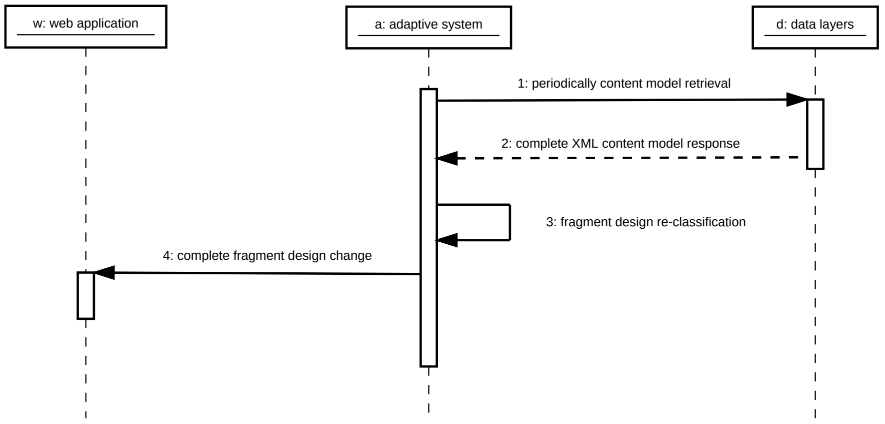
In the same way, the result of the classification process —the states of the edges— needs to be transmitted to the application server in order to change the content fragment design. The module in charge of transmitting the ODGex is the *Fragment Design Service*. The *Fragment Design* module is in charge of storing the fragments defined by the ODGex. Obviously, the application server should be enabled to adapt its content fragment design and to interact with our adaptive system. The *Fragment Manager* module uses the data in the *Fragment Design* module and it adapts the fragments of the content of the CAS.

5.2.1 Communication among COFRADIAS modules

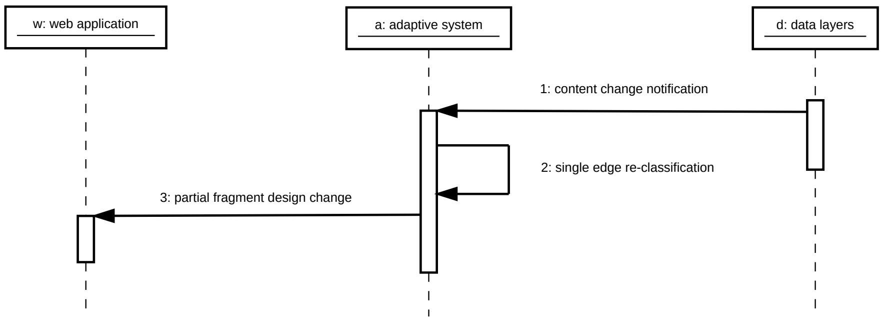
The communication and transmission between the adaptive core and the CAS can be done by using Web Services. These Web Services can be implemented by using SOAP (Simple Object Access Protocol) or REST (Representational

State Transfer) based communication, but the last one is implemented easier and consumes less resources [1].

Even though these communication processes and services executions have low overheads, the communication is not established regularly. The data can be interchanged in periods of time (*periodical mode*) or when changes occur in the content elements (*event-driven mode*).


w: web application
a: adaptive system
d: data layers
1: periodically content model retrieval
2: complete XML content model response
3: fragment design re-classification
4: complete fragment design change

(a) Periodical mode


w: web application
a: adaptive system
d: data layers
1: content change notification
2: single edge re-classification
3: partial fragment design change

(b) Event-driven mode

Fig. 5.2. Sequence diagrams for the communication between the content aggregation system tiers and the adaptive system.

The adaptive system behaves as the client component in the *periodical mode*. The process is summarized in Figure 5.2(a). In this mode, the adaptive system core requests periodically, in fixed intervals of time, the ODGex model —the extended content aggregation model— of the content elements within the characterization parameters values. REST requests are sent from the adaptive system (client) to the CAS tiers (servers). Once the core has these data, it re-classifies all the edges of the ODGex model and, finally, it sends the new content fragment design, the states of the edges, to the application server. A REST request is also used in this case to send the ODGex model, and its edge states, from the adaptive system (client) to the web application (server). In this mode, the adaptive system always behaves as the client element.

In the case of *event-driven* mode, Figure 5.2(b), a REST request notification is sent between the CAS tiers (client) and the adaptive system (server) every time a content element changes, instead of requesting the ODGex model in fixed periods of time. This event is followed by a re-classification of only the involved aggregation relationships. Finally, only the affected edges, the updated ones, are sent to the web application (server) by the adaptive system (client). In this mode, the adaptive system behaves alternatively as a client and as a server.

The advantage of the *event-driven mode* is that the size of the data transmitted among the elements of the architecture is significantly smaller than in the *periodical mode*. In this last case, all the ODGex structure is sent in each communication process. On the contrary, the number of communication events is bigger for the *event-driven mode*.

5.2.2 Format of the interchanged data

The last issue about the interfaces between the elements or the architecture is the representation of the data interchanged among them. The interchanged data are content structure and content characteristics represented by an ODGex model (Section 3.7). As the usual way to interchange data between web services is the use of structure data in XML files, we need to represent the ODGex model in some standard XML language.

The survey of Rodriguez [75] states that the best XML schemas to represent graphs are Heidi, GraphML and XGMML. He makes a comparison of the three schemas based on their expressibility, extendability, simplicity and robustness. He concludes that GraphML has the best skills in these four aspects. For us, the XML schema is only a tool so we do not need to make a deeper analysis among the alternatives. GraphML can be extended in order to represent the characterization parameters and the edges states, so it is a good choice for us.

GraphML allows us to define additional attributes to the vertex and to the edges. We define the state of the aggregation relationship using an edge attribute, and the characterization parameters as vertex attributes. Listing 5.1 shows an example of two related nodes. The example corresponds to three vertexes: one father vertex with two children.

The number of children and fathers of a vertex is easily calculated by the analysis of a DAG. We only need to count the incoming and outgoing edges. Despite this, we have included two attributes (`d4` and `d5` in the example) to represent this information in the XML structure. This is needed in the case of *event-driven mode*. In this mode, only a partial ODGex structure is

Listing 5.1. Example of GraphML that represents our ODGex model.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
 http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key id="d0" for="node" attr.name="req.rate"
                          attr.type="double"/>
  <key id="d1" for="node" attr.name="upd.rate"
                          attr.type="double"/>
  <key id="d2" for="node" attr.name="size"
                          attr.type="double"/>
  <key id="d3" for="node" attr.name="n.parents"
                          attr.type="integer"/>
  <key id="d4" for="node" attr.name="n.childs"
                          attr.type="integer"/>
  <key id="d5" for="edge" attr.name="join"
                          attr.type="integer"/>
  <graph id="feedmaster" edgedefault="directed">
    <node id="4">
      <data key="d0">0</data>
      <data key="d1">2</data>
      <data key="d2">32512</data>
    </node>
    <node id="5">
      <data key="d0">0</data>
      <data key="d1">2</data>
      <data key="d2">3974</data>
    </node>
    <node id="6">
      <data key="d0">0</data>
      <data key="d1">2</data>
      <data key="d2">12345</data>
    </node>
    <edge id="0" source="4" target="5">
      <data key="d5">1</data>
    </edge>
    <edge id="1" source="4" target="6">
      <data key="d5">0</data>
    </edge>
  </graph>
</graphml>
```

sent between CAS tiers and the adaptive core, so not all of the incoming and outgoing edges are included in this partial XML file. The other three parameters (request rate, update rate and content size) are always explicitly represented by using `d0`, `d1` and `d3` attributes. The names of the attributes are defined arbitrarily.

The structure of the graph is represented by defining each graph vertex with the tag `node` and, after that, each edge with the tag `edge`. In the example in Listing 5.1, the states are represented as integer values 0 and 1.

The two interfaces of COFRADIAS, *Fragment Design Service* and *Gathering Service*, do not need all the data in the XML file. The *Fragment Design Service* module only needs the data related to the edges and their states (`edge` tags). The *Gathering Service* module only needs the data related to the vertexes and their characterization values (`node` tags).

5.2.3 Content aggregation systems modifications

Once we have explained the details of the two new interfaces, we are going to explain how the interface between the web cache proxy and the application server has been modified. This interface is called *Fragment Manager* in COFRADIAS framework.

Traditionally, the interaction between proxy caches and web servers has been done using HTTP requests and responses. In the case of content aggregation systems (CAS), this remains valid. The only difference between both cases is that, in the last one, there are content element requests besides web page requests.

In CAS systems, the interaction between the proxy and the application server is based on HTTP and HTML. The web cache requests elements (pages, templates or content elements) by using HTTP requests. The application server also responds by using the HTTP protocol. The application server includes data inside the HTML code of the HTTP response. This new code informs to the cache proxy about which other content elements should be requested to create the whole web page. These data are codified using ESI (Edge Side Includes) tags. Every time a HTTP response arrives to the cache proxy, it parses the HTML code to find ESI tags. These tags are translated into the proper action. This proper action is, in most of the cases, a new HTTP request to the application server. The HTML of the server response replaces the ESI tag. Previously to this replacement, the HTML code of the response has been also parsed in order to find new ESI tags.

Figure 5.3 shows the process explained in the previous paragraph. Each of the content elements is identified by a unique identifier. This identifier is added to the include ESI tag within an URL. The web server uses the pair URL and identifier to select the content element to be retrieved. In our proposed system, the difference in the interaction between the cache and the web server is that the cache requests content fragments instead of content elements. A content fragment is a set of one or more content elements which are pre-assembled in

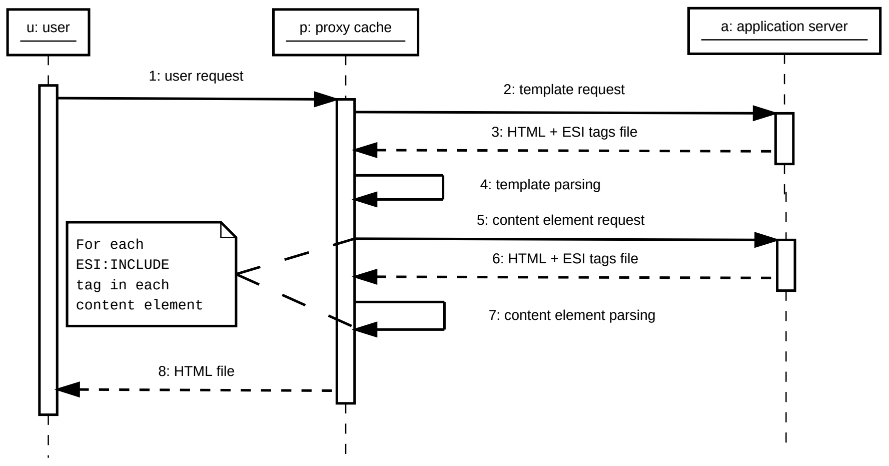


Fig. 5.3. Sequence diagram for an ESI-enabled proxy cache.

the application server. In COFRADIAS, the interaction also takes place using HTTP requests and responses, HTML code and ESI tags. The only problem to solve is how to identify these content fragments.

Content fragments are defined as a connected sub-graph with only one source vertex, the representative one (Section 3.7). If we had created a new identifier in order to request all the different content fragments, the management of these identifiers would be complex. Instead of creating new identifiers, we use the representative vertex for each content fragment. We can observe an example in Figure 5.4 where the identifiers of the content elements and of the content fragments are shown.

When the cache requests to the application server a content fragment by using the identifier of the representative vertex, the server retrieves the content of the corresponding content element and all its children. This process is done by the *Fragment Manager* module. The module retrieves and assembles the content for each child in *join* state. The information about the states of the aggregation relationships is obtained from the *Fragment Design* module. This process is recursively repeated for each child until all the content elements of the content fragment are pre-assembled.

For example, in Figure 5.4, if the content fragment CF2 is requested, its content and its children (CE3 and CE4) are retrieved. After that, the contents of CE3 and CE4 are retrieved and assembled with the previous content. Finally, the child of content element CE6 is retrieved and assembled. In this way, it is easy to manage the identifiers of the content fragments and to create the content fragments in the web server.

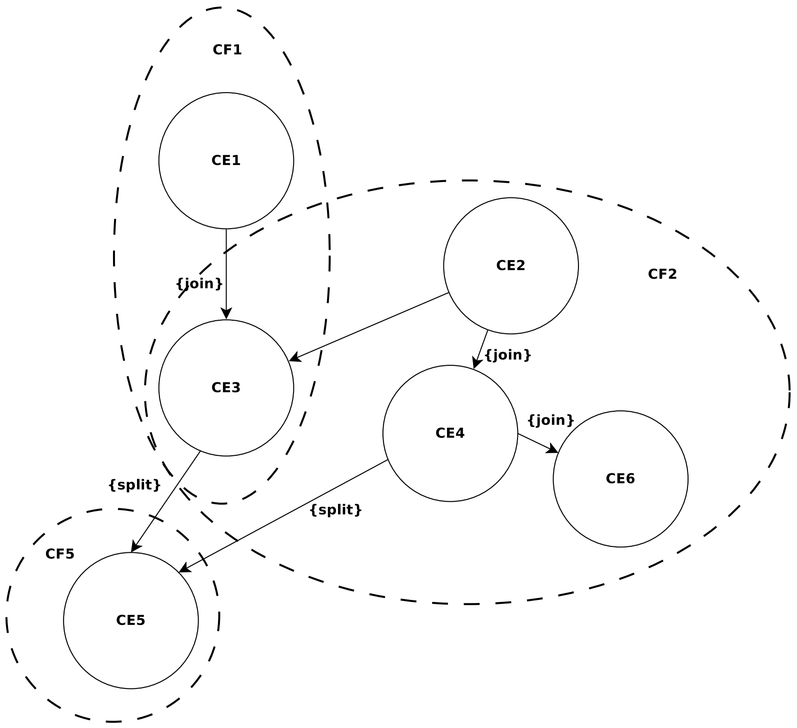


Fig. 5.4. Example of fragment element identification.

5.3 Real system implementation: DRUPAL-COFRADIAS extension

Besides the definition of a general architecture of a content fragment adaptive aggregation system, we have adapted a real and commercial system in order to validate the proposed architecture. This implementation has been also used in the experimental phase. Results obtained from a wide-used tool are always more reliable and they are extrapolated more easily than in the case of using a tool developed for a specific experimental process.

There are several popular content aggregation web sites. Some of them are only content aggregators, and others are also applications, services or widgets aggregators. All of them are considered as personal web portals, start pages, customized dashboard publishing platform or customized content feeds aggregators.

Other types of web applications also behave as a content aggregator system, but without all the features of a CAS system, for example, news web sites. In this type of applications, the news and their contents are obtained from remote providers and they are published in several pages of the web site. Thus, these systems share the same content in different web pages. The main difference is the customization of the web pages. News web sites do not usually allow users to set up their own web pages. Anyway, content management systems (CMS) have similar features to CAS systems.

In order to integrate our proposed system (COFRADIAS) in a real web system, we need to study current CAS systems in order to select one of them. Most popular web content aggregation systems are *iGoogle*, *Pageflakes*, *Netvibes* and *My Yahoo!* [64]. These web systems are only available on-line and they cannot be extended in any way.

Another important requirement is to isolate the system from other workloads in order to create the same conditions among the experiment executions. We need to select a tool which can be deployed in our own servers in order to control the environment. Finally, we also need to modify the web application in order to allow it to manage content fragments, to pre-assemble content elements and to interact with the adaptive core. Therefore, we need to choose a tool whose source code is available.

Any of the most usual CAS web sites allows users to download the web application and, in consequence, we are not either allowed to modify or extend the tool and its source code. From our own knowledge, there is not any CAS system available as a downloadable and installable system. CMS are the most similar tools to CAS systems. Furthermore, there are a lot of distributions of CMS which can be downloaded and installed. The most common open source ones are Joomla, Drupal and Wordpress [77].

The criteria, to select one of the three CMS, are based on the next issues:

- Extendibility. This feature considers how easy it is to develop new modules or functionality in the system.
- Availability of user and programming documentation. An important feature is the documentation available on-line or in references. This is important for the programming process.
- User friendliness. How easy it is, from the user point of view, the way in which the system is managed and the contents created.

After comparing the tree CMS, we conclude that Wordpress is the most user friendly. The problem with Wordpress is its extendibility. On the contrary, Joomla is the most extendible, but it is clearly the less user friendly. Finally, Drupal has more documentation available than the other two. Its level of friendliness is close to the Worpress one and, from far, it is easier than Joomla. Thus, from our point of view, Drupal is the best balanced among the three criteria.

The necessary work to adapt Drupal to COFRADIAS is divided into four main tasks, and they are related to each of the new modules of COFRADIAS framework:

(1) *Aggregation Manager Module.* To extend Drupal in order to be able to create web pages from aggregation of remote content. This task involves to adapt Drupal to behave as a CAS instead of a CMS.

(2) *Fragment Manager Module*. To allow Drupal to manage content fragments and to create them using the content fragment designs. This task is related to the implementation of the *Fragment Manager* and *Fragment Design* modules.
(3) *Characterization Manager Module*. To allow Drupal to gather data about the characterization parameters of the content elements. This task involves the implementation of the *Characterization Monitor* module.
(4) *Adaptive Core Services Module*. To create several interfaces in order to send to the adaptive core the ODGex model and in order to receive the content fragment designs from it. This task is related to the implementation of the *Fragment Design Service* and *Gathering Service* modules.

We have developed Drupal modules —module is the common name for extensions of the CMS— in order to cover all the new requirements of the tool. Figure 5.5 shows the typical Drupal architecture and we have highlighted the modules and changes we have developed. The modules were developed for Drupal version 6.20. The files of the modules are organized in folders with the name of the module (`/<moduleName>/`). There are three main files:

- `<moduleName>.info`, the textual description of the module.
- `<moduleName>.install`, the tasks that are executed when the module is installed. The creation of the *Fragment Design* database is done by this file when the *Fragment Manager* module is installed.
- `<moduleName>.module`, the source code for the implementation of all the functions of the module.

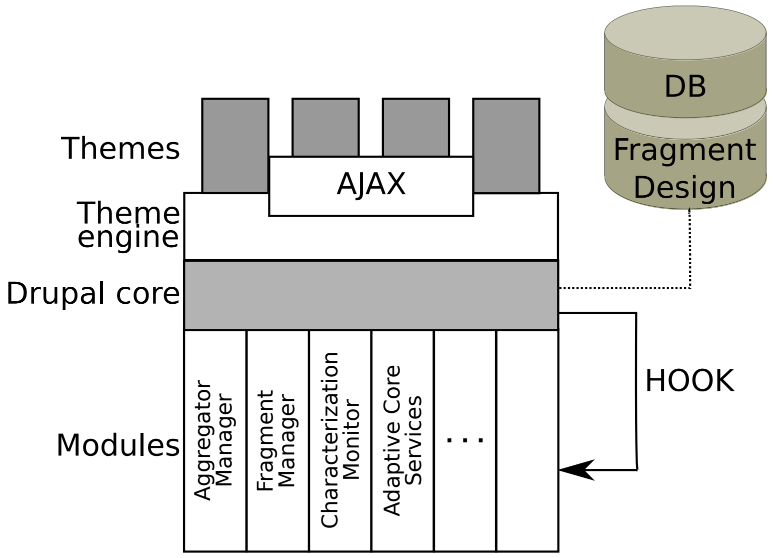


Fig. 5.5. Architecture of Drupal with the modules for COFRADIAS.

Once the first task is implemented, the Drupal administrator is able to manage remote content sources. Only this administrator role is allowed to add new content sources. The other users are allowed to create unlimited aggregation designs by defining which content sources are included and their structures, *i.e.*, the users create their own DAG of content aggregations. Changes in the database of Drupal have been done in order to store the data of the content sources and of the aggregation structures.

The second task involves the analysis of the user requests that arrive to Drupal. If the request corresponds to one of the new page types, content aggregation pages, the structure of the content aggregations needs to be retrieved. The content aggregations need to be sent back to the proxy cache within the content fragments defined in the content fragment design. Therefore, the database does not only need to store data about the aggregation structure, it also needs to store the content fragment design by using the states of the aggregation relationships (edges). All these requirements are defined in the *Fragment Design* module of COFRADIAS framework. Thus, Drupal has to manage three different types of requests: (a) Requests for traditional CMS pages; (b) Requests for templates of content aggregation pages; (c) Requests for content fragment elements.

In order to include the web page design information in the web pages, ESI tags have been included in the web content. A new tool script has been created to answer the content fragment requests. The script recursively generates the content fragments by retrieving all the content elements related by edges in state *join*. This script corresponds to the implementation, in Drupal, of the *Fragment Manager* module of COFRADIAS framework.

The third task involves gathering the values of the size, the request rate, the update rate, and the content structure of the content elements. This corresponds to the *Characterization Monitor* module of COFRADIAS framework. On the one hand, some of the parameters (size, request rate and content structure) are gathered directly from Drupal databases, the proxy cache and from the own content elements. On the other hand, we do not have access to the logs of the remote content services. The information about the update rates is in these logs. Therefore, we need to approximate the update rates. We have two ways to do that:

(a) Using the HTTP headers of the documents received from the remote content services. Sometimes, the HTTP response headers have information about the invalidation time of the contents (`Expires`) or their last updates (`Last-Modified`). We may calculate the update rate by using the values of these parameters among different content response.
(b) When the values of these parameters are missed in the headers, we can only compare the content among different requests over the same services and detect changes among the content of these requests.

Finally, the interactions between the tiers of Drupal and the adaptive core are done by the interchange of XML files. These XML files use GraphML to represent the ODGex model. We have implemented both communication methods: *event-driven* and *periodical mode*. For the *periodical* mode, two REST web services have been deployed: one to request the ODGex model from Drupal —*Gathering Service* module— and the other one to send the fragmentation design to the application tier of Drupal —*Fragment Design Service* module—. In the case of *event-driven mode*, a request to the adaptive core is sent every time the processes, in charge of gathering the characterization parameters, detect a change in the contents.

For both cases, the integration of these REST web services in Drupal have been implemented as scripts which are totally independent to the Drupal modules. These new scripts only need to interact with the data tier of Drupal, and that could be done by using the Drupal data libraries.

The implementation of DRUPAL-COFRADIAS extension has been published in the repository of Drupal modules. It is available for the community in order to deploy it, change its source code or re-use portions of it [26].

This description of the Drupal extension is only an overview since the details are not interesting for this dissertation. Although, it is very important and remarkable that the contributions of the Ph.D. research have been implemented in a commercial tool and are available to the community.

5.4 Summary

In this chapter we have defined the general guidelines to integrate our adaptive system in CAS architectures. COFRADIAS is the name for the framework created for the integration of our adaptive core tool in a CAS system. The guidelines define the changes in the interfaces among tiers of the CAS architecture, the changes in the application tier, and the changes in the interfaces between the CAS architecture and the adaptive core.

Finally, a proof of concept (a concrete development) of COFRADIAS framework has been explained. Just some general details have been explained about the deployment of the tool because deep details of the process are not interesting from a research and innovation point of view. The proof of concept has been built as a module for Drupal CMS in order to take profit of all the available tools and modules. The interfaces have been deployed by using web services. The ODGex models have been expressed in XML format by using GraphML. We have also changed some parts of the application tier in order to allow Drupal to manage a new type of elements: content fragments.

To summarize, the main contributions explained in this chapter are two:

- Definition of a general framework to integrate a technique to improve the performance of CAS systems based on the adaptation of content fragment. The name of the framework is COFRADIAS.
- Development of a proof of concept of the COFRADIAS framework by using Drupal, a commercial content management system.

Part III

Validation

6
Experimentation design

Insanity is doing the same thing, over and over again, but expecting different results.
—Albert Einstein—

The validation of our work has been done by the execution of experiments on a test-bed. These experiments have been changed in different ways in order to cover several cases and situations. In this chapter we describe the features that define an experiment, and how the values of these features have been changed.

6.1 Introduction

We have experimented with a real web application to validate the contributions of this dissertation. The real web system is developed as an extension of Drupal (Section 5.3). This real system is part of the deployment of a test-bed environment for the emulation of the experiments.

To design the experiments, we have firstly identified the features which define them. And secondly, we have defined the variation of these features to create different cases. These cases are defined to cover different situations to validate our contributions.

The experimentation is not only the definition of the features that can vary, but also the metrics we want to measure from the system. These metrics are related to the performance and the system resource consumption.

Finally, it is important to define the execution patterns of the experiments. In order to obtain more reliable values, the experiments should be repeated several times. The size of the experiment should be big enough to get reliable results.

6.2 Validation of the research contributions

All the experiment phases are addressed to validate the contributions of our research work (Part II). We summarize, in this section, the issues to be validated by experimentation:

- *Validate the use of data mining and decision trees.* COFRADIAS framework is based on an adaptive core which classifies the aggregation relationship by assigning a state (or assembly point). This core has been implemented with decision trees obtained from a data mining process. Suitable experiments need to be done to validate the improvement obtained with this approach.
  Firstly, the improvement of the system needs to be compared with the two traditional scenarios. These scenarios correspond to the content elements that are assembled in the proxy cache or in the web server. And secondly, the results should be compared with other approach. In our case, the research work done by Hassan in [50, 48]. Hassan presented a solution called MACE to improve the performance of the cache in CAS systems. We have used the algorithm suggested in MACE to implement the adaptive core of the COFRADIAS framework. We have compared the results obtained with the adaptive core based on decision trees with those obtained with the MACE implementation.
- *Validate the process of creation of the training data set.* We have proposed a method to create synthetic training data sets. Data mining techniques are applied over these sets in order to obtain a knowledge representation based on decision trees. The creation process of the data sets defines the way that the data should be obtained from the real system. The validation of this process needs to be done by studying the performance results obtained from architectures with different configurations. Therefore, several mining processes are done over the synthetic data created in different architectures or hardware conditions.
- *Validate the representation of the data instances and the independent attributes.* Several data representations (entire, ratio, difference and distance) and several independent attributes (update rates, request rates, size, number of aggregators, and number of aggregations) have been proposed to be the independent attributes. We need to study their validity and to find which ones are more suitable to be used. This objective can be achieved by comparing the performance results of the CAS system using different decision trees to deploy the adaptive core. These decision trees are generated from different training data sets in which the independent attributes have been represented in different ways and, in some of the cases, some of the attributes have been removed.
- *Validate the low overhead of the system.* One of the most important requirements of the solution is the necessity of generating a low overhead. This requirement has affected to the entire definition of COFRADIAS framework, and the implementation of the adaptive core of COFRADIAS. We need to study if our proposal generates, or not, a low overhead over the system. We have studied the performance that our solution generates, in comparison with the overhead of MACE approach [50, 48], and with traditional scenarios.

6.3 Definition of the experiments

We have identified the features that define an experiment and we have varied the setup of the experiments by changing these features. We have considered five different features: (a) the CAS hardware architecture; (b) the algorithm to define the content fragment designs; (c) the content page model; (d) the user behaviour and load activity; and (e) the changes in the values of the independent attributes. In the next subsections we describe each experiment feature.

6.3.1 Hardware architecture

Alterations in the architecture of the CAS system generate different latencies. These changes are not covered by the training data sets. Every time a change in the elements of the architecture (hardware, software, operating system, architecture, network elements, setups, etc.) occurs, we need to train again the adaptive core, by generating new training data sets, doing a new data mining process, and generating new decision trees. Therefore, the first feature that defines an experiment is the architecture of the CAS system.

In particular, we have experimented with two hardware architectures. The computers and servers used in each of them are different. The tiers of the architecture and their configurations are also different. In one of them, the proxy cache server and the web application server are placed in different computers. We called this as *distributed* hardware architecture. In the other case, we use the same computer to deploy the two tiers and we called it as *centralized* hardware architecture. The hardware characteristics of each computer are indicated in Table 6.1 and 6.2.

The versions of the operating system, server software and applications are the same in all the cases and computers. The operating system is *Ubuntu Server 10.04 LTS*[1]. The web server applications are *Apache 2.2*[2] and *PHP 5.2*[3]. The database management system is *MySQL 5.1*[4]. The proxy cache is *Oracle Application Server Web Cache 10g*[5].

6.3.2 Algorithm for the creation of the content fragment designs

Several alternatives for the deployment of the adaptive core have been taken into account. Three groups are differentiated: solutions based on a traditional scheme (assembly points in the proxy cache or in the web application server); solutions based on the use of one of the 24 proposed decision trees; and another

[1] `http://releases.ubuntu.com/lucid/`
[2] `http://httpd.apache.org/docs/2.2/`
[3] `http://php.net/releases/5_2_0.php`
[4] `http://dev.mysql.com/doc/refman/5.1/en/`
[5] `www.oracle.com/technetwork/middleware/ias/overview/index.html`

Table 6.1. Hardware characteristics of the *centralized* hardware architecture.

Cache proxy and Application server
Dell Precision T3400 Intel(R) Core(TM)2 Duo E6850 3.00GHz DDR2 SDRAM 800MHz Dual Symetric 2.0GB Serial ATA 250GB 7200RPM 3.0GB/s

Table 6.2. Hardware characteristics of the *distributed* hardware architecture.

Cache proxy
HP Proliant ML530 G2 Dual Processor PIII XEON 2.8 GHz 2.0GB DDR SDRAM PC1600-MHz Ultra-Wide SCSI 146GB 7200RPM 40MB/s

Application server
HP DC5700 Microtower E4300 Intel(R) Core(TM)2 Duo E4300 1.80GHz 2.0GB DDR2 SDRAM 667MHz Dual Symetric 2.0GB Serial ATA 250GB 7200RPM 3.0GB/s

approach, MACE. The identification of the experiments, for each algorithm deployment, is done as follows:

1. *split*, for the traditional scheme in which the content elements are assembled in the proxy cache.
2. *join*, for the traditional scheme in which the content elements are assembled in the application server.
3. *[relationPattern]/[removedAttribute]*, in the case in which decision trees are used to implement the adaptive core. The *[relationPattern]* and the *[removedAttribute]* are the indexes of the correspondent decision tree names.
4. *mace*, for the deployment of the adaptive core by using the approach of Hassan [50, 48].

6.3.3 Content page models

The validity of the framework and the implementation of the adaptive core need to be studied in different CAS system types. Thus, different experiments with different content page models (ODGex models) have been done. The details of the content page models used in the experiments are given in the next paragraphs. They have been based on different content aggregation system types, and they have been extracted from real web sites.

Content page models are created from real systems because the contents are public and available for general users in most of the cases. By the creation of a crawler, the content, its structure, and some of its characteristics are harvested from web sites. We created an extension of *JMeter* [46] to extract the content models from the web sites.

Our *JMeter* extension downloads a list of web pages from a web site. The content of these web pages is downloaded at different points in time. The structure of the contents —detection of content elements, size of the content elements, aggregators and aggregations of content elements— are obtained by the analysis of the downloaded web pages. The analysis from different downloads could give us some clues about the update rates of the contents. We have used this tool to obtain the content page model (ODGex) which has been used in the experiments.

The first content page model was harvested from a news web site. News sites are considered as an especial case of content management systems (CMS), and these systems have a lot of similarities with CAS systems [23]. We selected the page of *The New York Times* [80]. The contents of all the web pages of the site were crawled during a week in March 2010. The obtained ODGex model had 3082 vertexes —source content elements (web pages) and other content elements (news)—. The news were aggregated in 482 different web pages, *i.e.*, 482 source vertexes in the ODGex. Finally, the relationships or edges of the model (aggregations) between content elements were 13979. We called this ODGex model as *nytimes*.

The second content page model was based on a personal dashboard or personal web portal. The most representative web sites, of this type, are *iGoogle*, *NetVibes*, *PageFlakes*, *myMessenger* and *Yahoo! Pipes*. The one with a higher number of users is *iGoogle*. The problem of this one is that the personal web pages are not public for other users. Thus, the web page of *Pageflakes* [66] was selected. The crawling of *Pageflakes* was done in October 2010. Instead of crawling the new web pages (pagecasts) of a period of time, we considered more interesting to crawl the 2000 web pages which were the most popular at the time the crawling was done. These pages aggregated 14803 content elements (widgets or flakes) using 24771 aggregation relationships. We called this ODGex model as *pageflakes*.

The third content page model was also obtained from a real personal web portal. In this case, it corresponds to *Yahoo! Pipes* [88]. This content page model has been used to compare the results with other solution to our problem (MACE). We called *yahoopipes* to this ODGex model.

This last page model was provided by the authors of MACE in a XML file format. We do not have the details about how the model was created. In order to work with this new model, we translated from the format of the XML into the one used in our system. This new web page model has 2000 web pages, 9182 content elements and 14000 aggregation relationships. The details of the content web pages are also included in Appendix B.

6.3.4 User behaviour

Content models are crawled from real web sites, but the behaviour of the users is not available. In researches not directly related to this thesis [30], we have proposed the public and open used of the data related to the behaviour of the users. This is achieved by creating centralized storages services, public accessible with APIs and web services. Simple extensions for the browsers are developed in order to monitor and store, in the centralized system, the behaviour of the users. The problem with this type of proposals is to convince the users to be *monitored*. Thus, we could not take profit of this tool.

Since user behaviour data were not available, we created a model based on statistical studies. The update rate and the browsing patterns were generated by using suitable statistical distributions due to it is not possible to determine the frequencies which the contents are updated or requested with. These distributions are based on wide accepted research works [31, 15, 45, 23, 6, 5].

A user behaviour model is defined by the popularity of the files (objects) and by the user session activity. Popularity is the request probability of a page. The user session activity shows the number of pages a user requests during a web session, the inter-request time –the time between two requests in the same session– and the interarrival time –the time between the start of two user sessions or user arrivals–.

Mostly of the researches of actual web sites agree in the distributions which model the user session activity parameters. In modern web systems, user interarrival times and inter-request times are modelled using exponential distributions [31, 81]. The session length or number of requests in a session is modelled by a Zipf distribution with parameter $\alpha = 1.765$ [6, 5]. The values of the parameters for the exponential distributions depend on the user load level which the system needs to be loaded. Thus, these two values are particular for each experiment execution of the scenarios. The popularity for each web page is modelled by a Zipf distribution with $\alpha = 0.56$ [31].

After the distributions selection, it is possible to create traces files with the information about which user actions should be done in a given point in time. Two traces files are used in each experiment, one for content requests and another one for update requests. Content request traces file has only the actions which correspond to user read requests. Update traces file has several types of requests: modifications of the content elements (size or aggregation relationships) and modifications of the page popularity. To create a more dynamic behaviour of the users, the popularity assigned to each page has been changed randomly over time. Thus, the request and update rates changed over time.

Using an update traces file, all the characterization parameters of the ODG model change throughout the experiment, and therefore, the effect of these changes (transient periods) on the performance results can be analysed. The use of traces files to model the user behaviour also allows us to repeat the same behaviour in different experiment executions and to compare the results among them.

The first parameter to define, in order to generate the traces file, is the total number of web pages in the ODGex model and the identifiers to request each of them. The second parameter is the size of the traces file, *i.e.*, the number of requests in the file. The parameters of the statistical distributions have always the same values. A time value is assigned to each request in the traces file. If we need to modify the update and request rates, we only need to scale this basic unit of time. We identified the different traces files, or user behaviour models, using the values of the update and request rate with a rate unit. For example, *0.5/0.01/*s^{-1} corresponds to an experiment with a request rate of 0.5 req./s and an update rate of 0.01 req./s.

6.3.5 Changes in the values of the independent attributes

If the values of the independent attributes of two content elements change, the output of the classification algorithm also changes. Changes in the independent attributes cause the change of the state of the edge in the ODGex. This results in new fragment designs.

Changes in the fragment design generate *small* and partial transient phases in the web cache. The old cached fragments are not requested again —they are evicted when their expiration times are reached—, and the new fragments are stored in the web cache only after they are requested.

We are interested in creating some experiments in which these partial transient phases are omitted. We have called this feature as *dynamism.* Therefore, we have defined experiments in which do not occur changes in the values of the independent attributes, which we call as *static* experiments, and other ones in which the values are continually updated, they are called as *dynamic* experiments.

It is necessary to explain that the *dynamism* of the experiments does not have any relation with the update rate of the content elements. Updates in the contents are also done in a *static* scenario. These updates only change the content, but not the values of the characterization parameters. In consequence, these contents have to be invalidated in the cache, but the fragment design does not have to be updated because the independent attributes have not been changed. In the case of a *static* scenario, the update traces file only contains requests corresponding to updates of the content elements. In case of a *dynamic* scenario, the update traces file contains all the types of update requests explained in the previous subsection.

Besides the problem of no generation of partial transient phases, the *static* scenarios do not reflect the behaviour of real environments because they only classify the aggregation relationship once at the beginning of the experiments. If the values of the characterization parameters remain constant, the classification results will also remain constant and, in consequence, only one classification process is needed.

Although *static* experiments do not reflect a real behaviour of the system, they are very interesting for us. They are used in the preliminary experiments to study the benefits of our solution isolated from their main drawbacks: overhead and transient states.

In all the *dynamic* experiments that we have designed, the adaptation of the fragment designs are done in *event-driven* mode, *i.e.*, every time that a characterization parameter occurs, the classification process of the involved edges are executed.

6.3.6 Experiments identification

The experiments are determined by the parameters and features described in the previous subsections. We have identified the experiments using a 5-tuple: `<[arquitecture]`, `[algorithm]`, `[pageModel]`, `[userModel]`, `[dynamism]>`. This identifier gives enough information to know the values of the features of an experiment.

An example of identifier is `<distributed`, `distance/size`, `nytimes`, `0.1/0.001/s`$^{-1}$, `static>` in which: the servers are deployed in different computers; the optimization algorithm is a decision tree with the attributes expressed as a distance values and the size attribute is not used; the ODGex model is based on the data extracted from the web site of the *The New York Times*; the request rate is 0.1 req./s and the update rate is 0.001 req./s; and, finally, changes in the characterization parameters values do not occur.

6.4 Design of the experiments

The experiments have been distributed in experiment sets. These sets are created in order to study some goal by comparing the behaviour of different experiments. We have designed five experiment sets. Each experiment execution is defined by the features of the experiments. Table 6.3 is a summary of all the values of the experiment features for our particular experiment design. The rest of the section is devoted to explain the goal of each experiment set and the experiments which are part of the set.

The goal of the experiment set 1 is to compare the traditional use of the CAS system with the use of the adaptive core based on decision trees. As traditional use, we mean the two basic scenarios in which the assembly point for all the content elements is the web proxy cache or it is the web application server. For the adaptive core, we have selected the decision tree with all the independent attributes expressed as entire values. We have used two hardware architectures and two page content models. Thus, three experiments subsets have been created. Table 6.4 indicates the experiments of each of the subsets.

Each experiment subset has three experiments: one for the use of decision trees as implementation of the adaptive core; other for the use of a *split* approach; and the last one for the *join* approach. In this first experiment set,

Table 6.3. Features of the experiment design.

Feature	Values
Hardware architecture	distributed, centralized
Definition of the fragment designs	split, join, mace, entire/∅, entire/requestRate, entire/updateRate, entire/size, entire/fathersNumber, entire/childrenNumber, ratio/∅, ratio/requestRate, ratio/updateRate, ratio/size, ratio/fathersNumber, ratio/childrenNumber, distance/∅, distance/requestRate, distance/updateRate, distance/size, distance/fathersNumber, distance/childrenNumber, difference/∅, difference/requestRate, difference/updateRate, difference/size, difference/fathersNumber, difference/childrenNumber
Content page model	nytimes, pageflakes, yahoopipes
Request rate	2, 4, 6, 8, 10 (req./s)
Update rate	0.5, 1, 1.5, 2, 2.5 (req./s)
Characterization parameters changes	static, dynamic

Table 6.4. Design of the experiment set 1.

Experiment subset	Experiment identification
1A	`<distributed, entire/∅, nytimes, 4/0.1/s`$^{-1}$`, static>`
	`<distributed, split, nytimes, 4/0.1/s`$^{-1}$`, static>`
	`<distributed, join, nytimes, 4/0.1/s`$^{-1}$`, static>`
1B	`<centralized, entire/∅, nytimes, 4/0.1/s`$^{-1}$`, static>`
	`<centralized, split, nytimes, 4/0.1/s`$^{-1}$`, static>`
	`<centralized, join, nytimes, 4/0.1/s`$^{-1}$`, static>`
1C	`<centralized, entire/∅, pageflakes, 4/0.1/s`$^{-1}$`, static>`
	`<centralized, split, pageflakes, 4/0.1/s`$^{-1}$`, static>`
	`<centralized, join, pageflakes, 4/0.1/s`$^{-1}$`, static>`

the user load level of the system remains the same in all the experiments, and the characterization parameters values do not change.

The second experiment set (Table 6.5) is addressed to study and explore the behaviour of the adaptive core by using several decision trees and they are compared with one of the basic scenarios (*split*). In Section 4.4.3, we mentioned that several decision trees are generated from several training data sets, by modifying the number of independent attributes and the way they are expressed. An experiment for each of these decision trees is done. Two more variations have been added to the experiment set. In the second subset of experiments, the request load level of the system is changed (4.0 req./s

Table 6.5. Design of the experiment set 2.

Experiment subset	Experiment identification
2A	`<distributed,` $*^1/*^2$`, nytimes, 4/0.1/`s^{-1}`, static>`
	`<distributed, split, nytimes, 4/0.1/`s^{-1}`, static>`
2B	`<distributed,` $*^1/*^2$`, nytimes, 0.7/0.1/`s^{-1}`, static>`
	`<distributed, split, nytimes, 0.7/0.1/`s^{-1}`, static>`
2C	`<centralized,` $*^1/*^2$`, nytimes, 4/0.1/`s^{-1}`, static>`
	`<centralized, split, nytimes, 4/0.1/`s^{-1}`, static>`

$*^1$ entire, ratio, distance, difference.
$*^2$ $\emptyset$, requestRate, updateRate, size, fathersNumber, childrenNumber.

Table 6.6. Design of the experiment set 3.

Experiment subset	Experiment identification
3A	`<distributed,` $*^1/*^2$`, pageflakes, 2/0.4/`s^{-1}`, dynamic>`
	`<distributed, split, pageflakes, 2/0.4/`s^{-1}`, dynamic>`
3B	`<distributed,` $*^1/*^2$`, pageflakes, 0.5/0.1/`s^{-1}`, dynamic>`
	`<distributed, split, pageflakes, 0.5/0.1/`s^{-1}`, dynamic>`

$*^1$ entire, ratio, distance, difference.
$*^2$ $\emptyset$, requestRate, updateRate, size, fathersNumber, childrenNumber.

and 0.7 req./s). In the last subset, the other hardware architecture is tested (*centralized* instead of *distributed*). There is not any change in the values of the characterization parameters (*static* experiment) of these experiments.

In the experiment set 3 (Table 6.6), the effect of updates on the characterization parameters values is included in the study. The experiment explores the use of several decision trees with different user load levels, both in requests and updates. They are compared with one of the basic scenarios (*split*).

The fourth experiment set is addressed to study the variations of the user load levels (Table 6.7). The most representative of the decision trees is compared with the most representative of the basic scenarios. The load levels, both requests and updates, are varied in a wide range of values, for a total number of 25 cases.

Finally, the experiment set 5 is addressed to compare our solution with similar approaches of other authors (Table 6.8). The most similar approach is MACE framework [50, 48]. The design of the experiment is similar to the previous one, number 4. Instead of using the ODGex model of *pageflakes*, *yahoopipes* is used.

Table 6.7. Design of the experiment set 4.

Experiment subset	Experiment identification

4A	`<centralized, entire/`$\emptyset$`, pageflakes, `$*^3/*^4/s^{-1}$`, dynamic>`
	`<centralized, split, pageflakes, `$*^3/*^4/s^{-1}$`, dynamic>`

$*^3$ 2, 4, 6, 8, 10.
$*^4$ 0.5, 1, 1.5, 2, 2.5.

Table 6.8. Design of the experiment set 5.

Experiment subset	Experiment identification

5A	`<centralized, entire/`$\emptyset$`, yahoopipes, `$*^3/*^4/s^{-1}$`, dynamic>`
	`<centralized, mace, yahoopipes, `$*^3/*^4/s^{-1}$`, dynamic>`

$*^3$ 2, 4, 6, 8, 10.
$*^4$ 0.5, 1, 1.5, 2, 2.5.

6.5 Definition of the metrics to be monitored

The contributions of our dissertation are addressed to improve the performance without increasing significantly its overhead. Therefore, the metrics to be measured during the experimentation should refer to performance and system overhead.

The performance improvement of our solution is addressed to reduce the user-perceived latency. This latency time is only measured in the user side, so the user emulation tools should measure and store these values.

A second performance metric, in web cache environments, is the hit ratio or percentage of web requests that are stored and served locally by the cache proxy. This metric is created by analysing the number of requests, cache hit ratio, or by the size of the requests, cache byte hit ratio [12]. These values are obtained by the analysis of logs generated by the cache proxy (*Oracle Application Server Web Cache*).

There are two different logs, one for the web pages requests, and other for the ESI fragments request. In both cases, the size of the elements is indicated. Each log line corresponds to a request, and it indicates if the request has been a *hit* or a *miss*. By the programming of a simple tool, the log lines are processed and hits, requests and bytes are calculated.

The overhead is measured by the utilization of the resources in the computer servers. We have focused on the CPU and memory utilizations. *Vmstat* and *top* [54] were the monitoring tools we used to gather the utilization values. They are system monitoring tools that summarize performance information.

Top differentiates the CPU and memory utilization for each process in the system. Thus, we could distinguish the workload generated by the web cache, the application server and the DBMS. *Vmstat* summarizes the utilization values for all the processes in the system. Both of them gather the utilization values in fixed periods of time. In our case, they were configured to gather the values each 5 seconds.

6.6 Test-bed architecture

All the experiments have been executed using a test-bed. The test-bed is divided into two parts: use of real tools and tools to emulate the behaviour of some elements (Figure 6.1). The figure of the architecture of the test-bed is almost identical to Figure 3.2, but it indicates the names of the particular tools of our experimental environment.

The experiments should be executed isolated, in order to create replicable experiments. Therefore, the users of the system and the content service providers need to be emulated to repeat their behaviour in successive experiments.

We have created user traces files which indicate the actions (request or updates) that different users do in given points in time. Thus, their behaviour can be repeated as many times as it is necessary. We have implemented a *Java* tool in charge of reading the traces files and creating threads in order to emulate the different user sessions. The available user actions are the request of web pages or the update, creation and removal of content elements or aggregation relationships.

The content service providers have been emulated with web services which deliver the content elements. These web services have been implemented with the same PHP scripts using a RESTful web API. The databases of these web services have been populated with the content elements modelled in the different ODGex models of the experiments (*yahoopipes*, *nytimes*, and *pageflakes*).

The other tiers of the test-bed architecture have been deployed with real systems or tools. The web cache proxy has been deployed using *Oracle Application Server Web Cache 10g*. The application tier has been deployed using *Drupal* and the extension that we developed, which is explained in Section 5.3. The databases of the customized web pages and the aggregations structure of the web pages (*Fragment Design* database) have been populated with the data from the ODGex models.

The adaptive core has been developed using *Java* and the API of *Weka*. The different decision trees, from the two studied hardware architecture and from all the data instances representations, have been implemented in the adaptive core. They can be selected individually for each experiment.

In order to create an unattended execution of the test-bed, experiments are driven by a scheduling file. We have implemented Java Remote Method

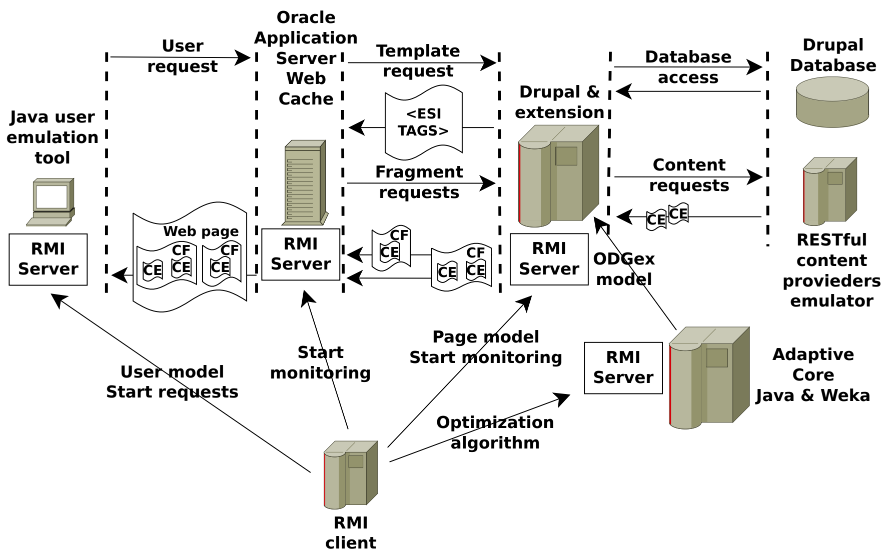


Fig. 6.1. Architecture of the test-bed.

Invocation (Java RMI) modules in all of the tiers. Java RMI [33] is the Java programming interface which implements remote procedure calls (RPC).

We have used RMI to coordinate all the tiers of the test-bed in order to restart the web proxy cache and the application server, to copy the log files with the performance data, to select the decision trees in the adaptive core, to select the user traces files in the user emulator, and to select the ODGex module in the applications server. A central RMI client is in charge of coordinating all the tiers by starting and stopping the tools and software in each of the tiers in the correct point in time. This client has the schedule of all the executions of the experiments.

6.7 Experiment execution

The execution time of the experiment is given by the number of requests of the traces files. All the traces files have 36000 requests. The time to execute the emulation depends on the request rate. For example, in the case of the experiment sets 4 and 5, the two extreme execution times would be one hour in the case of 10 req./s and five hours in the case of 2 req./s.

The traces files corresponding to the update actions have 45000 update requests. Not all of these requests are executed in all the experiments, just in those with the enough time to execute them. For example, in the experiments with 5 hours of execution and an update rate of 2.5 req./s, all the updates take places. But in an experiment of one hour of execution time and update rate of 0.5 req./s, just the first 1800 requests are done.

The performance of web caches is directly affected by the eviction algorithm. Our proposed system works independently to the eviction algorithm used in the cache. Therefore, we are not interested in studying the influence of the eviction policy. In order to avoid this influence, the total store size of the web cache has been configured high enough that all the content elements and fragment elements could be stored in the cache. Thus, the eviction from the cache is only caused because of the invalidation of the contents.

But the most important issue in the execution of the experiments is to guarantee the reliability of the results. We have used a similar method to the explained for creating the training data sets (Section 4.4.1). We have repeated the experiments several times (replicas) and we have studied the confidence interval for the means of requests among different replicas.

We have created the same number of replicas of each experiment, and we have checked if the 95% confidence level is smaller than the 2.5% of the average latency in the 90% of the samples of the replicas. If the experiment replicas did not fulfil this condition, the number of replicas for all the experiments in an experiment set was increased.

For example, in the case of the experiment set 5, the number of replicas of the experiment was 10. This experiment set is composed by 25 experiments, and the time to execute all of them is approximately 57 hours. So, the total emulation time for this experiment set, considering 10 replicas, is 570 hours, or, 24 days.

The confidence interval value, for samples in a replica, would be very high because of the high variation in the values of the latencies for a given web page. This variation is due to the influence of the cache proxy on the latency.

In the process of the calculation of the mean value, samples with cache hits and samples with cache misses are mixed, so, the values are very different. These alterations are increased in the experiments in which updates in the characterization parameters of the contents are taken into account. In these cases, the same web page request has different content characteristics (size, number of content elements, etc.) during the same experiment execution.

6.8 Summary

This chapter has been devoted to explain the experiments to validate the contributions of the dissertation. These experiments have been addressed to study performance metrics of the system in several conditions and variations of the test-bed environment. The results of the experiments are explained in next chapters.

The performance metrics to be studied in the experiments are related to the user and to the system. The user-perceived latency is studied to validate the improvement obtained with our solution. The system metrics, as CPU utilization, are analysed in order to validate the low overhead generated by the solution.

The details of the variations of the experiments have been explained. These variations are addressed to study our solution in different environments and with different conditions. Variations in the page models, user behaviour, and system architecture have been done.

The experiments have been executed in a test-bed environment. The tiers of the test-bed have been deployed using commercial and emulation tools. Some elements, as users or content services providers, have been emulated. The others tiers of the system have been deployed using commercial tools. We have also defined the execution conditions in order to obtain reliable results in the performance metrics.

7
Cache and user performance analysis

There are three principal means of acquiring knowledge available to us: observation of nature, reflection, and experimentation. Observation collects facts; reflection combines them; experimentation verifies the result of that combination.
—Denis Diderot—

This chapter is about the analysis of the results of the experimental phase. There are two main analysis blocks, the performance improvement and the overhead generated by the solution.

In this chapter, the conclusions about the performance improvement obtained with our solution are explained. The performance analysis is done using a user metric, the user-perceived latency, and two system metrics, the cache hit ratio and the cache byte hit ratio.

7.1 Introduction

The analysis of the results obtained in the experimental phase has been addressed to two fields: the performance improvement and the overhead generated. This chapter is devoted to explain the first one. The analysis of the performance improvement is done separately for each experiment set presented in the previous section. Firstly, we present the results of the data mining process over the two hardware architectures that we have taken into account: *centralized* and *distributed*.

It is important to remind that the objective of COFRADIAS framework is to improve the user perceived-latency. We have previously explained that the improvement of the latency is obtained in spite of a reduction of the performance of the cache (hit ratios). Therefore, the strength of our solution will be validated if the experiments show that our solution reduces the latency in comparison with the traditional CAS cache schemes or with other research studies. Besides this, we have also analysed the behaviour of the cache hit ratios in order to get more accurate conclusions.

7.2 Results of the data mining process

The first analysis of the results is related to the decision trees used in the classification algorithm of the adaptive core. We need to create a different version of the decision trees for each hardware architecture of the experiments. As we mentioned in Section 4.4, different hardware architectures require of different training process. We have used two different hardware architectures (*centralized* and *distributed*). Therefore, two processes of creation of decision trees have been done.

We have created one synthetic content model in order to obtain the performance results and to mine them. The parameterization of the setup for the creation of the synthetic model have been done generally enough in order to cover all the values ranges of the three content page models of the experiments (*nytimes*, *pageflakes* and *yahoopipes*). The features of the crawled web sites have been used to define the setup of the creation process. These values are indicated in Appendix B.

Table 7.1. Decision trees obtained from the *distributed* architecture.

Decision Tree	Selected	Size	Leaves	Coverage(%)	Notes
`<entire/∅>`	✓	91	46	92.14	
`<entire/updateRate>`	✓	5	3	81.04	
`<entire/requestRate>`	✓	19	10	90.78	
`<entire/size>`		1	1	65.21	1 state
`<entire/childrenNumber>`		1	1	65.21	1 state
`<entire/fathersNumber>`		1	1	65.21	1 state
`<ratio/∅>`	✓	63	32	92.72	
`<ratio/updateRate>`	✓	13	7	88.33	
`<ratio/requestRate>`	✓	9	5	82.18	
`<ratio/size>`	✓	49	25	89.99	
`<ratio/childrenNumber>`		1	1	65.21	1 state
`<ratio/fathersNumber>`	✓	27	14	91.08	
`<diff./∅>`	✓	7	4	81.11	
`<diff./updateRate>`		7	4	81.11	`<diff./∅>`
`<diff./requestRate>`	✓	9	5	93.84	
`<diff./size>`		7	4	81.11	`<diff./∅>`
`<diff./childrenNumber>`		1	1	65.21	1 state
`<diff./fathersNumber>`		7	4	81.11	`<diff./∅>`
`<dist./∅>`		1	1	65.21	1 state
`<dist./updateRate>`		1	1	65.21	1 state
`<dist./requestRate>`	✓	3	2	79.73	
`<dist./size>`		1	1	65.21	1 state
`<dist./childrenNumber>`		1	1	65.21	1 state
`<dist./fathersNumber>`		1	1	65.21	1 state

Because of the variations in the number of independent attributes of the data instances, and in the way that these attributes are expressed, we obtain a maximum of 24 decision trees. As an example of a decision tree, we can observe the graphical representation of the *ratio/∅* of the *distributed* hardware architecture in Figure 7.1. The other decision trees are presented in Appendix A.

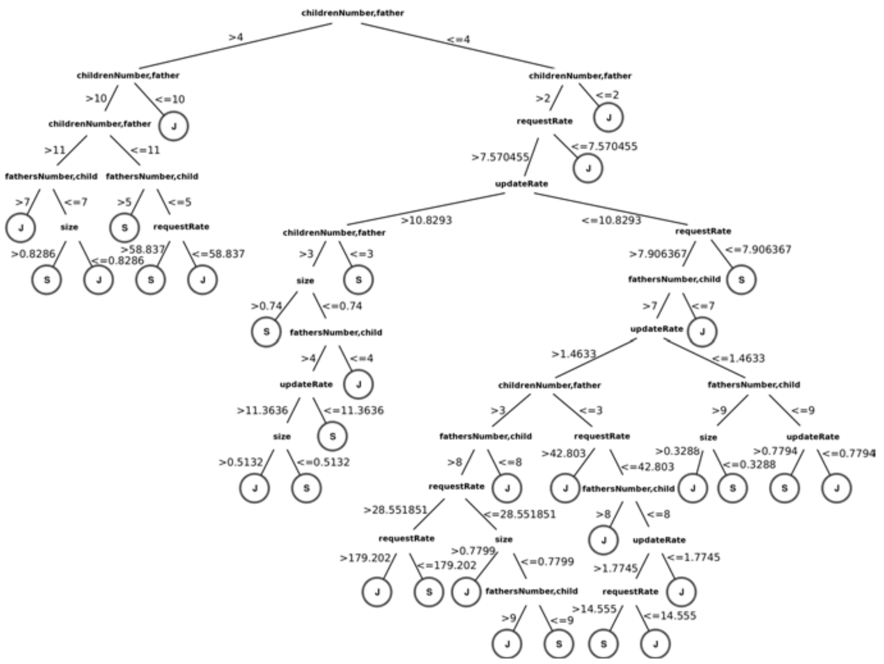


Fig. 7.1. Decision tree corresponding to the *ratio/∅* of the *distributed* hardware architecture.

After the mining process, we realised that some of the obtained trees had to be rejected. This is because, in some of the cases, the obtained tree has a size of 1. And this means that all the instances are classified in same state, *i.e.*, the ODGex model obtained with one of these trees would behave in the same way that one of the traditional schemes —where all the content fragments are assembled in the web cache or in the web application—. We do not have to execute the experiments corresponding to these decision trees because their performance results would be the same that using *split* or *join* experiments.

We interpreted the trees with size 1 as cases where the data instances do not offer enough information to create a suitable decision tree, and the algorithm considers, as the best option, to classify all the instances in the same class. This is because the C4.5 algorithm cannot find any independent attribute that splits the samples into two subsets effectively, or, the candidate attributes split the samples into two subsets with less coverage than the case of classifying all the samples in a same class [86, 47].

Tables 7.1 and 7.2 list all the decision trees and, in particular, those rejected because of their sizes. These tables indicate the total number of nodes (*size*) and the number of nodes that assign a state (*leaves nodes*). The trees, which always assign the same state, have only 1 leave node. This situation has been labelled as *1 state* in the column *Notes* of the tables.

We have neither considered the experiments with the same decision tree than a previous one. This mainly occurs when the size of the tree is small and few independent attributes are really used in the decision tree. The trees that are equal to other ones have the name of their equivalent trees in the column *Notes*.

Finally, the coverage value of the tree is also indicated in the table. The coverage value is the ratio of instances from the test data set that have been classified correctly by the use of the decision tree.

Table 7.2. Decision trees obtained from the *centralized* architecture.

Decision Tree	Selected	Size	Leaves	Coverage(%)	Notes
`<entire/∅>`	✓	51	26	91.26	
`<entire/updateRate>`		1	1	59.88	1 state
`<entire/requestRate>`	✓	25	13	93.19	
`<entire/size>`	✓	7	4	85.16	
`<entire/childrenNumber>`		1	1	59.88	1 state
`<entire/fathersNumber>`		7	4	85.16	`<entire/size>`
`<ratio/∅>`	✓	51	26	95.73	
`<ratio/updateRate>`	✓	43	22	92.84	
`<ratio/requestRate>`	✓	15	8	90.12	
`<ratio/size>`	✓	35	18	96.07	
`<ratio/childrenNumber>`		1	1	59.88	1 state
`<ratio/fathersNumber>`		1	1	59.88	1 state
`<diff./∅>`	✓	17	9	89.60	
`<diff./updateRate>`	✓	11	6	88.99	
`<diff./requestRate>`		11	6	88.99	`<diff./upd.Rate>`
`<diff./size>`		1	1	59.88	1 state
`<diff./childrenNumber>`		1	1	59.88	1 state
`<diff./fathersNumber>`		1	1	59.88	1 state
`<dist./∅>`	✓	9	5	87.75	
`<dist./updateRate>`	✓	5	3	81.09	
`<dist./requestRate>`		5	3	81.09	`<dist./upd.Rate>`
`<dist./size>`	✓	9	5	86.45	
`<dist./childrenNumber>`		1	1	59.88	1 state
`<dist./fathersNumber>`		1	1	59.88	1 state

By observing the sizes of the trees, we will be able to differentiate the trees corresponding to *entire* and *ratio* from *difference* and *distance*. The group of the first two has, in general terms, bigger sizes. This is quite reasonable because there are data losses with the two second representation groups. Indeed, the last group, *distance*, has the smallest tree sizes and it is the representation with less information about the independent attributes, only the absolute value of the difference of the common attributes. By observing the performance results of the experiments, we will be able to state if it is obtained, or not, a higher performance with experiments using bigger decision trees. Initially, *ratio* and *entire* representations seem to be the best ones.

About the number of independent attributes, we can observed that there is a high number of cases of decision tree sizes equal to 1 when *childrenNumber* and *fathersNumber* are removed from the samples. This may be interpreted as these two attributes are very correlated with the *highest performance* state because a suitable decision tree cannot be created when they are removed.

To summarize, only the experiments corresponding to the decision trees with a tick symbol (✓) in the tables are executed. The other experiments are rejected because their results would be the same than the traditional schemes.

7.3 Performance results

This section is devoted to the analysis of performance results of the experiments. The experiments have been organized as they were presented in the experiment design (Section 6.4). We present information about the user-perceived latency and about the web cache performance metrics for each experiment subset.

The results of the user-perceived latency are shown as speed-ups. Instead of showing the absolute latency times, we considered that the value of the speed-up of each execution related to a common execution case is more illustrative. In almost all the experiment sets, this common execution case is one of the corresponding to the traditional schemes —content fragments assembled in web cache *split* or assembled in web application *join*—, except for the final experiment in which we have compared the results of the experiments of our proposal with the obtained by using a solution of other researchers, the MACE approach [48].

Despite the user-perceived results are expressed as speed-ups compared with a basic common experiment, we are also going to show the latency values in one of the experiments (experiment subset 1A) to show the magnitude of their absolute values. Our analysis is done independently of the latency value because we are interested in the improvement of our solution compared with other solutions.

In our experiments, we request web pages that are completely different among them. They have a wide range of values of their characterization parameters. Thus, the response times of different web pages are very heterogeneous, and the improvement of using our solution is also very different between web pages. For example, we can obtain greater improvements in pages with higher number of content elements. A higher number of content elements generate a higher number of possible fragment designs and, in consequence, a greater scope to improve the single latency of the page. Therefore, we prefer to show all the single improvements of all different web pages.

We have represented the speed-ups in a rank order plot. Each point of the plot is the speed-up of a single web page. The y-axis, of the rank order plot, represents the value of the speed-up in relation with the common experiment. The x-axis represents the position of a web page in a rank order of the speed-ups. The value n of the x-axis is the web page with the n-th improvement. The first element in the x-axis ($x = 1$) is the page with the smallest improvement, the last element in the x-axis (for example $x = 481$) is the web page with the greatest improvement. The element with $x = n$ has $n - 1$ web pages with smaller improvements, and it is in the n-th place in an ordered list based on the speed-up of the web pages.

$$Speed - up = \frac{UPL(commonExecutionWebPage)}{UPL(analysedExecutionWebPage)} \tag{7.1}$$

The speed-up is calculated as the Equation (7.1) where UPL is the user-perceived latency. Speed-ups of value 1.0 are the cases where the latency is the same in the *analysed* execution and in the *common* execution. If the speed-up is above 1.0, the latency of the web page is shorter when the solution of the *analysed* execution is used. If the value is below 1.0, the solution of the *common* execution case has a shorter latency. In all of the experiments we have considered that the *analysed* execution latencies correspond to the latencies of our proposed solution. The *common* execution latencies correspond to the cases which we are comparing with, one of the traditional schemes or, in the last experiment set, the MACE solution.

An experiment execution is better than other one when the plot of the first one is above of the plot of the second one, *i.e.*, when the speed-ups are bigger. The problem appears when the previous fact only occurs in a subset of all the web pages of an experiment. We have also calculated the mean value of the web pages speed-ups of an experiment execution in order to compare it with other executions. We noticed that the speed-ups values of the extreme cases —the web pages associated with the best speed-ups and with the worse ones— sometimes have a great weight in the results. Therefore, we have also considered to calculate the mean value of the speed-ups of the samples between the 10th percentile (P_{10}) and the 90th percentile (P_{90}). We use these mean values only to compare the experiments themselves. We are not interested in the dispersion of the data, so we have not considered calculating the variance of the standard deviation of the speed-ups of the set of web pages of an experiment.

Each experiment execution has been repeated several times (replicas) in order to avoid data alterations because of temporal system anomalies. The minimum number of executions has been already explained in Section 6.7. To calculate the mean values of the latencies, we have calculated first the mean value of the same samples (requests) among the replicas of an experiment. This is possible because the order of the requests is always the same. Thus, we get mean latency value of each request of an experiment. In the case of *static* experiments, we can finally calculate the mean value of a given web page, considering all the requests of the same page during the execution. But in the case of the *dynamic* experiments, the web pages change their structure and size (number of content elements, size of the content elements, etc.) during the execution, so the content of the response of a given request is completely different along the time. In these cases, we have analysed the latency of each single request independently. On the contrary, in *static* experiments, we have analysed the latency of web pages.

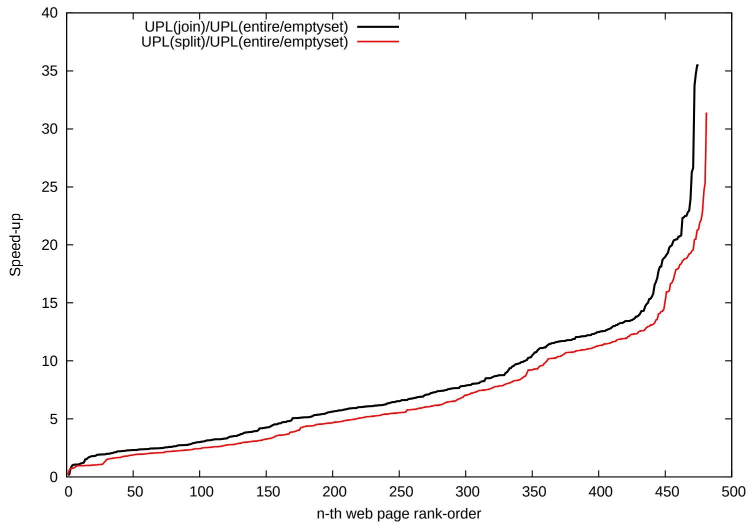
The metrics related to the performance of the web cache that we have analysed are the hit ratio and the byte hit ratio. They are, respectively, the percentage of requests and the percentage of bytes that have been served from the web cache. We have calculated both ratios for each replica of an experiment and, finally, we have calculated the mean value of the hit and byte hit ratio of the replicas of each experiment. The standard deviation has been also calculated. It is important to remind that the invalidations in the cache are only due to the expiration of the contents, *i.e.*, the cache misses are only generated because the content elements have changed. This is because the size of the cache has been assigned big enough to store all the possible fragments of the content page model.

Finally, we have also considered the size of the transient state. The transient state is the period of time at the beginning of an experiment execution. To detect this period, we have analysed the mean value of the cache hit ratio. The transient state finishes when the cache hit ratio keeps inside a small range of values. The samples in the transient state are rejected. We have analysed the transient state using the means among replicas of an experiment.

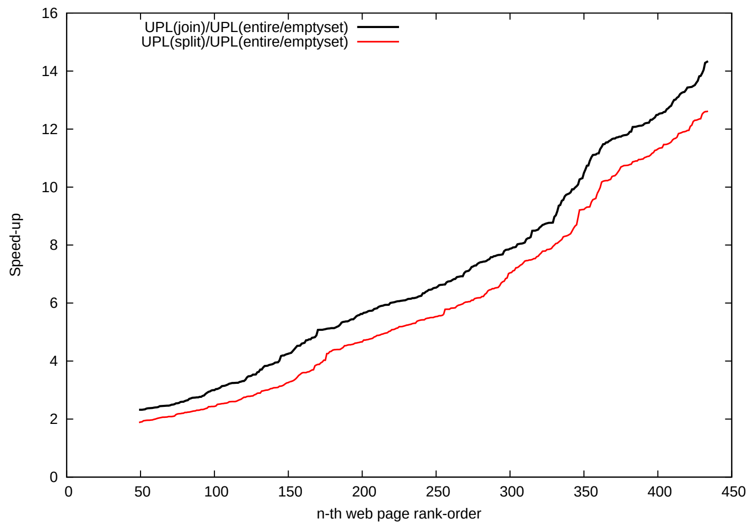
7.3.1 First experiment set

The experiment set 1 is addressed to study if our proposed system obtains shorter user-perceived latencies than with the use of traditional schemes —all the assemblies of the content elements occur in the web cache or in the web application—. Thus, three different experiment subsets have been defined in order to study the improvement in different hardware architectures and page models (Table 6.4). We presented similar and partial results in [44]. But in this publication, the experimental phase was done with shorter emulation time and using only one replica execution. The results presented in this dissertation are quite more reliable.

In the first experiment subset (1A) we have used the page model extracted from a news site (*nytimes*), the emulation have been executed in a distributed hardware architecture —the web cache and the web applications are deployed



(a) 100% of the samples



(b) P_{10}-P_{90} of the samples

Fig. 7.2. Speed-up of the user perceived latency in experiment subset 1A.

Table 7.3. Speed-ups of the user-perceived latency for experiment subset 1A.

Speed-up = UPL(`<ExId>`)/UPL(`<entire/∅>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<join>`	7.7031	2.3016–14.3103	7.0028
`<split>`	6.7306	1.8639–12.6024	6.0618

in different computers—, and the request rate is high, in comparison with the update rate (4.0 req./s and 0.1 req./s respectively). The results of the emulation are in Figure 7.2 and Table 7.3.

As this experiment subset is *static*, we can show the mean latency time for each web page. Therefore, the rank order plot shows the speed-ups for the 481 web pages of the page model. We can clearly observe that most of the samples show better latencies when the execution has been done with our proposed adaptive fragment design, and this occurs both in comparison with the *join* scenario and the *split* scenario. The obtained speed-ups are significant, but it is important to remind that this is a *static* experiment without overhead from the classification process (it is done at the beginning of the emulation) and without temporal and partial transient states due to the changes in the structure and characteristics of the content elements and web pages.

This execution cannot be considered the example for a real environment, but it allows us to check that our proposal is promising and it offers important improvements. The extreme cases —the web pages with smallest and greatest improvements— are very different to the regular ones, but they do not have much influence on the mean values because both mean values —considering all the web pages or only the web pages inside P_{10}-P_{90} range— are quite similar.

An important conclusion, to be validated with the next two experiments is that the scenario in which all the content elements are assembled in the web cache (*split*) obtains shorter latencies than the *join* scenario. We observe that the improvement of our solution is greater when we compared it with the *join* scenario.

As an example, we have also represented the latencies of the *split* and *entire/∅* experiments. We are interested in the improvement of the latencies instead of the values of these latencies. Therefore, we are not going to show the values of the latencies in the next experiments. But we have considered that, in order to have a complete view of the system, it could be interesting to show the latencies of some of the experiments. We can observe the latencies of the web pages of this experiment subset in the plot of Figure 7.3. There are three data series in the plot. Two of the series correspond to the latencies of the web pages for the experiments *entire/∅* and *split* in an ordered way. The x-axis represents the web pages, but, for these two series, the same x-values of two points do not mean that they are the latencies of the same web page.

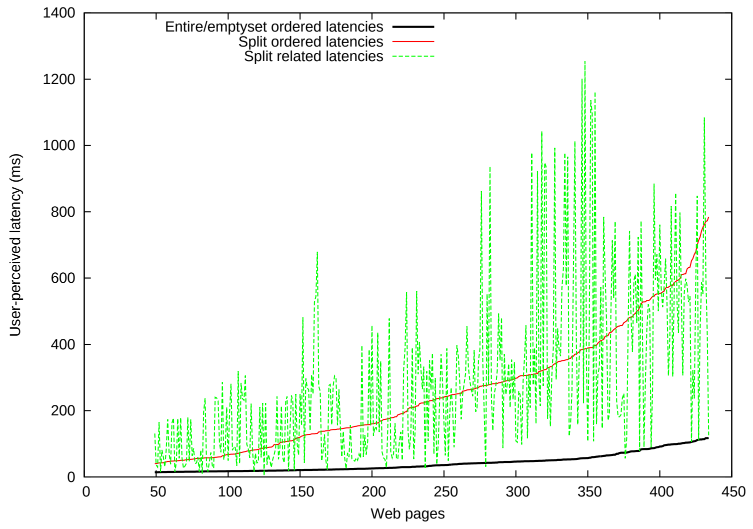


Fig. 7.3. User-perceived latencies of the experiment subset 1A for the experiment executions *split* and *entire/∅*.

This is because the data series are ordered by the latency values, and the web page orders are not the same for the two experiments. The last data series corresponds to the *split* scenario latencies of the web pages in relation to the same web page of the ordered series of the *entire/∅*, *i.e.*, for these two data series, two points with the same x-value correspond to the same web page.

The web cache metrics results for experiment subset 1A are presented in Table 7.4. As expected, the best hit ratio and byte hit ratio are obtained with the *split* scenario because it has the smallest fragment elements. Our solution worsens the hit ratio in order to reduce the overhead times of the assembling process. The *join* scenario shows worse hit ratios because each content update invalidates a whole web page.

Our dissertation is focussed on reducing the overhead times of the assembling process, balancing this improvement with the hit ratio losses of pre-assembling some content elements in the web application tier. As we explained in Section 2.3.2, these overhead times correspond to database connections, data transmission, network delays, content parsing, etc. In a *centralized* architecture, some of these overheads are practically nil because the communication process among the system tiers takes place inside the same computer.

The second experiment subset (1B) is addressed to evaluate our solution in an environment with lower overhead, as the *centralized* hardware architec-

Table 7.4. Cache performance for the executions of the experiment subset 1A.

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<entire/∅>`	94.9456	0.9471	91.9454	0.8741	1100
`<join>`	86.1592	0.8421	80.3284	0.9712	900
`<split>`	95.8284	0.8412	92.2621	0.7411	850

ture. The only difference between experiments 1A and 1B is the hardware architecture. The data mining off-line process and the use of decision trees are also validated, because the decision trees are obtained in both different data mining process for both hardware architectures.

The improvement of our solution, in terms of user-perceived latency, are shown in Figure 7.4 and in Table 7.5. As in the first experiment, the speed-up values of almost all the web pages are above 1.0. So the suitability of our proposal is also validated in the *centralized* scenario. But, as expected, the mean speed-ups are significantly smaller, because of the reduction of the overhead times. If the overhead times are shorter, the improvements of our solution are smaller. This mainly affects the improvements over the *split* scenario. Despite this, our solution is still better than the two traditional scenarios, and the *split* scenario is again better than the *join* one.

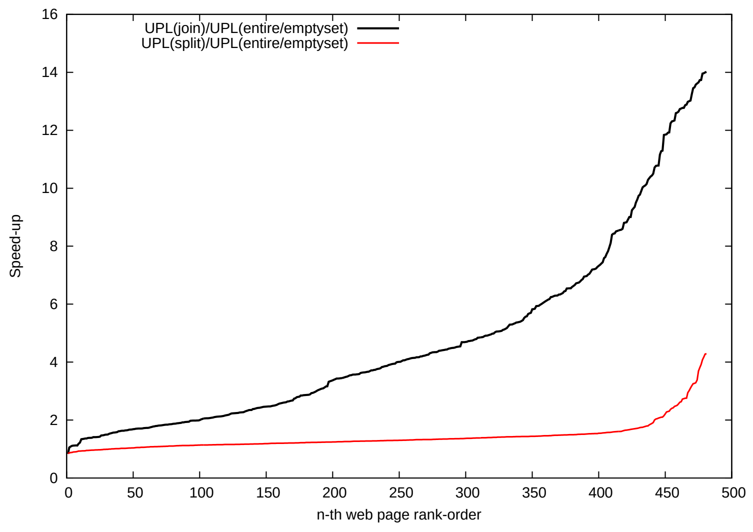
Once again, the extreme values do not have much influence on the mean value of the speed-ups of all the web pages. This is because the worse cases balance the best ones, *i.e.*, there are a very similar number of both cases and with an inversely proportional relationship.

Table 7.5. Speed-ups of the user-perceived latency for experiment subset 1B.

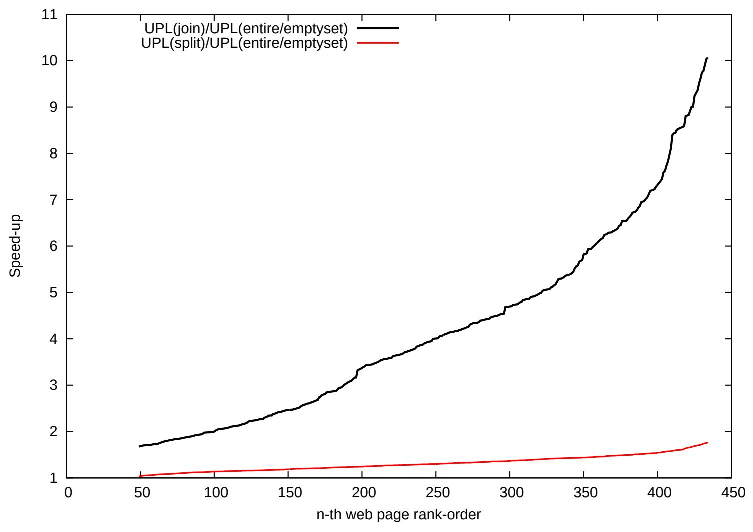
Speed-up = UPL(`<ExId>`)/UPL(`<entire/∅>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<join>`	4.7665	1.7059–10.4026	4.3252
`<split>`	1.4113	1.0347–1.7665	1.3106

The results of the web cache metrics (Table 7.6) validate again the explanations given in experiment 1A. The *split* scenario shows the best hit ratio, and our proposal worsens the hit ratio, but it improves the user-perceived latency. Indeed, the hit ratios, for the executions of the *split* and *join* scenarios of this experiment, should be identical to the results of experiment 1A. This is because the user behaviour model for the emulation (request log and update log) is the same.



(a) 100% of the samples



(b) P_{10}-P_{90} of the samples

Fig. 7.4. Speed-up of the user perceived latency in experiment subset 1B.

Table 7.6. Cache performance for the executions of the experiment subset 1B.

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<entire/∅>`	93.7201	0.5788	91.0390	0.4781	950
`<join>`	86.4017	0.0019	79.4391	0.9741	1000
`<split>`	96.1629	0.6174	92.0072	0.7120	900

Finally, the experiment is repeated in a *centralized* scenario, but using a second content page model. Once we have studied the validity of our solution in two different hardware architectures, we want to know if the solution is also valid for other content page models. We have done the comparison with the worst scenario for our solution, the *centralized* one. The new page model corresponds to *pageflakes* and this affects in the plots because now there are 2000 web pages in the model instead of 481. The x-axis of the plots represents the list of the pages ordered by their improvement. The results are presented in Table 7.7 and Figure 7.5.

Through the observation of the plots, the improvement obtained by our solution seems to be smaller than in the previous cases. However, if we analyse the mean values of the speed-ups we observe that the mean values of experiment 1B and 1C are quite similar. The main difference is in the extreme cases. For this experiment the speed-ups values of the extreme cases are more similar to the regular ones. Probably, the setup of the web pages and the fragment designs generated by the decision tree are more similar among the pages. Or it can be even explained by the user behaviour emulation. The user behaviour log for this experiment is different to those generated for experiments 1A and 1B because of the use of a different page model.

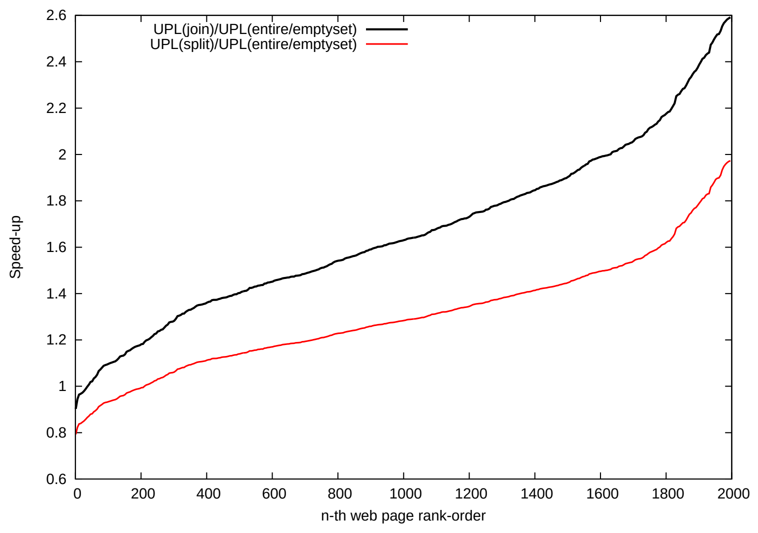
In any case, we observe levels of general improvement similar to experiment 1B and lower to the 1A. This is explained again by the use of a *centralized* hardware architecture.

This experiment uses a different user behaviour model to the two previous ones. Therefore, the cache performance is different to the cases 1A and 1B (Table 7.8). But these differences are minimal. The pattern of the best ratios in the *split* scenario and the worst ratios in the *join* one is repeated again.

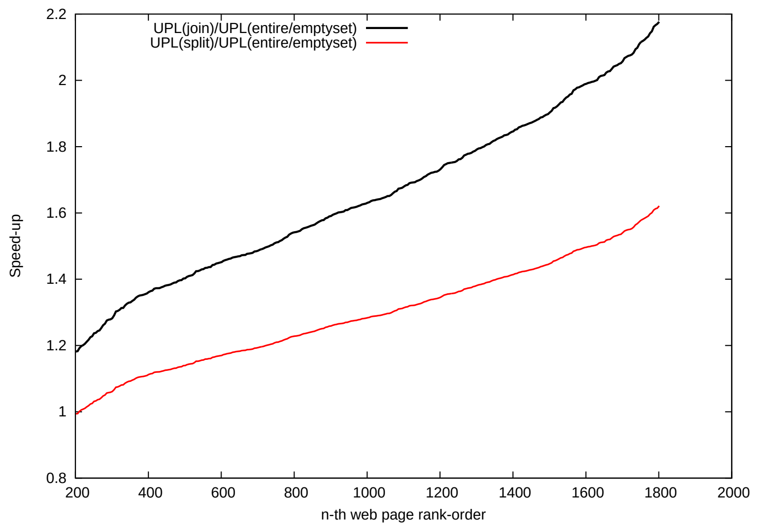
Table 7.7. Speed-ups of the user-perceived latency for experiment subset 1C.

Speed-up = UPL(`<ExId>`)/UPL(`<entire/∅>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<join>`	1.6666	1.1770–2.1701	1.6489
`<split>`	1.3052	0.9907–1.6145	1.2915



(a) 100% of the samples



(b) P_{10}-P_{90} of the samples

Fig. 7.5. Speed-up of the user perceived latency in experiment subset 1C.

Table 7.8. Cache performance for the executions of the experiment subset 1C.

Decision tree	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<entire/∅>`	88.7300	0.0148	88.1943	0.1008	950
`<join>`	80.2731	0.8124	78.8818	0.4178	1000
`<split>`	91.7204	0.0513	90.1782	0.6178	900

After the analysis of the results of the first experiment set we have validated that:

- Our solution improves the performance, in terms of user-perceived latency, in spite of it worsens the hit ratio of the cache.
- The scenarios using a *split* approach have better performance results than the other traditional solution for CAS systems.
- Our solution shows greater improvement in the hardware architectures in which the assembling overheads are higher.

Therefore, in the next experiments, we are going to compare the experiment based on our approach only with the better traditional scenario *split*.

7.3.2 Second experiment set

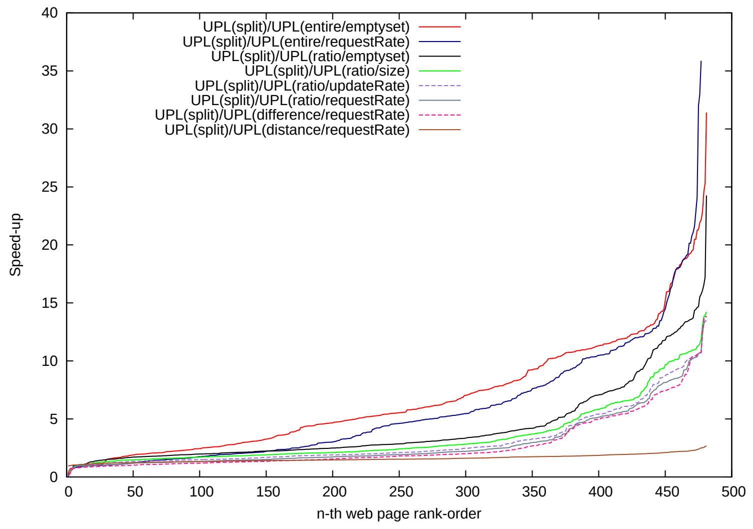
The second experiment set is addressed to explore the decision trees obtained by the variation on the training data set representations and on the number of the independent attributes of these data sets. Its experiment subsets and executions were shown in Table 6.5. We have also taken into account the variations in the hardware architecture and in the user load level of the user requests.

The experiment subsets are executed in a *static* configuration and with the same page model, *nytimes*. The results of this experiment set have been partially published in [42] and in [43].

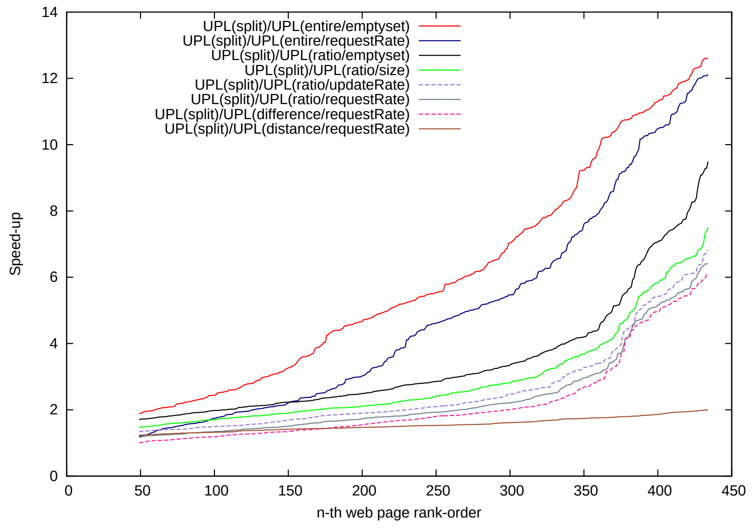
The results for the latencies speed-ups of the first experiment subset (2A) are presented in Table 7.9 and Figure 7.6. For a better analysis, the identifiers of the figure legend are ordered as the mean values presented in the table. All the speed-ups have been calculated in relation with the latency results of the execution in the *split* traditional scenario.

There are two experiment executions —*entire/∅* and to *entire/requestRate* decision trees— with a considerable higher performance than the rest of executions. On the contrary, the execution which uses the *distance/requestRate* decision tree shows the smallest improvements. The rest of executions have similar performance.

In Table 7.1, there were 11 decision trees selected to be analysed in the experiment executions. But we have presented only the performance of 8 of



(a) 100% of the samples



(b) P_{10}-P_{90} of the samples

Fig. 7.6. Speed-up of the user perceived latency in experiment subset 2A.

Table 7.9. Speed-ups of the user-perceived latency for experiment subset 2A.

Speed-up = UPL(`<split>`)/UPL(`<ExId>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<entire/∅>`	6.7306	1.8639–12.6024	6.0618
`<entire/requestRate>`	5.6883	1.1832–12.0783	4.9182
`<ratio/∅>`	4.1893	1.6855–9.2910	3.4624
`<ratio/updateRate>`	3.1301	1.3359–6.7255	2.6026
`<ratio/requestRate>`	2.9116	1.1637–6.4018	2.4000
`<ratio/size>`	3.4745	1.4630–7.3901	2.9200
`<difference/requestRate>`	2.7168	1.0086–6.0691	2.2156
`<distance/requestRate>`	1.5766	1.2309–2.0213	1.5523

them. This experiment is based on a *static* execution and, in consequence, the classification process is done just one time at the beginning of the emulation. We observed, in this classification process, that the three remaining trees classified all the edges of the content page model in the same state, *i.e.*, the result of the classification is one of the traditional scenarios. This is the reason for rejecting the use of these decision trees in the experiments in which the *nytimes* page model is used in combination with a *static* execution.

If we compare the performance obtained in the emulation and the characteristics of the decision trees for the *distributed* hardware architecture, we do not observe any relationship between the size, and the coverage, of the tree with the speed-ups obtained. It is true that the best experiment execution corresponds to the decision tree with the biggest size (*entire/∅* with 91 vertexes). But, for example, the execution using the third biggest decision tree (*ratio/size* with 49 vertexes) shows a regular performance. Its performance is very similar to the execution of the *difference/∅*, which has only a size of 7 vertexes, and it is quite smaller than the performance of *entire/requestRate*, which only has 19 vertexes.

The data about the web cache results are presented in Table 7.10. As expected, the *split* scenario has the highest cache ratios, and all the executions based on decision trees have lower ones. The decision tree with the best latencies (*entire/emptyset*) also has the best hit ratio, except for the *split* one. Therefore, we could think that the benefits, in terms of latency, are achieved because of the hit ratio instead of the benefits of our solution. But the hit ratios of the second and third *best* decision trees (*entire/requestRate* and *ratio/∅*) are the worse ones. Therefore, there is not a direct relationship between the performance of the cache and the improvement of the latencies. The hit ratio has influence on the user-perceived latency, but also the overhead of the assembling process. Indeed, the results of the experiments show us that the latency can be improved in spite of our proposed solution obtains lower hit ratios than the *split* scenario.

Table 7.10. Cache performance for the executions of the experiment subset 2A.

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<entire/∅>`	94.9456	0.9471	91.9454	0.8741	1100
`<entire/requestRate>`	82.1692	0.0587	78.5413	0.3577	1200
`<ratio/∅>`	78.9625	0.6417	74.9054	0.2884	800
`<ratio/updateRate>`	85.6546	0.5005	74.6698	0.9471	950
`<ratio/requestRate>`	90.1876	0.7142	80.5245	0.2844	900
`<ratio/size>`	89.5868	0.8471	80.2355	0.9841	1000
`<diff./requestRate>`	86.6386	0.0984	76.8914	0.1007	900
`<dist./requestRate>`	80.8227	0.7142	74.6325	0.7418	950
`<split>`	95.8284	0.8412	92.2621	0.7411	850

The setup of the second experiment (2B) is almost identical to the first one, but we have changed the user load level of the system. This second experiment studies the behaviour of the decision trees in low load levels, with 0.7 read requests per second and 0.1 update requests per second. We are interested in studying if the *improvement order* of the trees remains equal in different hardware and load conditions.

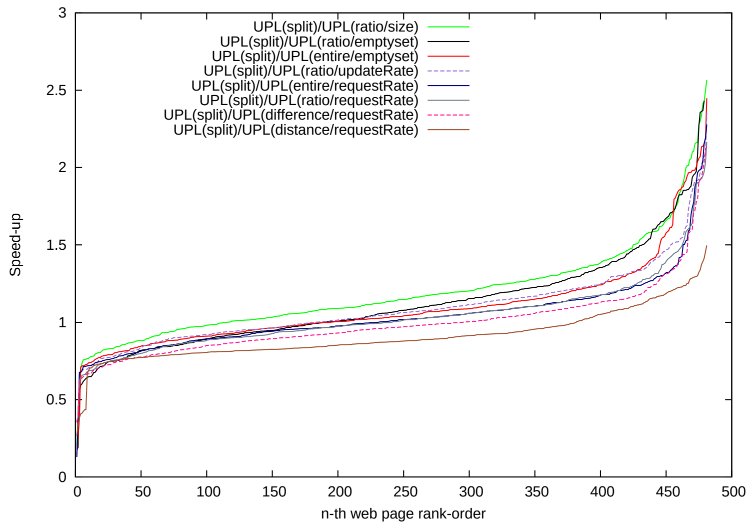
The results of this experiment are in Figure 7.7 and Table 7.11. For a better analysis, the identifiers of the figure legends are ordered as the mean values presented in the table.

The first conclusion of the analysis of the results is that in low level conditions the benefits of our proposed solution are smaller. We can observe that all the experiments based on decision trees shows smaller speed-ups in this experiment than in experiment 2A, and the only difference between both experiments is the user load level. Therefore, the clear explanation of these losses in the improvement is related to this change in the user load level.

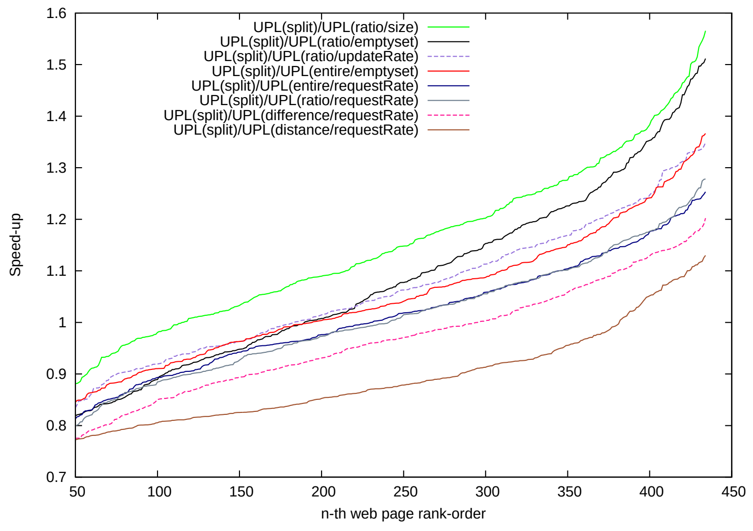
The second conclusion is that the differences between the improvements of the decision trees have been reduced. This is probably because the reduction in the user load level. In order to validate this, further experiments are done in experiment set 3.

In the previous experiment there were three different groups of decision trees, based on their improvements. The best results are now experimented by the cases of *ratio/∅* and *ratio/size*, but *entire/∅* remains as one of the three best experiments. The worst results are obtained with the same trees than in the previous experiment (*difference/requestRate* and *distance/requestRate*).

The results related to web cache are presented in Table 7.12. Once again, there is not relation between the cache performance (hit and byte hit ratios) and the speed-ups obtained by the trees of our proposed system. The values of the hit ratios are smaller than in the previous experiment (2A), because the request rate of 2B is lower than in 2A, and the update rate remains constant.



(a) 100% of the samples



(b) P_{10}-P_{90} of the samples

Fig. 7.7. Speed-up of the user perceived latency in experiment subset 2B.

Table 7.11. Speed-ups of the user-perceived latency for experiment subset 2B.

Speed-up = UPL(`<split>`)/UPL(`<ExId>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<entire/∅>`	1.0926	0.8416–1.3625	1.0517
`<entire/requestRate>`	1.0377	0.8066–1.2484	1.0144
`<ratio/∅>`	1.1157	0.8123–1.5207	1.0849
`<ratio/updateRate>`	1.0884	0.8251–1.3416	1.0639
`<ratio/requestRate>`	1.0347	0.7948–1.2773	1.0110
`<ratio/size>`	1.1884	0.8801–1.5539	1.1525
`<difference/requestRate>`	0.9924	0.7745–1.1906	0.9676
`<distance/requestRate>`	0.9064	0.7715–1.1253	0.8927

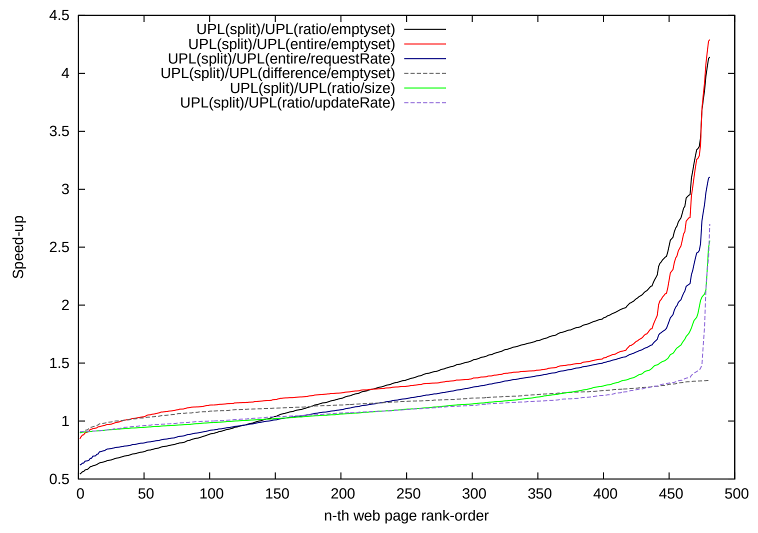
And, finally, the ratios of the *split* scenario are bigger than the rest of the cases.

The last experiment (2C) of this experiment set is defined to study the behaviour with another hardware architecture. Thus, experiment 2C is identical to the first one (2A) except that 2C uses a *centralized* hardware architecture. As the hardware architecture is different to the previous ones, the selected decision trees are also different. We obtained 12 suitable trees for the *centralized* architecture, but we have executed only the experiments corresponding to 6 of them because, in the other cases, all the aggregations (edges) were classified in the same state, *i.e.*, the result of the classifications was one of the traditional scenarios (*split* or *join*). The results of this experiment are in Figure 7.8 and Table 7.13. For a better analysis, the identifiers of the figure legend are ordered as the mean values presented in the table.

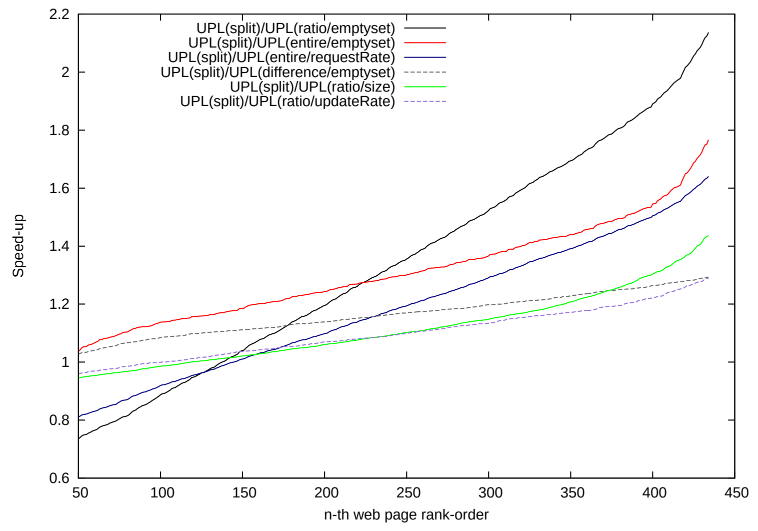
As it was observed in the experiment set 1, the results of experiments with a *centralized* hardware architecture show worse results than in the case of the

Table 7.12. Cache performance for the executions of the experiment subset 2B.

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<entire/∅>`	87.1274	0.7496	84.1742	0.9748	1000
`<entire/requestRate>`	79.7719	0.6684	73.4812	0.4178	900
`<ratio/∅>`	74.8744	0.4591	71.9527	0.3991	950
`<ratio/updateRate>`	80.8347	0.5711	71.8338	0.3011	1050
`<ratio/requestRate>`	84.4400	0.1289	78.1749	0.3675	1000
`<ratio/size>`	83.1937	0.0571	73.7492	0.1878	900
`<diff./requestRate>`	80.7433	0.1794	74.0081	0.5574	1100
`<dist./requestRate>`	77.8477	0.7415	73.7198	0.5946	1050
`<split>`	90.1748	0.0770	87.1664	0.0911	800



(a) 100% of the samples



(b) P_{10}-P_{90} of the samples

Fig. 7.8. Speed-up of the user perceived latency in experiment subset 2C.

distributed one. Thus, the hypothesis of greater improvement in the cases with higher assembling overheads is validated again with the experiment set 2. Experiment 2A shows better results than 2C and the only difference is the hardware architecture.

Once again, the experiments based on the *entire/∅* and *ratio/∅* trees are in the group of cases with best results. Therefore, it seems that these two decision trees are the best ones to implement the classification algorithm of the adaptive core. They show the greatest improvements of the user-perceived latency independently of the conditions of the experiment. Although the mean latency values of *ratio/∅* are quite interesting, it shows longer latencies than the reference scenario (*split*) in 137 web pages (around the 28.5% of web pages).

Finally, the improvement of the decision trees does not seem to have any relation with the size or the coverage of the trees. For example, the size of the tree *ratio/size* is smaller than *entire/requestRate*, but the performance is higher for the second one. On the contrary, the two biggest trees (*entire/∅* and *ratio/∅*) correspond to the experiments with the highest performance. We cannot finally conclude any clear statement about the relation of the sizes of the trees and the improvement of their corresponding experiments. Further research work could be addressed in this direction.

Table 7.13. Speed-ups of the user-perceived latency for experiment subset 2C.

Speed-up = UPL(`<split>`)/UPL(`<ExId>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<entire/∅>`	1.4113	1.0347–1.7665	1.3106
`<entire/requestRate>`	1.2353	0.8089–1.6405	1.1848
`<ratio/∅>`	1.4282	0.7320–2.1365	1.3445
`<ratio/updateRate>`	1.1180	0.9588–1.2882	1.0994
`<ratio/size>`	1.1565	0.9446–1.4358	1.1138
`<difference/∅>`	1.1619	1.0262–1.2929	1.1637

The patterns of the results for the cache performance (Table 7.14) remain similar to all the previous experiments: the highest hit ratios correspond to the *split* experiments and there is not relationship between the hit ratios and the improvement, in terms of latency, of the experiments based on the use of decision trees.

The summary of the conclusions for the second experiment are:

- Our solution improves the performance, in terms of user-perceived latency, in spite of it worsens the hit ratio of the cache.
- Our solution shows greater improvement in the hardware architectures in which the assembling process overheads are higher.

Table 7.14. Cache performance for the executions of the experiment subset 2C.

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<entire/∅>`	93.7201	0.5788	91.0390	0.4781	950
`<entire/requestRate>`	85.3192	0.0345	81.0745	0.2341	1050
`<ratio/∅>`	80.1734	0.7100	75.3391	0.8001	950
`<ratio/updateRate>`	85.2810	0.9176	79.9347	0.7342	1000
`<ratio/size>`	86.3347	0.3748	82.9455	0.2841	950
`<difference/∅>`	88.2174	0.4117	85.4700	0.4318	1000
`<split>`	96.1629	0.6174	92.0072	0.7120	900

- The adaptive cores of the experiments with the greatest latency are implemented with *entire/∅* and *ratio/∅*.
- The latency improvements of our proposal are independent from the improvements or the decrease of the cache hit ratios.
- Our solution obtains greater improvements as the user load level gets higher.
- The size and coverage values of the decision trees are not clearly related to the improvement obtained in the execution of the experiments.

7.3.3 Third experiment set

The goal of the experiment set 3 (Table 6.6) is the same that the previous one, but with important changes in the experiment setup. The page model and the user load levels are different in this case. This experiment is focused on the study of the decision trees behaviour for the *distributed* hardware architecture using the *pageflakes* content page model. But this experiment is a *dynamic* one, another important difference with the previous experiment.

Dynamic experiments are the cases in which changes in the characteristics of the content elements occurs. This means that content elements do not have the same values for their sizes, update rates, request rates, number of aggregations and number of aggregators during the execution of the emulation. This supposes a continuous re-evaluation of the aggregation state relationships. Thus, the classification process should be executed continuously during the emulation. This adds overhead times to the system and it also creates partial transient states between the point in time that the fragment design is changed and the new fragment elements are requested and stored in the cache. *Dynamic* experiments better reflect a real system and the improvement of our solution should be reduced in comparison with *static* ones.

As a consequence of the changes in the content elements (structure and size), during the experiment, we cannot compare the latencies of a given web pages in different points in time. In this case, the latencies do not only vary because of the cached elements or the defined fragment elements, but also because of the changes of the web pages themselves. The results are expressed as the mean values of each request among the replicas of the same experiment. Therefore, the x-axis of the plots, and consequently the points of the plots, corresponds to requests instead of web pages. Therefore, the plots of the *dynamic* experiments have 36.000 points.

This experiment set is divided into two experiment subsets, one with high user load level and another one with low level. The aim is to validate the previous hypothesis about our solution obtains greater improvements as the load of the system increases. In this experiment, we study the 11 decision trees corresponding to the *distributed* hardware architecture. We monitored the classification results during the experiment executions and we observed that the edges were classified in different states.

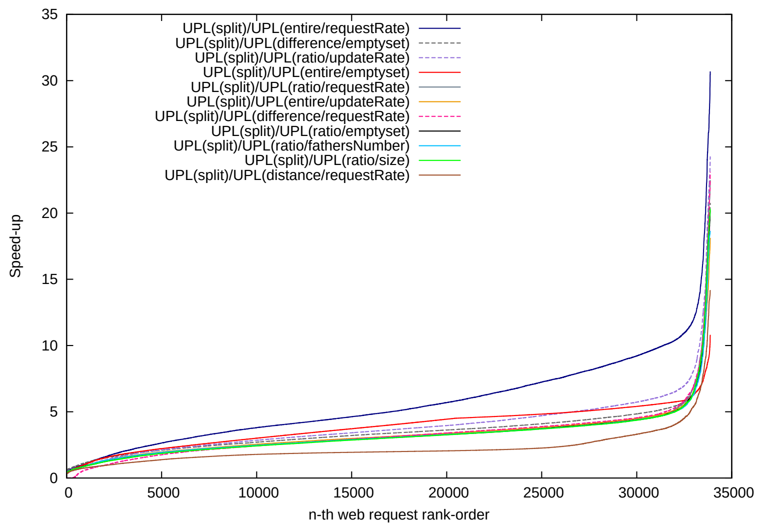
Table 7.15. Speed-ups of the user-perceived latency for experiment subset 3A.

Speed-up = UPL(`<split>`)/UPL(`<ExId>`)

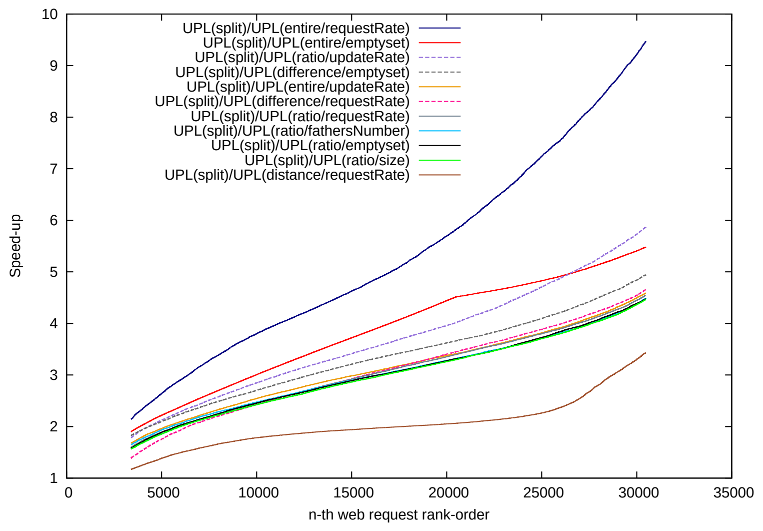
Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<entire/`$\emptyset$`>`	3.8487	1.8998–5.4800	3.8734
`<entire/updateRate>`	3.2594	1.6771–4.5889	3.1283
`<entire/requestRate>`	5.6229	2.1410–9.4709	5.3264
`<ratio/`$\emptyset$`>`	3.1904	1.5762–4.4749	3.0352
`<ratio/updateRate>`	3.8999	1.7841–5.8664	3.6933
`<ratio/requestRate>`	3.2637	1.5942–4.5418	3.0778
`<ratio/size>`	3.1780	1.5580–4.4675	3.0114
`<ratio/fathersNumber>`	3.1874	1.6480–4.4869	3.0458
`<difference/`$\emptyset$`>`	3.5199	1.8242–5.9422	3.3569
`<difference/requestRate>`	3.2481	1.3838–4.6575	3.0916
`<distance/requestRate>`	2.2290	1.1688–3.4317	2.0417

Figure 7.9 and Table 7.15 show the results of the latency speed-ups for the experiment subset 3A. For a better analysis, the identifiers of the figure legend are ordered as the mean values presented in the table. In previous experiments, *entire/*$\emptyset$ and *ratio/*$\emptyset$ seem to be the best decision trees to implement the classification algorithm of the adaptive core. But in this experiment, the improvements of the second tree are reduced considerably. On the contrary, the first tree, *entire/*$\emptyset$, is again in the group of the trees which obtain the highest performances, mainly if we analysed the mean improvement values for the P_{10}-P_{90} samples.

Comparing the results with previous experiments, we observe that the general speed-ups are smaller than in experiment 2A. This is not only explained by the differences in the user load level, but also because experiment 3A is a *dynamic* one. Anyway, the general improvement for experiment 3A is big-



(a) 100% of the samples



(b) P_{10}-P_{90} of the samples

Fig. 7.9. Speed-up of the user perceived latency in experiment subset 3A.

Table 7.16. Cache performance for the executions of the experiment subset 3A.

Experiment identification	Cache hit ratio (%) Mean	Cache hit ratio (%) Deviation	Cache byte hit ratio (%) Mean	Cache byte hit ratio (%) Deviation	Samples in transient state
`<entire/`$\emptyset$`>`	87.9668	0.9471	86.1132	0.5178	2100
`<entire/updateRate>`	87.7047	0.5113	83.1533	0.3178	1800
`<entire/requestRate>`	77.5086	1.0079	73.1432	0.7911	1500
`<ratio/`$\emptyset$`>`	83.9661	0.3941	80.9942	0.3314	1750
`<ratio/updateRate>`	83.6284	0.4077	80.8725	0.2834	1800
`<ratio/requestRate>`	84.0230	0.9811	81.3173	0.7761	1850
`<ratio/size>`	83.6467	0.3411	79.7570	0.0847	2000
`<ratio/fathersNumber>`	90.9688	0.1947	88.9542	0.1200	2050
`<difference/`$\emptyset$`>`	83.8249	0.3361	80.3958	0.1124	1650
`<diff./requestRate>`	78.3011	0.9714	75.2860	0.8140	1450
`<dist./requestRate>`	80.5348	0.2111	78.3667	0.1392	1800
`<split>`	91.2277	0.6006	88.7846	0.4381	1000

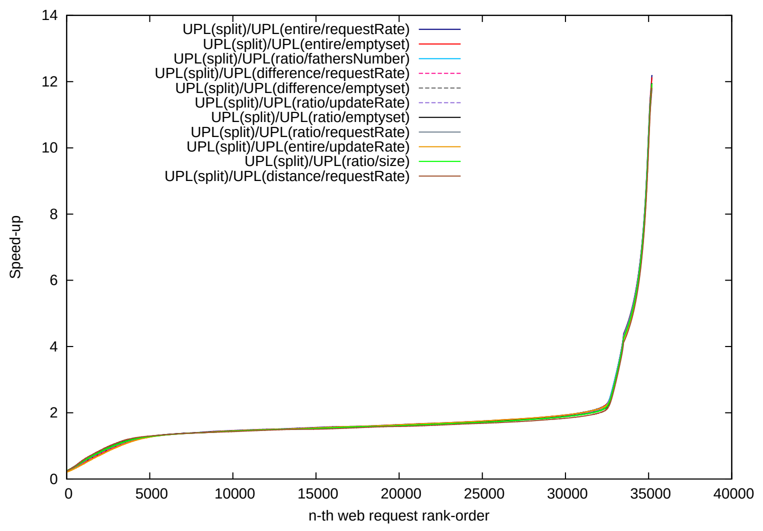
ger than for experiment 2C, which is *static*, but with a *centralized* hardware architecture. It seems that the benefits of our solution are more affected by the assembling overhead times than by the *dynamism* of the characterization parameters, the overheads generated by the classifications and the partial transient states.

The cache metrics of this experiment (Table 7.16) present significantly smaller hit ratios than, for example, the corresponding to experiments 2A and 2B. This is not only due to the creation of bigger fragments, but also because there are more update requests in experiment 3A than in the other two.

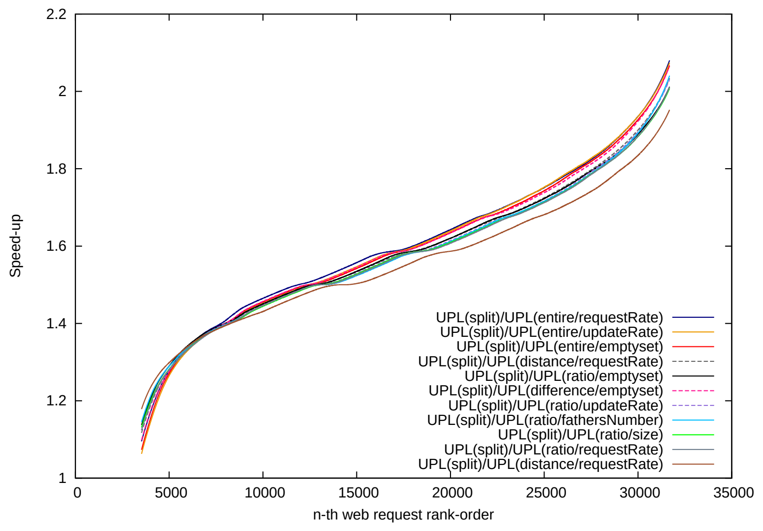
As in the previous experiments, *split* experiment shows the best hit ratios. Once again, there is not a relationship between the hit ratios of the experiments based on decision trees and the latency improvements obtained in these experiments.

The results of experiment 3B (Table 7.17 and Figure 7.10) show that the improvements of all the trees are now almost identical. For a better analysis, the identifiers of the figure legend are ordered as the mean values presented in the table. The significant differences of experiment 3A have disappeared between them, and the only variation between the experiments is the user load level. As in experiment 2B, the benefits of the use of the decision trees are reduced as the user load level is reduced. Therefore, we conclude that the differences in the improvements among the decision trees get equal as the user load level of the system decreases.

If we compare the improvement mean values of experiments 3A and 3B we also observe that the latency speed-ups are higher in the first one. This is explained again because of the difference in the user load level. We also detected this pattern in experiment set 2. This was entirely expected, because



(a) 100% of the samples



(b) P_{10}-P_{90} of the samples

Fig. 7.10. Speed-up of the user perceived latency in experiment subset 3B.

Table 7.17. Speed-ups of the user-perceived latency for experiment subset 3B.

Speed-up = UPL(`<split>`)/UPL(`<ExId>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<entire/`$\emptyset$ `>`	1.8134	1.0724–2.0665	1.5979
`<entire/updateRate>`	1.7985	1.0618–2.0733	1.6008
`<entire/requestRate>`	1.8248	1.0943–2.0797	1.6065
`<ratio/`$\emptyset$ `>`	1.7995	1.1365–2.0114	1.5877
`<ratio/updateRate>`	1.8014	1.1241–2.0305	1.5836
`<ratio/requestRate>`	1.7993	1.1230–2.0106	1.5797
`<ratio/size>`	1.7959	1.1305–2.0074	1.5802
`<ratio/fathersNumber>`	1.8067	1.1446–2.0401	1.5831
`<difference/`$\emptyset$ `>`	1.8014	1.1162–2.0337	1.5877
`<difference/requestRate>`	1.8065	1.0972–2.0652	1.5976
`<distance/requestRate>`	1.7682	1.1778–1.9519	1.5595

the overhead times of the assembling process (transmission times, database connections, etc.) are increased as the load of the system increases. And our solution shows greater improvements in environments with long overhead times.

The values of the web cache metrics are presented in Table 7.18. In experiments 3A and 3B, the user behaviour model is the same, but the cache ratios have some variations. The user load levels are different, but the proportion of request and update rates keeps constant, so we should think that the cache ratios would be the same. The fragment design of the content elements and web pages has a great influence on the cache ratios, and this is the explanation for the changes in the cache ratios between 3A and 3B. 3A shows higher cache performance. If we compare the hit ratios among the experiment executions of this experiment, the pattern of previous experiments is shown again.

The results of the third experiment set help us to conclude, reinforce or validate that:

- Our solution improves the performance, in terms of user-perceived latency, in spite of it worsens the hit ratio of the cache.
- Our solution obtains greater improvements as the load level gets higher.
- Our solution shows greater improvement in the hardware architectures in which the assembling process overheads are higher.
- Our solution shows greater improvement in scenarios in which there are not changes in the characterization parameters of the content elements.
- The reduction of the benefits of our solution, in terms of user-perceived latency, is more influenced by the reduction of the assembling overhead times than by the increase of the number of the classification executions.
- The *entire/*$\emptyset$ is finally considered as the decision tree which best implements the classification algorithm of the adaptive core in order to achieve the best latencies speed-ups. The *ratio/*$\emptyset$ is finally rejected as one of the

Table 7.18. Cache performance for the executions of the experiment subset 3B.

Experiment identification	Cache hit ratio (%) Mean	Cache hit ratio (%) Deviation	Cache byte hit ratio (%) Mean	Cache byte hit ratio (%) Deviation	Samples in transient state
`<entire/`$\emptyset$ `>`	86.8704	0.4789	84.6294	0.3441	750
`<entire/updateRate>`	86.1267	0.9471	81.0551	0.7134	650
`<entire/requestRate>`	78.3043	0.0542	74.0385	0.2471	700
`<ratio/`$\emptyset$ `>`	83.4756	0.3599	80.5949	0.5714	600
`<ratio/updateRate>`	82.3988	0.3298	78.5299	0.4754	700
`<ratio/requestRate>`	82.5616	0.1794	79.5415	0.3554	750
`<ratio/size>`	80.4010	0.9778	78.4936	1.0741	700
`<ratio/fathersNumber>`	89.6449	0.9471	87.9965	0.7993	750
`<difference/`$\emptyset$ `>`	82.3796	0.7913	79.8796	0.8519	800
`<diff./requestRate>`	78.0753	0.5863	73.9665	0.7465	800
`<dist./requestRate>`	79.9374	0.1685	77.9502	0.3500	700
`<split>`	90.1339	0.7438	87.4143	0.5547	600

best decision trees because its low performance in the last set of experiments.

- The latency improvements of our proposal are independent from the improvements or the decrease of the cache hit ratios.
- The differences of the improvements of our proposed adaptive core, among the decision trees, are reduced as the load level of the system is reduced.
- The cache hit ratios are not affected only by the proportion of the update and read request and the system load level, but also by the design of the fragment elements.

As a consequence of these results, the adaptive core is going to be implemented only by the use of the *entire/*$\emptyset$ decision tree in the rest of the experiments. We have contrasted that it is the best, or one of the best, trees in most of the cases.

7.3.4 Fourth experiment set

The experiment set 4 is designed to study how the benefits of our approach change as the system load level is modified, in comparison with the *split* traditional scheme (Table 6.7). All the experiment executions of this experiment set are identical except for the system load level, *i.e.*, the request and update rates. The request rate has been varied from 2 req./s up to 10 req./s in steps of 2. The update rate has been varied from 0.5 req./s up to 2.5 req./s in steps of 0.5. Thus, the experiment is composed of 25 executions where our solution is used, and 25 pairs of executions. The *entire/*$\emptyset$ decision tree is used to implement the adaptive core of the framework in all of the executions using our solution.

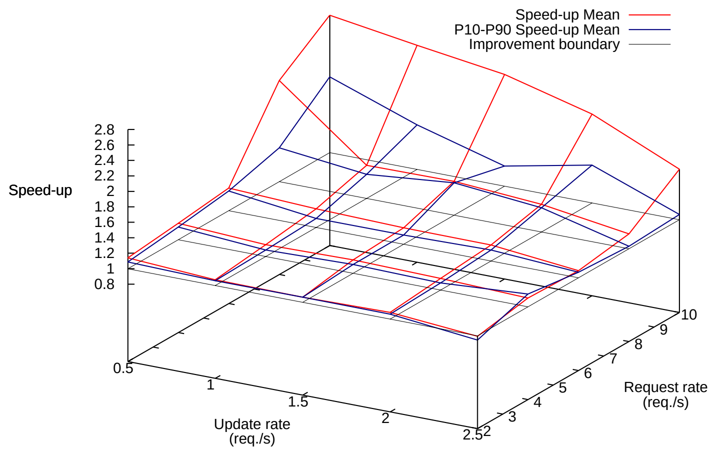


Fig. 7.11. Speed-up of the user perceived latency in experiment subset 4A.

The experiment has been designed in order to reproduce an environment (web system) with the features where our solution presents the worst behaviour. This means that we have decided to execute this experiment set in a *dynamic* system with the *centralized* hardware architecture. Since the solution has experimented similar improvements when the *pageflakes* and *nytimes* have been used, we have arbitrarily selected the first of them. If our solution shows improvement under this execution conditions, we ensure that our solution is validated to be used in CAS systems.

Due to the high number of experiments, we have decided not to present the speed-up values corresponding to each request of each experiment execution. We have previously observed that, in general terms, an experiment in which the mean value of all the request speed-ups is bigger than in other experiment, the single speed-ups of the first one are also bigger for almost all the requests. It means that the mean value of all the speed-ups is representative of the improvements of an experiment. Therefore, we are going to show only the mean values of all the requests, but also the means of the requests inside P_{10}-P_{90}, for each of the 50 experiment executions of this experiment set. These results are in Figure 7.11 and Table 7.19.

Observing the graphical results (Figure 7.11), it is easy to check that the improvements of our solution get great as the request rate is increased. This increase is observed in any of the studied update rates. The increase is more exacerbated for the two highest request rates (8 and 10 req./s). The difference between the mean values of all the requests and the P_{10}-P_{90} requests is also increased with the two highest request rates. This reflects that there is a high number of *positive* extreme values, requests with highest speed-ups, at these request levels.

On the contrary, the update rates increases affect inversely to the speed-ups improvements. When the update rate gets higher, the improvement showed by our solution is smaller. Anyway, this decrease is smaller than the observed with the increases of the request rates. In the case of having more updates than requests (request rate = 2.0 req./s and update rate = 2.5 req./s) our solution does not experiment any improvement (speed-ups equal to 1.0), even it shows longer latencies than the *split* traditional scenario.

Table 7.19. Speed-ups of the user-perceived latency for experiment subset 4A.

Speed-up = UPL(`<split,*/*/s`$^{-1}$`>`)/UPL(`<ExId>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<entire/∅,2.0/0.5/s`$^{-1}$`>`	1.1409	0.8829–1.3989	1.0877
`<entire/∅,2.0/1.0/s`$^{-1}$`>`	1.0738	0.8254–1.3222	1.0621
`<entire/∅,2.0/1.5/s`$^{-1}$`>`	1.0668	0.7993–1.3342	1.0668
`<entire/∅,2.0/2.0/s`$^{-1}$`>`	1.0856	0.7517–1.3679	1.0633
`<entire/∅,2.0/2.5/s`$^{-1}$`>`	0.9946	0.6745–1.3148	0.9450
`<entire/∅,4.0/0.5/s`$^{-1}$`>`	1.2129	0.8728–1.5531	1.1638
`<entire/∅,4.0/1.0/s`$^{-1}$`>`	1.1531	0.8389–1.4672	1.0835
`<entire/∅,4.0/1.5/s`$^{-1}$`>`	1.1729	0.8276–1.5181	1.1134
`<entire/∅,4.0/2.0/s`$^{-1}$`>`	1.1500	0.7851–1.4691	1.0921
`<entire/∅,4.0/2.5/s`$^{-1}$`>`	1.1131	0.7665–1.5157	1.1647
`<entire/∅,6.0/0.5/s`$^{-1}$`>`	1.2933	0.8387–1.6717	1.2552
`<entire/∅,6.0/1.0/s`$^{-1}$`>`	1.2407	0.8339–1.6476	1.1170
`<entire/∅,6.0/1.5/s`$^{-1}$`>`	1.2122	0.8298–1.5947	1.1195
`<entire/∅,6.0/2.0/s`$^{-1}$`>`	1.2072	0.7930–1.6114	1.1421
`<entire/∅,6.0/2.5/s`$^{-1}$`>`	1.0912	0.7560–1.5975	1.0711
`<entire/∅,8.0/0.5/s`$^{-1}$`>`	2.3077	0.8018–2.0737	1.4377
`<entire/∅,8.0/1.0/s`$^{-1}$`>`	1.4288	0.7625–1.8609	1.3117
`<entire/∅,8.0/1.5/s`$^{-1}$`>`	1.4381	0.7706–2.0667	1.4186
`<entire/∅,8.0/2.0/s`$^{-1}$`>`	1.3575	0.7540–1.7997	1.3134
`<entire/∅,8.0/2.5/s`$^{-1}$`>`	1.1926	0.7297–1.8129	1.0328
`<entire/∅,10.0/0.5/s`$^{-1}$`>`	2.7754	0.9001–3.2119	1.9801
`<entire/∅,10.0/1.0/s`$^{-1}$`>`	2.6032	0.8836–2.8227	1.5757
`<entire/∅,10.0/1.5/s`$^{-1}$`>`	2.4442	0.2629–2.2580	1.2604
`<entire/∅,10.0/2.0/s`$^{-1}$`>`	1.1485	0.3809–3.0552	1.4892
`<entire/∅,10.0/2.5/s`$^{-1}$`>`	1.6526	0.3417–3.0814	1.0680

The decrease of the improvement of our solution when the update rate gets higher is logical because of the behaviour of the cache ratios. When the update rate is increased, the cache hit ratios are reduced.

In Tables 7.21 and 7.22 and Figure 7.12, we observe how the differences between the cache ratios, either hit ratio and byte hit ratio, of the *split* solution and the solution based on the adaptive core are increased when the update rate is increased. The differences are also increased when the number of update requests, in proportion to the number of read requests, is bigger. Therefore, our solution experiments more significant reductions of hit ratios when the update rate is increased.

The reduction of the hit ratios is bigger in the case of our solution because the fragment elements are bigger than in the *split* scenario. Part of this increase is counteracted, in terms of latency, with our solution.

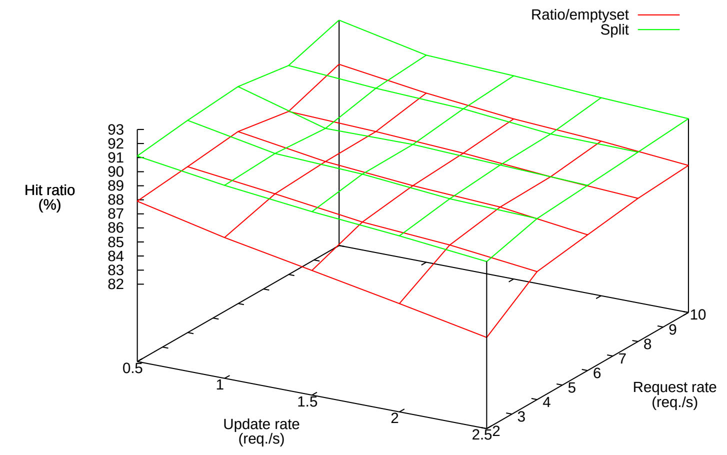
But as the update rate is increased, the difference gets greater and the weight of the benefit obtained with our solution gets smaller. This benefit is reduced to the level in which the losses in web cache ratios are more significant and, in consequence, our solution experiments bigger latencies than the traditional scenario.

The list of conclusions of this experiment are summarized as:

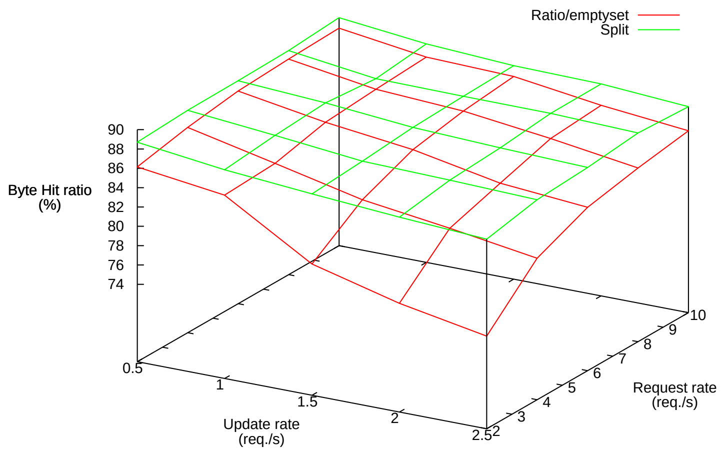
- Our solution shows shorter latencies independently of the load level of the system. The latencies of the web pages using our solution get equal to the latencies in the *split* scenario when the update rate and the request rate are almost equal.
- The improvements of our solution are greater as the request rate is increased and the update rate is decreased.
- The reductions of the hit ratios are bigger when our approach is used.
- Our solution counteracts better the effect of the loss of hit ratio, compared with the *split* scenario, when the update rates are low and the request rates are high.
- The improvement obtained with our solution is not associated to the web cache hit ratios.

7.3.5 Fifth experiment set

The last experiment set is addressed to compare our solution with a solution proposed by other researches. We have selected MACE, proposed in [50, 48], because it is the only research work, from our knowledge, that also bases the problem of web caching on CAS system on defining adaptive fragments of the content of a web page. They try to find cacheable points using a cost function. Cacheable points are vertices of the ODGex web page model for which their content and the content of their children are cached all together. The cost functions are based on the cost of caching a part of the web page, the cost of retrieve the content from the web server and the cost of keeping the consistence of the contents in the cache. Their solution recursively calculates



(a) Hit ratio



(b) Byte hit ratio

Fig. 7.12. Cache performance for the executions of the experiment subset 4A.

the cost associated to each vertex and it decides which ones are the best from a performance point of view.

The experiment set 5 (Table 6.8) has been designed with the same features of the previous one. We have selected the hardware architecture more unfavourable, the *centralized* one, in a *dynamic* execution context. The web page model was provided by the authors of MACE. We considered using their web page model in order to compare with a third one. This web page model is based on *Yahoo! Pipes*, a personal data mashup that aggregates web feeds, web pages, and other services, creating Web-based apps from various sources. The latencies of both approaches have been studied at different system load levels by changing the request and update rates in the same way than in experiment set 4.

As in 4A, we have decided to show only the global speed-up values instead of the speed-ups of each single request due to the high number of experiment executions of this set. The compared latencies of both experiments are shown in Figure 7.13 and Table 7.20. For these cases, the latencies of our proposed solution are compared with the latencies obtained with MACE instead of the *split* scenario. Thus, the common and reference latency values —the values of the numerator of the division to calculate the speed-up— correspond to the MACE executions.

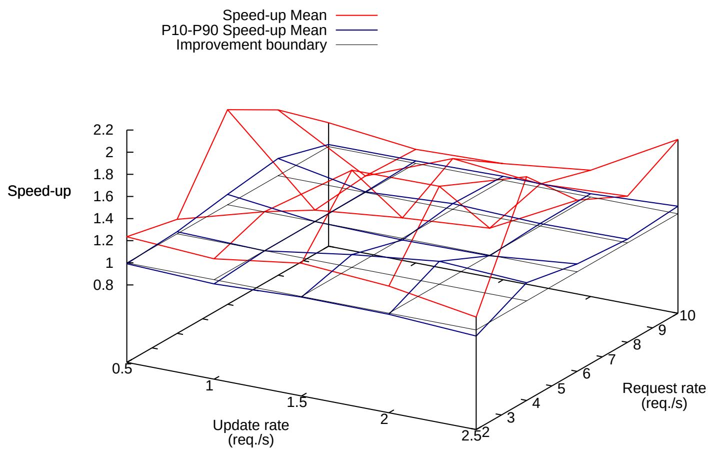


Fig. 7.13. Speed-up of the user perceived latency in experiment subset 5A.

By the analysis of the results of the experiment, we observe that our approach obtains shorter latencies for all the values of the system load. There is not a pattern between the values of request and update rates and the values of the speed-up, but the speed-up is always higher than 1.0, so our solution is better in terms of user-perceived latency.

But the improvements are reduced if we considered only the P_{10}-P_{90} samples instead of all the samples. This is explained because the extreme cases —the web pages with highest and smallest speed-ups— are more favourable for our solution. Anyway, most of the P_{10}-P_{90} mean values are bigger than 1.0, and the smaller ones are very close to this frontier value. There are only three cases with significant losses: `2.0/1.0/s`$^{-1}$, `2.0/2.5/s`$^{-1}$ and `6.0/1.5/s`$^{-1}$. There are six cases in which our solution is less than 1% faster or slower. And finally, for the other 15 cases, our solution is better. It is important to remind that the design experiment has been done taking into account the most unfavourable conditions for our approach.

Table 7.20. Speed-ups of the user-perceived latency for experiment subset 5A.

Speed-up = UPL(`<mace,*/*/s`$^{-1}$`>`)/UPL(`<ExId>`)

Execution id. (ExId)	Mean	P_{10}-P_{90} range	P_{10}-P_{90} mean
`<entire/∅,2.0/0.5/s`$^{-1}$`>`	1.2362	0.6269–1.3370	0.9920
`<entire/∅,2.0/1.0/s`$^{-1}$`>`	1.1880	0.6361–1.2877	0.9619
`<entire/∅,2.0/1.5/s`$^{-1}$`>`	1.3004	0.6849–1.3023	0.9936
`<entire/∅,2.0/2.0/s`$^{-1}$`>`	1.1886	0.7092–1.2629	0.9960
`<entire/∅,2.0/2.5/s`$^{-1}$`>`	1.1173	0.7131–1.1769	0.9450
`<entire/∅,4.0/0.5/s`$^{-1}$`>`	1.1312	0.6913–1.3467	1.0190
`<entire/∅,4.0/1.0/s`$^{-1}$`>`	1.3513	0.6272–1.3683	0.9978
`<entire/∅,4.0/1.5/s`$^{-1}$`>`	1.8778	0.6210–1.6101	1.1156
`<entire/∅,4.0/2.0/s`$^{-1}$`>`	1.8900	0.6503–1.9427	1.2965
`<entire/∅,4.0/2.5/s`$^{-1}$`>`	2.1220	0.6320–1.6973	1.1647
`<entire/∅,6.0/0.5/s`$^{-1}$`>`	1.8585	0.7636–1.4240	1.0938
`<entire/∅,6.0/1.0/s`$^{-1}$`>`	1.1035	0.6444–1.3349	0.9997
`<entire/∅,6.0/1.5/s`$^{-1}$`>`	1.1850	0.5871–1.3614	0.9842
`<entire/∅,6.0/2.0/s`$^{-1}$`>`	1.3017	0.5602–1.4672	1.0137
`<entire/∅,6.0/2.5/s`$^{-1}$`>`	1.6478	0.5748–1.5675	1.0711
`<entire/∅,8.0/0.5/s`$^{-1}$`>`	1.5949	0.8197–1.4912	1.1555
`<entire/∅,8.0/1.0/s`$^{-1}$`>`	1.1484	0.6683–1.3428	1.0055
`<entire/∅,8.0/1.5/s`$^{-1}$`>`	1.4575	0.6518–1.4501	1.0509
`<entire/∅,8.0/2.0/s`$^{-1}$`>`	1.3023	0.5705–1.4181	1.0289
`<entire/∅,8.0/2.5/s`$^{-1}$`>`	1.4217	0.5247–1.5409	1.0328
`<entire/∅,10.0/0.5/s`$^{-1}$`>`	1.2166	0.5140–1.3569	1.0195
`<entire/∅,10.0/1.0/s`$^{-1}$`>`	1.1274	0.6903–1.3554	1.0228
`<entire/∅,10.0/1.5/s`$^{-1}$`>`	1.1496	0.6370–1.3617	1.0347
`<entire/∅,10.0/2.0/s`$^{-1}$`>`	1.3327	0.6241–1.4945	1.0593
`<entire/∅,10.0/2.5/s`$^{-1}$`>`	1.6718	0.5283–1.6078	1.0680

If we analysed the results for the hit ratio and the byte hit ratio (Figure 7.14 and Tables 7.23 and 7.22) we observe that MACE obtains higher ratios in both metrics than our solution, based on the use of decision trees. Indeed, the differences between both solutions are bigger in the case of the byte hit ratio.

The trend of the results shows better cache ratios when the update rate is reduced and the request rate is increased. This trend is shown in the results of both solutions and for both metrics. In any case, the influence of the update and request rate seems to be higher on the results of our solution. The cache performance results for MACE are more homogeneous.

Finally, in all the previous scenarios and experiments, the hit ratio has been higher than the byte hit ratio. But in the case of MACE, this has been swapped, and it shows higher byte hit ratios than hit ratios.

Therefore, MACE approach seems to have a higher performance from the point of view of the cache system. It also seems to be less influenced by the changes in the update and request rates.

The conclusions extracted from the analysis of the results of experiment set 5 are summarized as:

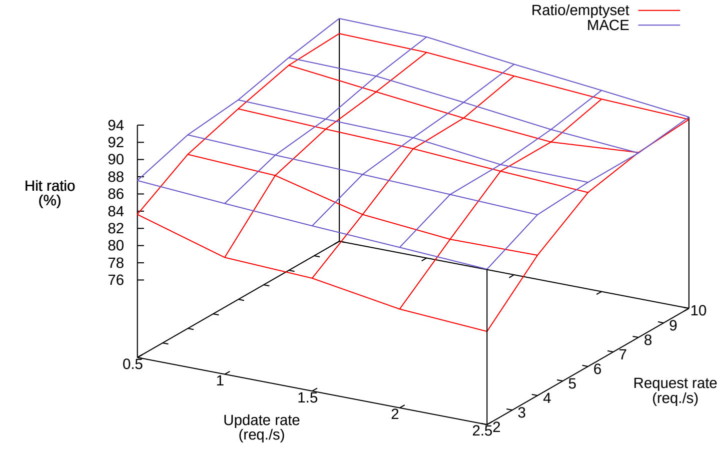
- Our solution is better, in terms of user-perceived latency, than MACE.
- It seems not to be a pattern between the update and request rates and the improvement of the latencies between our approach and MACE.
- Our solution is higher influenced, in its improvements, by the speed-ups obtained in the extreme cases.
- MACE solution is better, in terms of web cache performance, than our solution.
- Our approach obtains higher hit ratios than byte hit ratios.
- MACE solution obtains higher byte hit ratios than hit ratios.

7.4 Summary

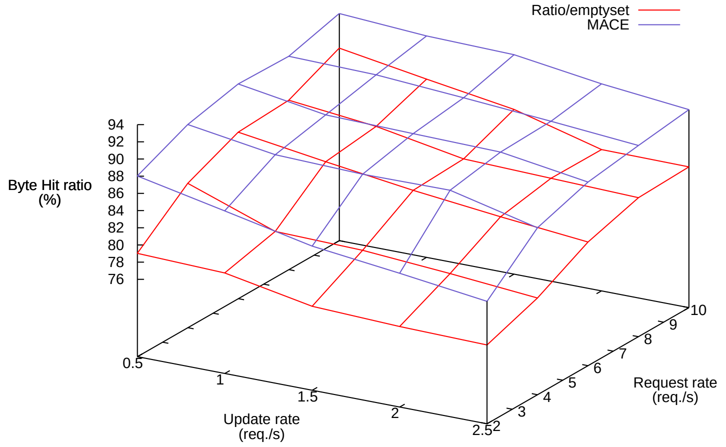
This section has been devoted to study and to analyse the results of the different experiment sets. We have studied the improvement in the user-perceived latency and the variations in the cache hit ratios. The latency improvement has been expressed as speed-ups, comparing the latencies when our solution is used to when a traditional scheme, or other solution, is used.

After the experiment analysis we conclude that, in general terms, our solution shows shorter user-perceived latency in comparison with the traditional CAS system cache schemes (*join* and *split*) and in comparison with other research solution, MACE approach. Therefore, it is validated that our proposal can be used to reduce the user-perceived latency in CAS system with a web cache.

But other important conclusions have been also done. In relation to our solution, we have concluded that it presents the best results when the times



(a) Hit ratio



(b) Byte hit ratio

Fig. 7.14. Cache performance for the executions of the experiment subset 5A.

of the assemblies are longer. The improvement of our solution is also greater when the load of the system, in terms of read request, is higher. On the contrary, when the update rates are increased, our solution shows smaller improvements in comparison with other solutions. In any case, our solution shows shorter latencies, than the traditional scenarios. And, obviously, our solution obtains greater improvement when there are not changes in the characterization parameters of the content elements, because it means that transient cache state does not appear.

As it was expected, the improvement in the latency of our solution is obtained in spite of a reduction in the cache hit ratios. This is because the reduction of the overhead times counteracts the losses in the hit ratios. It had been also shown that the improvement of our solution is not due to the changes in the cache hit ratios, thus the changes in the fragment designs help to reduce the latencies.

As conclusions from the exploring of several decision trees to implement the classification algorithm of the adaptive core, we have observed that, in general terms, the experiment executions with the best results use the *entire/∅* decision tree. This is the decision tree which uses all the independent attributes and expresses them as entire values.

We have also observed that the differences of improvement among the decision trees are reduced when the load of the system, in terms of request rate, is reduced. We have not been able to conclude anything about the relation between the sizes and coverages of the decision trees and the improvement obtained with them. This point remains as part of the future work.

After comparing our solution with other approach (MACE), we concluded that our solution is better than MACE in terms of user-perceived latency. On the contrary, MACE improves the cache hit ratio and byte hit ratio in comparison with our solution.

These analysis conclusions are used to validate a part of the contributions of our dissertation. Firstly, we have validated the use of a data mining and decision trees. We have reduced the user-perceived latencies, in comparison with other solution and traditional schemes, in a several number of experiments. These experiments have used a knowledge extracted from a previous data mining process to classify the aggregation relationships and, in consequence, to create adaptive fragment designs. The knowledge has been extracted from an off-line mining of performance data obtained from the emulation, in a real system, of synthetic web page content models. We have represented the extracted knowledge using decision trees.

Consequently, it is also validated the guidelines that we have defined to create the synthetic content model. As the latency results are better than using other solutions, all the processes and phases of our solution have been validated.

Finally, the use of the five characterization parameters as inputs of our adaptive core is suitable because the experiments with the best results are those based in the *entire/∅* decision tree. This decision tree uses all the attributes that we initially considered, so the use of all of them is validated. This is reinforced because the second best decision tree, *ratio/*$\emptyset$, also uses all the independent attributes. Concerning to the representation of the independent attributes, it seems that the *entire* representation is the best one, and it is explained because this representation has a higher amount of information.

Table 7.21. Cache performance for the executions of the experiment subset 4A (*entire/∅* experiments).

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<ent./∅,2.0/0.5/`s^{-1}`>`	87.9073	0.2405	86.1467	0.4159	2050
`<ent./∅,2.0/1.0/`s^{-1}`>`	86.5095	0.2237	84.9632	0.3399	1800
`<ent./∅,2.0/1.5/`s^{-1}`>`	85.3633	0.3693	79.5733	0.3672	2100
`<ent./∅,2.0/2.0/`s^{-1}`>`	84.2012	0.2583	77.2480	0.3333	2300
`<ent./∅,2.0/2.5/`s^{-1}`>`	82.9833	0.5630	75.5893	0.4989	2150
`<ent./∅,4.0/0.5/`s^{-1}`>`	88.2916	0.4146	87.2279	0.3348	1950
`<ent./∅,4.0/1.0/`s^{-1}`>`	87.5529	0.3073	85.2386	0.4545	2000
`<ent./∅,4.0/1.5/`s^{-1}`>`	86.7389	0.3100	83.2200	0.1466	1950
`<ent./∅,4.0/2.0/`s^{-1}`>`	86.3049	0.6550	82.0025	0.2186	2150
`<ent./∅,4.0/2.5/`s^{-1}`>`	85.6017	0.9401	80.6327	0.3242	2100
`<ent./∅,6.0/0.5/`s^{-1}`>`	88.7259	0.5020	87.9816	0.0516	1850
`<ent./∅,6.0/1.0/`s^{-1}`>`	87.7848	0.7428	86.4948	0.1813	2000
`<ent./∅,6.0/1.5/`s^{-1}`>`	87.2630	1.0404	85.3905	0.1921	2300
`<ent./∅,6.0/2.0/`s^{-1}`>`	86.9356	0.3095	83.6567	0.0767	2150
`<ent./∅,6.0/2.5/`s^{-1}`>`	86.1510	1.3893	82.8987	0.2710	1900
`<ent./∅,8.0/0.5/`s^{-1}`>`	88.0907	0.6220	88.2884	0.0263	1850
`<ent./∅,8.0/1.0/`s^{-1}`>`	87.8511	1.4210	86.9030	0.3709	2100
`<ent./∅,8.0/1.5/`s^{-1}`>`	87.5090	0.9250	86.3848	0.3195	2100
`<ent./∅,8.0/2.0/`s^{-1}`>`	87.0280	0.4516	85.2734	0.2297	2200
`<ent./∅,8.0/2.5/`s^{-1}`>`	86.6811	0.3412	83.9649	0.1436	2350
`<ent./∅,10.0/0.5/`s^{-1}`>`	89.3789	0.1013	88.4819	0.1217	1900
`<ent./∅,10.0/1.0/`s^{-1}`>`	88.5267	0.9974	87.2278	0.1624	2200
`<ent./∅,10.0/1.5/`s^{-1}`>`	87.8811	0.1395	86.9611	0.2411	2200
`<ent./∅,10.0/2.0/`s^{-1}`>`	87.4978	0.2694	85.7059	0.2460	1950
`<ent./∅,10.0/2.5/`s^{-1}`>`	86.9569	2.6826	84.8085	0.8700	2150

Table 7.22. Cache performance for the executions of the experiment subset 4A (*split* experiments).

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<split,2.0/0.5/s`$^{-1}$`>`	91.1180	0.0449	88.6983	0.0846	1500
`<split,2.0/1.0/s`$^{-1}$`>`	90.2182	0.0374	87.5806	0.1915	1450
`<split,2.0/1.5/s`$^{-1}$`>`	89.5391	0.0893	86.8236	0.1908	1500
`<split,2.0/2.0/s`$^{-1}$`>`	89.0105	0.0665	86.1445	0.0379	1650
`<split,2.0/2.5/s`$^{-1}$`>`	88.3754	0.0877	85.6045	0.0638	1450
`<split,4.0/0.5/s`$^{-1}$`>`	91.5884	0.3058	88.9954	0.0787	1400
`<split,4.0/1.0/s`$^{-1}$`>`	90.4253	0.1340	88.1310	0.0968	1750
`<split,4.0/1.5/s`$^{-1}$`>`	90.1478	0.0553	87.2015	0.1708	1600
`<split,4.0/2.0/s`$^{-1}$`>`	89.5832	0.0322	86.9786	0.1033	1600
`<split,4.0/2.5/s`$^{-1}$`>`	89.3718	0.0309	86.6848	0.0625	1350
`<split,6.0/0.5/s`$^{-1}$`>`	91.9219	0.4608	89.0489	0.3035	1400
`<split,6.0/1.0/s`$^{-1}$`>`	90.1368	0.0739	88.4896	0.0424	1400
`<split,6.0/1.5/s`$^{-1}$`>`	90.1908	0.4914	87.6860	0.3334	1650
`<split,6.0/2.0/s`$^{-1}$`>`	89.9198	0.3082	87.3131	0.2001	1350
`<split,6.0/2.5/s`$^{-1}$`>`	89.6652	0.3897	87.0172	0.1507	1650
`<split,8.0/0.5/s`$^{-1}$`>`	91.3576	0.4086	89.1636	0.4039	1450
`<split,8.0/1.0/s`$^{-1}$`>`	90.9293	0.0133	88.0000	0.1140	1500
`<split,8.0/1.5/s`$^{-1}$`>`	90.6597	0.2361	87.9094	0.0585	1350
`<split,8.0/2.0/s`$^{-1}$`>`	90.0634	0.0297	87.8712	0.1910	1550
`<split,8.0/2.5/s`$^{-1}$`>`	89.9648	0.1350	87.5804	0.0076	1450
`<split,10.0/0.5/s`$^{-1}$`>`	92.5196	0.6012	89.5644	0.9663	1550
`<split,10.0/1.0/s`$^{-1}$`>`	91.2229	0.3655	88.5968	0.5030	1500
`<split,10.0/1.5/s`$^{-1}$`>`	90.9475	0.1884	88.0739	0.0396	1350
`<split,10.0/2.0/s`$^{-1}$`>`	90.5797	0.3363	87.9152	0.5015	1400
`<split,10.0/2.5/s`$^{-1}$`>`	90.2808	0.5758	87.2997	0.5043	1550

Table 7.23. Cache performance for the executions of the experiment subset 5A (*entire/*∅ experiments).

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<ent./∅,2.0/0.5/s`$^{-1}$`>`	83.6303	0.9222	79.0315	0.5611	3800
`<ent./∅,2.0/1.0/s`$^{-1}$`>`	80.5801	0.7731	78.6977	0.1538	3200
`<ent./∅,2.0/1.5/s`$^{-1}$`>`	80.1115	1.2557	76.7795	0.5760	3650
`<ent./∅,2.0/2.0/s`$^{-1}$`>`	78.4802	1.2419	76.3739	0.1177	2800
`<ent./∅,2.0/2.5/s`$^{-1}$`>`	77.8283	0.9331	76.1694	0.0570	4050
`<ent./∅,4.0/0.5/s`$^{-1}$`>`	87.2272	0.7990	83.7999	0.2294	2550
`<ent./∅,4.0/1.0/s`$^{-1}$`>`	86.7239	0.7676	80.1560	0.2198	3400
`<ent./∅,4.0/1.5/s`$^{-1}$`>`	84.1320	0.8777	79.9110	0.6778	3150
`<ent./∅,4.0/2.0/s`$^{-1}$`>`	83.2060	0.8055	79.1531	0.2733	3100
`<ent./∅,4.0/2.5/s`$^{-1}$`>`	83.3061	0.8743	78.2638	0.1788	2950
`<ent./∅,6.0/0.5/s`$^{-1}$`>`	89.1376	1.0601	86.3714	0.1930	2300
`<ent./∅,6.0/1.0/s`$^{-1}$`>`	88.7332	1.0404	84.8833	0.0966	3900
`<ent./∅,6.0/1.5/s`$^{-1}$`>`	88.4086	1.1129	83.5024	0.5956	3550
`<ent./∅,6.0/2.0/s`$^{-1}$`>`	87.7449	0.9310	82.3807	0.0813	3650
`<ent./∅,6.0/2.5/s`$^{-1}$`>`	87.2129	1.1913	81.3921	0.3853	2350
`<ent./∅,8.0/0.5/s`$^{-1}$`>`	90.8181	0.5374	86.7150	0.0667	3500
`<ent./∅,8.0/1.0/s`$^{-1}$`>`	89.6862	1.1143	85.6215	0.0577	2750
`<ent./∅,8.0/1.5/s`$^{-1}$`>`	88.5838	1.0274	83.7840	0.0671	3000
`<ent./∅,8.0/2.0/s`$^{-1}$`>`	87.7599	1.0358	83.4851	0.2868	3650
`<ent./∅,8.0/2.5/s`$^{-1}$`>`	88.4823	1.1064	83.1705	0.2656	3150
`<ent./∅,10.0/0.5/s`$^{-1}$`>`	91.1267	1.3329	89.3921	2.0476	2950
`<ent./∅,10.0/1.0/s`$^{-1}$`>`	90.8931	0.9919	87.7407	0.0777	3500
`<ent./∅,10.0/1.5/s`$^{-1}$`>`	90.1011	0.9277	86.1600	0.2095	3050
`<ent./∅,10.0/2.0/s`$^{-1}$`>`	89.3721	1.0633	83.4427	0.0431	2850
`<ent./∅,10.0/2.5/s`$^{-1}$`>`	88.9520	0.8549	83.3835	0.3378	3600

Table 7.24. Cache performance for the executions of the experiment subset 5A (MACE experiments).

Experiment identification	Cache hit ratio (%)		Cache byte hit ratio (%)		Samples in transient state
	Mean	Deviation	Mean	Deviation	
`<mace,2.0/0.5/s`$^{-1}$`>`	87.5619	0.0736	88.0274	0.0106	3250
`<mace,2.0/1.0/s`$^{-1}$`>`	86.8467	0.0766	85.9120	0.0103	2350
`<mace,2.0/1.5/s`$^{-1}$`>`	86.2013	0.0924	83.7664	0.0201	1950
`<mace,2.0/2.0/s`$^{-1}$`>`	85.6476	0.0900	82.5640	0.0179	2050
`<mace,2.0/2.5/s`$^{-1}$`>`	85.0534	0.1155	81.2424	0.0264	2100
`<mace,4.0/0.5/s`$^{-1}$`>`	89.4871	0.1501	90.6407	0.0069	3150
`<mace,4.0/1.0/s`$^{-1}$`>`	89.0811	0.1800	89.0708	0.0661	2850
`<mace,4.0/1.5/s`$^{-1}$`>`	88.7910	0.1817	88.7544	0.0309	2050
`<mace,4.0/2.0/s`$^{-1}$`>`	88.4163	0.1521	88.8372	0.1058	3000
`<mace,4.0/2.5/s`$^{-1}$`>`	88.0120	0.1515	86.4253	0.0323	2600
`<mace,6.0/0.5/s`$^{-1}$`>`	90.1708	0.2230	91.9732	0.3303	2450
`<mace,6.0/1.0/s`$^{-1}$`>`	89.8920	0.2431	90.3323	0.0073	2300
`<mace,6.0/1.5/s`$^{-1}$`>`	89.6812	0.2751	90.0947	0.0599	2650
`<mace,6.0/2.0/s`$^{-1}$`>`	88.5104	0.2998	89.8981	0.0140	2250
`<mace,6.0/2.5/s`$^{-1}$`>`	88.4004	0.2610	88.3633	0.0393	3100
`<mace,8.0/0.5/s`$^{-1}$`>`	91.7053	0.2977	91.8149	0.9612	2400
`<mace,8.0/1.0/s`$^{-1}$`>`	91.5139	0.2817	91.6155	0.0111	2250
`<mace,8.0/1.5/s`$^{-1}$`>`	90.4159	0.3613	90.8873	0.2177	2100
`<mace,8.0/2.0/s`$^{-1}$`>`	89.2087	0.2962	90.1130	0.0078	3150
`<mace,8.0/2.5/s`$^{-1}$`>`	88.4278	0.3589	89.2480	0.0675	3000
`<mace,10.0/0.5/s`$^{-1}$`>`	92.8851	0.3070	93.4223	0.0740	3050
`<mace,10.0/1.0/s`$^{-1}$`>`	92.6833	0.4442	92.7626	0.0378	2400
`<mace,10.0/1.5/s`$^{-1}$`>`	91.4636	0.3537	92.5208	0.0202	2200
`<mace,10.0/2.0/s`$^{-1}$`>`	90.3842	0.4906	91.0883	0.0329	2150
`<mace,10.0/2.5/s`$^{-1}$`>`	89.2310	0.4377	90.0382	0.0535	2700

8

System overhead analysis

It's hardware that makes a machine fast. It's software that makes a fast machine slow.
—Craig Bruce—

The improvements obtained with our solution are achieved due to an increase of the operations and calculations performed by the system. This additional workload, or overhead, is concentrated in the server tiers of the system. New operations are done in the web cache, the DBMS (Database Management System) and a new module is deployed, the adaptive core. In this chapter, we are going to study the workload generated by each of these software elements.

8.1 Introduction

This chapter is devoted to evaluate the overhead generated by our solution over the system, in terms of CPU utilization. The evaluation has been done over the same experiments than in the previous chapter. But, from the point of view of the overhead analysis, the experiments with *dynamic* schemes are more interesting. The *static* experiments only execute the classification process at the beginning of the execution. The *dynamic* experiments execute the classification algorithm every time that a change is detected in the characterization parameters of the content elements. Thus, it is more realistic to study the overhead in *dynamic* environments. Experiments sets 3, 4 and 5 are the *dynamic* ones. But we have studied only the two last sets because they are focussed on the study of several user load levels.

The data for the utilization study have been gathered using *top*. *Top* is a system monitoring tool which provides system summary information about the computer processes in execution. It provides the units of time that each process consumes of the CPU and the size of the memory assigned to each process. We have been executing *top* during the experiment executions in order to create file logs with usage information. We have gathered only information of the processes of the software involved in the deployment of the CAS system tiers, *i.e.*, the web server (*apache2* service), the web cache (*webcached* daemon), the DBMS (*mysqld* daemon) and the adaptive core (the java application).

Traditional workload studies analyse the CPU, I/O and memory consumption. In our case, all the processes had a fix size of memory, thus there were not changes in the memory size during the experiment executions, neither between the executions with different approaches. I/O accesses of CAS system are usually done over the tables of the databases. This type of load is better studied by the CPU utilization of the DBMS process instead of using the number of disk accesses. This is because DBMS usually has an important part of the databases tables in memory in order to improve the performance and, under these conditions, the accesses to the disk do not reflect the real workload over the DBMS. Therefore, we have studied only the CPU utilization.

The process to calculate the CPU utilization values was firstly to calculate the percentage of the total time of an experiment execution that a process occupied the CPU. Finally, we calculated the mean value among all the replicas of the same experiment. The standard deviation among the replicas of the same experiment has been also calculated. The analysis of the usage results is done from two points of view: on the one hand, it is done by considering the utilization percentage of a process in relation with the total time of the experiment execution, the global percentages; on the other hand, it is done by considering the percentage of the time of a process in relation with the sum of the times of the web cache, web server, DBMS and adaptive core, the particular percentages. These particular percentages show the weight of each process in the total usage time for the processes related to the CAS system.

This chapter is divided into two sections, one for each of both evaluated experiment sets. The two first parts of each section are devoted to the analysis of the evolution of the utilization values for one of the approaches (*split* scenario, adaptive core based on decision trees or MACE). The last parts of the subsections are devoted to compare the solutions between them.

8.2 Overhead analysis of the fourth experiment set

Our first evaluation is done over the executions of the experiment set 4. This experiment set is composed by experiments in which the *split* scenario and the *entire/∅* decision tree are used. Several experiments are created by the variation of values of the update and request rates.

The hardware architecture is *centralized*, which means that the web application, the web cache and the adaptive core are installed in the same computer. Therefore, the maximum value for the sum of the CPU utilization values of the processes is 100%.

The utilization percentage is given in two different ways. On the one hand, the results are presented as global percentage. The values indicates the real CPU consumption in terms of percentages, *i.e.*, the utilization values indicate the percentage of time that a given process has occupied the CPU in comparison with the total execution time of an experiment. The results corresponding to the global percentages are shown in Tables 8.1 and 8.2. On the other hand, the results are presented as particular percentages. The percentages are calculated in relation with the total time that the CPU has been occupied by some of the processes used to deploy the CAS system (web server, web cache, DBMS and adaptive core). Therefore, the sum of the particular percentages of the four processes is 100%. The values of this second case are presented in Tables 8.3 and 8.4.

The analysis of the utilization is going to be analysed in two different phases. Firstly, we are going to analyse the workload when the system load conditions change and, secondly, comparing the overhead among different approaches.

Table 8.1. Global utilization ratios (%) of the different application process for *entire/∅* in experiment subset 4A.

Experiment	Web Cache		Adaptive core		Apache		DBMS		Total
	μ	σ	μ	σ	μ	σ	μ	σ	
`<ent./∅,2.0/0.5/s`$^{-1}$`>`	0.990	0.110	0.161	0.020	0.007	0.003	0.422	0.019	1.581
`<ent./∅,2.0/1.0/s`$^{-1}$`>`	0.944	0.093	0.281	0.020	0.012	0.005	0.638	0.009	1.876
`<ent./∅,2.0/1.5/s`$^{-1}$`>`	1.012	0.107	0.379	0.023	0.032	0.021	0.864	0.011	2.289
`<ent./∅,2.0/2.0/s`$^{-1}$`>`	1.046	0.085	0.485	0.029	0.017	0.003	1.073	0.012	2.622
`<ent./∅,2.0/2.5/s`$^{-1}$`>`	1.021	0.038	0.663	0.091	0.013	0.003	1.256	0.020	2.954
`<ent./∅,4.0/0.5/s`$^{-1}$`>`	1.120	0.044	0.200	0.041	0.032	0.018	0.628	0.050	1.981
`<ent./∅,4.0/1.0/s`$^{-1}$`>`	1.093	0.056	0.328	0.041	0.043	0.018	0.865	0.029	2.331
`<ent./∅,4.0/1.5/s`$^{-1}$`>`	1.325	0.087	0.411	0.061	0.042	0.021	1.053	0.035	2.833
`<ent./∅,4.0/2.0/s`$^{-1}$`>`	1.249	0.120	0.425	0.025	0.042	0.014	1.194	0.027	2.912
`<ent./∅,4.0/2.5/s`$^{-1}$`>`	1.145	0.134	0.608	0.063	0.063	0.021	1.465	0.032	3.284
`<ent./∅,6.0/0.5/s`$^{-1}$`>`	1.387	0.019	0.313	0.063	0.124	0.054	0.831	0.015	2.655
`<ent./∅,6.0/1.0/s`$^{-1}$`>`	1.614	0.098	0.309	0.072	0.078	0.064	1.014	0.041	3.017
`<ent./∅,6.0/1.5/s`$^{-1}$`>`	1.651	0.084	0.502	0.082	0.312	0.259	1.310	0.077	3.776
`<ent./∅,6.0/2.0/s`$^{-1}$`>`	1.770	0.040	0.648	0.091	0.152	0.075	1.580	0.060	4.151
`<ent./∅,6.0/2.5/s`$^{-1}$`>`	1.644	0.080	0.527	0.025	0.072	0.021	1.593	0.041	3.837
`<ent./∅,8.0/0.5/s`$^{-1}$`>`	1.848	0.190	0.436	0.072	0.170	0.009	1.100	0.000	3.556
`<ent./∅,8.0/1.0/s`$^{-1}$`>`	1.680	0.286	0.559	0.057	0.325	0.023	1.485	0.069	4.051
`<ent./∅,8.0/1.5/s`$^{-1}$`>`	1.927	0.142	0.515	0.114	0.144	0.021	1.460	0.041	4.047
`<ent./∅,8.0/2.0/s`$^{-1}$`>`	2.078	0.011	0.561	0.104	0.228	0.075	1.690	0.111	4.557
`<ent./∅,8.0/2.5/s`$^{-1}$`>`	1.700	0.000	0.486	0.000	0.067	0.000	1.648	0.000	3.903
`<ent./∅,10.0/0.5/s`$^{-1}$`>`	2.280	0.132	0.397	0.125	0.175	0.052	1.280	0.072	4.133
`<ent./∅,10.0/1.0/s`$^{-1}$`>`	2.478	0.116	0.256	0.047	0.288	0.030	1.314	0.027	4.337
`<ent./∅,10.0/1.5/s`$^{-1}$`>`	2.485	0.005	0.489	0.007	0.289	0.068	1.714	0.005	4.978
`<ent./∅,10.0/2.0/s`$^{-1}$`>`	2.498	0.064	0.711	0.072	0.141	0.060	1.946	0.003	5.297
`<ent./∅,10.0/2.5/s`$^{-1}$`>`	2.659	0.000	1.069	0.000	0.159	0.000	2.495	0.000	6.384

Table 8.2. Global utilization ratios (%) of the different application processes for the *split* scenario in experiment subset 4A.

Experiment	Web Cache		Adaptive core		Apache		DBMS		Total
	μ	σ	μ	σ	μ	σ	μ	σ	
`<split,2.0/0.5/s`$^{-1}$`>`	0.965	0.007	0	0	0.026	0.016	0.318	0.013	1.308
`<split,2.0/1.0/s`$^{-1}$`>`	0.983	0.162	0	0	0.019	0.021	0.424	0.024	1.425
`<split,2.0/1.5/s`$^{-1}$`>`	1.061	0.048	0	0	0.015	0.013	0.571	0.020	1.647
`<split,2.0/2.0/s`$^{-1}$`>`	1.022	0.119	0	0	0.013	0.005	0.705	0.018	1.740
`<split,2.0/2.5/s`$^{-1}$`>`	1.018	0.051	0	0	0.223	0.298	0.827	0.011	2.067
`<split,4.0/0.5/s`$^{-1}$`>`	1.398	0.096	0	0	0.044	0.008	0.462	0.037	1.904
`<split,4.0/1.0/s`$^{-1}$`>`	1.242	0.051	0	0	0.050	0.011	0.632	0.022	1.924
`<split,4.0/1.5/s`$^{-1}$`>`	1.251	0.142	0	0	0.065	0.028	0.700	0.023	2.016
`<split,4.0/2.0/s`$^{-1}$`>`	1.268	0.000	0	0	0.054	0.025	0.878	0.022	2.199
`<split,4.0/2.5/s`$^{-1}$`>`	1.347	0.083	0	0	0.042	0.017	0.964	0.053	2.353
`<split,6.0/0.5/s`$^{-1}$`>`	1.751	0.040	0	0	0.091	0.007	0.567	0.026	2.409
`<split,6.0/1.0/s`$^{-1}$`>`	1.627	0.124	0	0	0.133	0.063	0.717	0.038	2.477
`<split,6.0/1.5/s`$^{-1}$`>`	1.663	0.102	0	0	0.396	0.371	0.935	0.052	2.994
`<split,6.0/2.0/s`$^{-1}$`>`	1.638	0.111	0	0	0.184	0.059	1.091	0.048	2.913
`<split,6.0/2.5/s`$^{-1}$`>`	1.594	0.150	0	0	0.184	0.041	1.102	0.054	2.880
`<split,8.0/0.5/s`$^{-1}$`>`	2.051	0.148	0	0	0.367	0.048	0.912	0.094	3.330
`<split,8.0/1.0/s`$^{-1}$`>`	1.844	0.173	0	0	0.162	0.011	1.042	0.046	3.049
`<split,8.0/1.5/s`$^{-1}$`>`	1.904	0.024	0	0	0.123	0.056	1.131	0.037	3.157
`<split,8.0/2.0/s`$^{-1}$`>`	2.039	0.132	0	0	0.583	0.333	1.312	0.014	3.934
`<split,8.0/2.5/s`$^{-1}$`>`	2.043	0.138	0	0	0.179	0.005	1.408	0.016	3.630
`<split,10.0/0.5/s`$^{-1}$`>`	2.876	0.359	0	0	0.239	0.021	1.278	0.139	4.393
`<split,10.0/1.0/s`$^{-1}$`>`	2.389	0.120	0	0	0.280	0.171	1.162	0.027	3.831
`<split,10.0/1.5/s`$^{-1}$`>`	2.399	0.115	0	0	0.339	0.163	1.314	0.074	4.052
`<split,10.0/2.0/s`$^{-1}$`>`	2.406	0.106	0	0	0.177	0.082	1.416	0.007	3.999
`<split,10.0/2.5/s`$^{-1}$`>`	2.404	0.195	0	0	0.369	0.135	1.551	0.158	4.324

8.2.1 Analysis of the workload generated by the adaptive core based on the use of decision trees

The results are firstly analysed using the utilization percentage in relation with all the processes of the system and (global utilizations), secondly, in relation with the utilization generated by the application of the CAS system (particular utilizations). The results for the global utilization are shown in Figure 8.1(a) and, for the particular utilizations, in Figure 8.1(b).

The first conclusion of observing the plots is that the web cache is the application which consumes more CPU resources; the second one is the database, followed by the adaptive core, and finally, the web server.

The usage generated by the web server is almost negligible although small increases are detected. These small increases are larger as the rates get higher.

Table 8.3. Particular utilization ratios (%) of the different application processes for *entire/∅* in experiment subset 4A.

Experiment	Web Cache	Adaptive core	Apache	DBMS
`<entire/∅,2.0/0.5/s`$^{-1}$`>`	62.617	10.222	0.443	26.718
`<entire/∅,2.0/1.0/s`$^{-1}$`>`	50.314	14.983	0.680	34.023
`<entire/∅,2.0/1.5/s`$^{-1}$`>`	44.245	16.584	1.407	37.764
`<entire/∅,2.0/2.0/s`$^{-1}$`>`	39.898	18.517	0.654	40.931
`<entire/∅,2.0/2.5/s`$^{-1}$`>`	34.582	22.438	0.448	42.532
`<entire/∅,4.0/0.5/s`$^{-1}$`>`	56.515	10.128	1.629	31.727
`<entire/∅,4.0/1.0/s`$^{-1}$`>`	46.917	14.081	1.878	37.124
`<entire/∅,4.0/1.5/s`$^{-1}$`>`	46.796	14.536	1.488	37.180
`<entire/∅,4.0/2.0/s`$^{-1}$`>`	42.898	14.625	1.456	41.020
`<entire/∅,4.0/2.5/s`$^{-1}$`>`	34.881	18.543	1.944	44.631
`<entire/∅,6.0/0.5/s`$^{-1}$`>`	52.225	11.810	4.675	31.290
`<entire/∅,6.0/1.0/s`$^{-1}$`>`	53.503	10.267	2.615	33.616
`<entire/∅,6.0/1.5/s`$^{-1}$`>`	43.740	13.302	8.262	34.696
`<entire/∅,6.0/2.0/s`$^{-1}$`>`	42.650	15.628	3.665	38.057
`<entire/∅,6.0/2.5/s`$^{-1}$`>`	42.856	13.749	1.883	41.512
`<entire/∅,8.0/0.5/s`$^{-1}$`>`	51.979	12.273	4.802	30.945
`<entire/∅,8.0/1.0/s`$^{-1}$`>`	41.484	13.801	8.044	36.670
`<entire/∅,8.0/1.5/s`$^{-1}$`>`	47.615	12.740	3.560	36.086
`<entire/∅,8.0/2.0/s`$^{-1}$`>`	45.599	12.311	5.004	37.086
`<entire/∅,8.0/2.5/s`$^{-1}$`>`	43.559	12.472	1.731	42.238
`<entire/∅,10.0/0.5/s`$^{-1}$`>`	55.177	9.6220	4.233	30.967
`<entire/∅,10.0/1.0/s`$^{-1}$`>`	57.135	5.9180	6.651	30.297
`<entire/∅,10.0/1.5/s`$^{-1}$`>`	49.923	9.8330	5.808	34.435
`<entire/∅,10.0/2.0/s`$^{-1}$`>`	47.162	13.424	2.676	36.737
`<entire/∅,10.0/2.5/s`$^{-1}$`>`	41.655	16.755	2.502	39.088

These web server utilization increases are more significant when the request rate increases. In fact, not clear trend is detected when the updates increase.

The trend observed in the web server is also presented in the web cache. The usage increases as the request rate increases. The update requests have not an important influence on the web cache CPU utilization. This behaviour is quite logical because these two tiers increase the amount of processing time as the number of requests that arrive to the system is bigger.

On the contrary, the adaptive core shows quite uniform CPU utilization values. This is because it does not depend on the number of requests (update or read ones) that arrive to the system. The resource consumption of the adaptive core depends more in the changes in the characterization parameters. And these changes are not related to the number of update or read request over the system.

The DBMS (Database Management System) utilization is not only generated by the database accesses that retrieve information about the content

Table 8.4. Particular Utilization ratios (%) of the different application processes for the *split* scenario in experiment subset 4A.

Experiment	Web Cache	Adaptive core	Apache	DBMS
<entire/∅,2.0/0.5/s^{-1}>	73.731	0	1.996	24.273
<entire/∅,2.0/1.0/s^{-1}>	68.966	0	1.302	29.732
<entire/∅,2.0/1.5/s^{-1}>	64.408	0	0.922	34.670
<entire/∅,2.0/2.0/s^{-1}>	58.744	0	0.740	40.516
<entire/∅,2.0/2.5/s^{-1}>	49.229	0	10.775	39.996
<entire/∅,4.0/0.5/s^{-1}>	73.449	0	2.290	24.261
<entire/∅,4.0/1.0/s^{-1}>	64.577	0	2.590	32.833
<entire/∅,4.0/1.5/s^{-1}>	62.055	0	3.228	34.717
<entire/∅,4.0/2.0/s^{-1}>	57.665	0	2.435	39.900
<entire/∅,4.0/2.5/s^{-1}>	57.227	0	1.791	40.982
<entire/∅,6.0/0.5/s^{-1}>	72.665	0	3.784	23.551
<entire/∅,6.0/1.0/s^{-1}>	65.703	0	5.354	28.943
<entire/∅,6.0/1.5/s^{-1}>	55.562	0	13.215	31.223
<entire/∅,6.0/2.0/s^{-1}>	56.233	0	6.308	37.459
<entire/∅,6.0/2.5/s^{-1}>	55.355	0	6.384	38.260
<entire/∅,8.0/0.5/s^{-1}>	61.588	0	11.022	27.390
<entire/∅,8.0/1.0/s^{-1}>	60.492	0	5.314	34.194
<entire/∅,8.0/1.5/s^{-1}>	60.303	0	3.886	35.811
<entire/∅,8.0/2.0/s^{-1}>	51.826	0	14.817	33.357
<entire/∅,8.0/2.5/s^{-1}>	56.270	0	4.940	38.790
<entire/∅,10.0/0.5/s^{-1}>	65.457	0	5.450	29.093
<entire/∅,10.0/1.0/s^{-1}>	62.369	0	7.300	30.331
<entire/∅,10.0/1.5/s^{-1}>	59.201	0	8.371	32.428
<entire/∅,10.0/2.0/s^{-1}>	60.151	0	4.435	35.414
<entire/∅,10.0/2.5/s^{-1}>	55.601	0	8.534	35.865

elements and web pages, but also by the database accesses to update and to retrieve the fragment designs. Every time a classification is done, the values of the characterization parameters are retrieved, and the states of the aggregation relations are updated. The system monitors are not able to differentiate among both types of workload generated over the DBMS. The increase in the DBMS utilization is due to the higher number of requests that arrives to the system, so that the overload generated by the adaptive core is quite constant.

The analysis of the particular utilizations also gives us some interesting information (Figure 8.1(b)). We observe that the values of the utilization for the adaptive core and web server remain quite uniform. But the increase of the update rates results in that the relative workload generated by the DBMS gets equal to the web cache. Therefore, the increase of the utilization of the DBMS when the update rate increases is more significant for the DBMS than for the web cache. On the contrary, when the increase occurs in the request rate, the DBMS and web cache increases are similar. Thus, the increases of both process consumptions have the same weight.

8.2.2 Analysis of the workload generated by *split* scenario

The behaviour of the *split* scenario is quite similar to our approach, but with an important difference (Figure 8.2). In the case of the *split* scenario the process of the adaptive core is not present because there is not adaptation of the fragment designs.

In general terms, all the processes increase their CPU consumption as the request rate is higher. Higher update rates produce that the three processes have different behaviours. First, the web cache keeps its CPU utilization in similar values. Second, the DBMS increases the utilization as the update ratio gets high. And, finally, the web server does not show a relation among the user load level and its resource consumption.

If we analysed the results corresponding to the CPU utilization particular percentages in Figure 8.2(b) —the percentage utilization of a process in relation with the total CPU consumption of only the processes corresponding to software used to deploy the CAS system—, the web server also experiments a strange behaviour not explained by the user load level. The database increases its weight in the total usage as the request rate and update rate are increased.

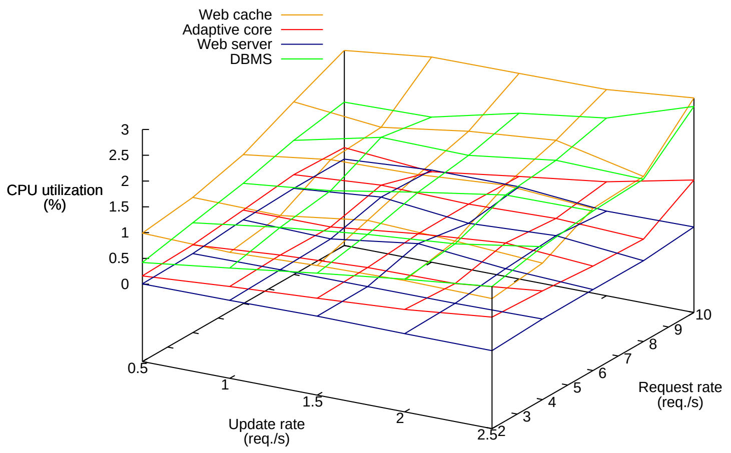
8.2.3 Comparison between *entire/∅* and the *split* scenario

In Figure 8.3, we observe the total CPU utilization values for the *split* scenario and the use of *entire/∅* decision tree. The CPU consumption is higher in the case of using the adaptive core. But the difference is very small. Thus, we consider that the increase of the overhead of our solution is counteracted by the improvement obtained in the user-perceived latency.

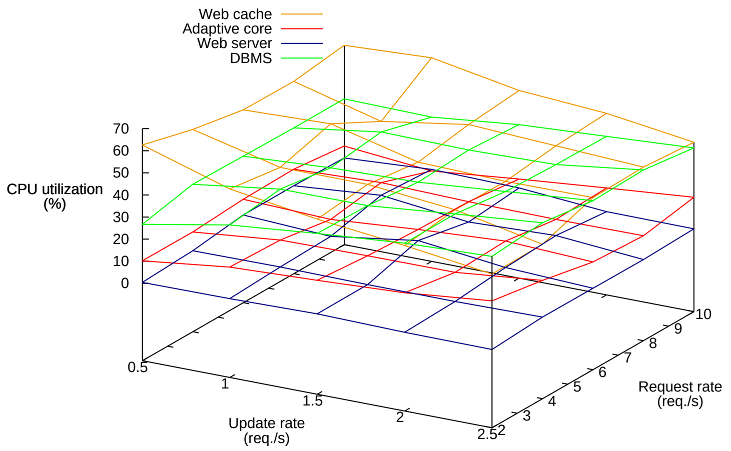
In general terms, we observe that the difference between both solutions increases as the update rate is increasing, but not with increases of the request rate. If we remind the results of the latency improvements, we conclude that our solution behaves better as the update rate is decreased, because the improvement is greater and the overhead is lower.

8.3 Overhead analysis of the fifth experiment set

The second evaluation of the system overhead generated by our solution is done by using the experiment set 5. In this experiment, the executions using our adaptive core implementation are compared with the case in which MACE is used. As in the previous experiment, the executions are changed in terms of update and request rates. The hardware architecture is also *centralized*. All the processes are executed in the same computer.

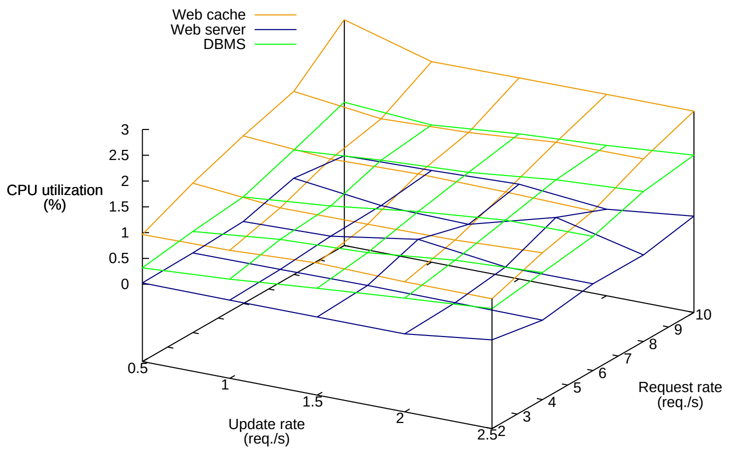


(a) Percentage considering all the processes of the system

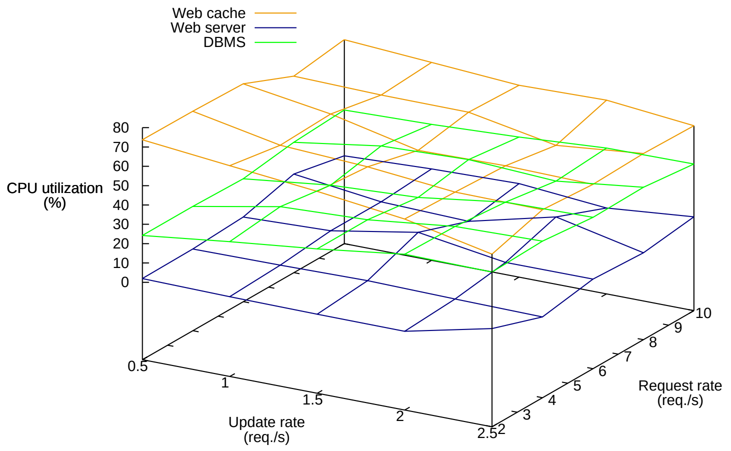


(b) Percentage considering only the processes in relation with the CAS system

Fig. 8.1. CPU utilization percentage for *entire*/$\emptyset$ in experiment set 4.



(a) Percentage considering all the processes of the system



(b) Percentage considering only the processes in relation with the CAS system

Fig. 8.2. CPU utilization percentage for the *split* scenario in experiment set 4.

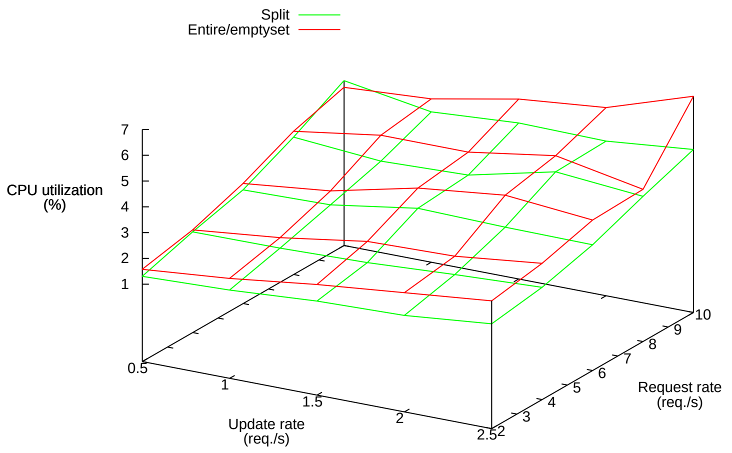


Fig. 8.3. Comparison of the overhead of *entire/∅* with the *split* scenario in experiment set 4.

In this analysis, the utilization percentage is also presented and analysed as global and particular percentages, as in the previous experiment. The values of the global percentages are shown in Tables 8.5 and 8.6, and the results of the particular values in Tables 8.7 and 8.8.

The analysis of the utilization is also going to be done in two different phases. Firstly, we are going to analyse the workload when the system load conditions changes and, secondly, comparing the overhead among different approaches.

8.3.1 Analysis of the workload generated by the adaptive core based on the use of decision trees

Firstly, we compare the results of this experiment with the previous workload analysis of our proposed implementation of the adaptive core. If we compare the usage plots for the global percentages of the experiment set 5 (Figure 8.4) with those of the previous experiment (Figure 8.1), we observe that almost all the processes have a similar behaviour. The trends of the data series corresponding to the web cache, adaptive core and DBMS are quite similar, although, in this last experiment, the evolution of the usage is less ordered.

The problem appears with the data series of the web server CPU utilization. In experiment set 5, the web server increases its usage considerably as

Table 8.5. Global utilization ratios (%) of the different application processes for *entire/∅* in experiment subset 5A.

Experiment	Web Cache		Adaptive core		Apache		DBMS		Total
	μ	σ	μ	σ	μ	σ	μ	σ	
`<ent./∅,2.0/0.5/s`$^{-1}$`>`	0.761	0.005	0.152	0.006	0.097	0.009	0.329	0.000	1.341
`<ent./∅,2.0/1.0/s`$^{-1}$`>`	0.909	0.002	0.225	0.008	0.164	0.049	0.543	0.000	2.184
`<ent./∅,2.0/1.5/s`$^{-1}$`>`	1.064	0.026	0.330	0.001	0.018	0.021	0.772	0.003	2.184
`<ent./∅,2.0/2.0/s`$^{-1}$`>`	1.220	0.080	0.412	0.006	0.091	0.021	0.932	0.000	2.657
`<ent./∅,2.0/2.5/s`$^{-1}$`>`	1.256	0.035	0.500	0.007	0.063	0.012	1.080	0.001	2.901
`<ent./∅,4.0/0.5/s`$^{-1}$`>`	0.837	0.014	0.204	0.006	0.070	0.015	0.404	0.002	1.517
`<ent./∅,4.0/1.0/s`$^{-1}$`>`	0.926	0.054	0.271	0.014	0.108	0.061	0.638	0.005	1.945
`<ent./∅,4.0/1.5/s`$^{-1}$`>`	1.039	0.000	0.002	0.000	0.001	0.063	0.878	0.001	1.922
`<ent./∅,4.0/2.0/s`$^{-1}$`>`	1.315	0.064	0.422	0.005	0.537	0.111	1.055	0.001	3.330
`<ent./∅,4.0/2.5/s`$^{-1}$`>`	1.481	0.122	0.494	0.019	0.340	0.345	1.250	0.000	3.566
`<ent./∅,6.0/0.5/s`$^{-1}$`>`	0.923	0.032	0.154	0.006	0.085	0.017	0.404	0.006	1.567
`<ent./∅,6.0/1.0/s`$^{-1}$`>`	0.834	0.016	0.291	0.015	0.432	0.347	0.648	0.009	2.206
`<ent./∅,6.0/1.5/s`$^{-1}$`>`	1.196	0.042	0.397	0.041	0.848	0.439	0.901	0.002	3.344
`<ent./∅,6.0/2.0/s`$^{-1}$`>`	1.282	0.045	0.506	0.049	0.545	0.208	1.132	0.003	3.466
`<ent./∅,6.0/2.5/s`$^{-1}$`>`	1.340	0.017	0.525	0.014	0.159	0.539	1.214	0.009	3.239
`<ent./∅,8.0/0.5/s`$^{-1}$`>`	0.926	0.085	0.266	0.018	0.461	0.112	0.518	0.004	2.172
`<ent./∅,8.0/1.0/s`$^{-1}$`>`	0.903	0.041	0.368	0.024	0.568	0.355	0.807	0.014	2.648
`<ent./∅,8.0/1.5/s`$^{-1}$`>`	0.987	0.030	0.392	0.043	1.129	0.565	0.905	0.005	3.415
`<ent./∅,8.0/2.0/s`$^{-1}$`>`	1.204	0.033	0.514	0.047	0.916	0.847	1.208	0.006	3.842
`<ent./∅,8.0/2.5/s`$^{-1}$`>`	1.314	0.046	0.552	0.007	0.260	0.181	1.295	0.009	3.423
`<ent./∅,10.0/0.5/s`$^{-1}$`>`	1.121	0.146	0.322	0.083	0.862	0.435	0.607	0.019	2.914
`<ent./∅,10.0/1.0/s`$^{-1}$`>`	1.067	0.203	0.293	0.019	0.911	0.552	0.708	0.009	2.980
`<ent./∅,10.0/1.5/s`$^{-1}$`>`	0.986	0.028	0.457	0.006	1.205	0.679	1.050	0.006	3.700
`<ent./∅,10.0/2.0/s`$^{-1}$`>`	1.434	0.191	0.532	0.037	1.506	0.683	1.141	0.013	4.615
`<ent./∅,10.0/2.5/s`$^{-1}$`>`	1.482	0.169	0.676	0.007	1.647	0.683	1.501	0.003	5.307

the request rate gets higher. This behaviour is not present in experiment set 4. The web server was the process with less CPU consumption in experiment 4. It also occurs in this experiment, but only in the cases with the lowest request rates. For the highest request rates, the web server involves to values of the web cache process (the process with the highest resource consumptions). Furthermore, in the previous experiment, the CPU utilization of the web server was very constant, but this does not occur in this case.

When we will analyse the CPU consumption of MACE, we will observe that these high increase of the web server process is also present. Thus, we can explain this due to the user behaviour model or to the web page model used in this experiment, because both features are the only differences with the experiment set 4.

In the case of the particular utilization percentage, calculating the utilization percentage over the total time of the four processes instead of over the

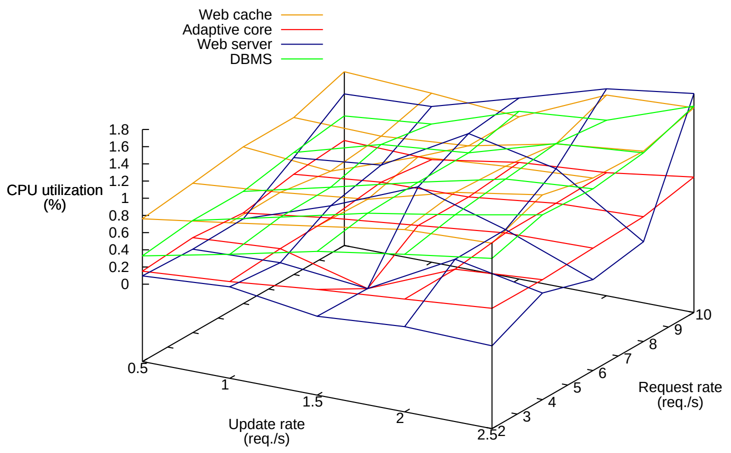
Table 8.6. Global utilization ratios (%) of the different application processes for MACE in experiment subset 5A.

Experiment	Web Cache		Adaptive core		Apache		DBMS		Total
	μ	σ	μ	σ	μ	σ	μ	σ	
`<mace,2.0/0.5/s`$^{-1}$`>`	0.869	0.110	0.132	0.008	0.037	0.040	0.183	0.001	1.222
`<mace,2.0/1.0/s`$^{-1}$`>`	0.875	0.088	0.175	0.005	0.025	0.016	0.279	0.000	1.355
`<mace,2.0/1.5/s`$^{-1}$`>`	0.984	0.045	0.267	0.006	0.007	0.055	0.411	0.001	1.671
`<mace,2.0/2.0/s`$^{-1}$`>`	0.989	0.040	0.328	0.008	0.020	0.042	0.503	0.001	1.841
`<mace,2.0/2.5/s`$^{-1}$`>`	0.981	0.073	0.390	0.016	0.012	0.036	0.599	0.002	1.983
`<mace,4.0/0.5/s`$^{-1}$`>`	0.904	0.018	0.171	0.013	0.050	0.050	0.253	0.001	1.379
`<mace,4.0/1.0/s`$^{-1}$`>`	0.999	0.078	0.212	0.010	0.021	0.091	0.356	0.001	1.589
`<mace,4.0/1.5/s`$^{-1}$`>`	0.976	0.132	0.269	0.010	0.025	0.062	0.450	0.001	1.721
`<mace,4.0/2.0/s`$^{-1}$`>`	1.239	0.135	0.337	0.023	0.156	0.063	0.548	0.001	2.282
`<mace,4.0/2.5/s`$^{-1}$`>`	1.189	0.103	0.394	0.012	0.233	0.145	0.648	0.003	2.465
`<mace,6.0/0.5/s`$^{-1}$`>`	1.148	0.144	0.172	0.004	0.447	0.368	0.268	0.003	2.036
`<mace,6.0/1.0/s`$^{-1}$`>`	1.136	0.130	0.229	0.028	0.235	0.061	0.401	0.001	2.002
`<mace,6.0/1.5/s`$^{-1}$`>`	1.181	0.121	0.329	0.026	0.288	0.092	0.537	0.003	2.335
`<mace,6.0/2.0/s`$^{-1}$`>`	1.307	0.121	0.412	0.003	0.153	0.127	0.659	0.004	2.532
`<mace,6.0/2.5/s`$^{-1}$`>`	1.228	0.227	0.400	0.016	0.101	0.284	0.692	0.002	2.423
`<mace,8.0/0.5/s`$^{-1}$`>`	1.095	0.146	0.197	0.013	0.616	0.329	0.353	0.009	2.263
`<mace,8.0/1.0/s`$^{-1}$`>`	1.335	0.154	0.309	0.028	0.552	0.180	0.514	0.000	2.711
`<mace,8.0/1.5/s`$^{-1}$`>`	1.280	0.151	0.302	0.019	0.208	0.189	0.552	0.006	2.344
`<mace,8.0/2.0/s`$^{-1}$`>`	1.233	0.076	0.443	0.015	0.253	0.328	0.722	0.007	2.653
`<mace,8.0/2.5/s`$^{-1}$`>`	1.411	0.257	0.431	0.020	0.378	0.541	0.750	0.006	2.973
`<mace,10.0/0.5/s`$^{-1}$`>`	0.983	0.138	0.281	0.055	0.993	0.695	0.426	0.011	2.685
`<mace,10.0/1.0/s`$^{-1}$`>`	1.218	0.134	0.240	0.034	0.332	0.140	0.461	0.012	2.253
`<mace,10.0/1.5/s`$^{-1}$`>`	1.204	0.289	0.358	0.023	0.267	0.035	0.659	0.011	2.490
`<mace,10.0/2.0/s`$^{-1}$`>`	1.223	0.084	0.399	0.013	0.486	0.137	0.698	0.012	2.808
`<mace,10.0/2.5/s`$^{-1}$`>`	1.502	0.151	0.537	0.044	0.516	0.197	0.900	0.006	3.457

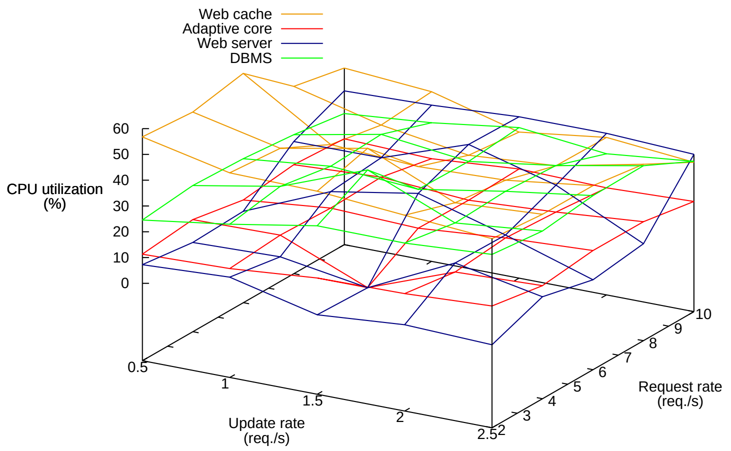
total time of the experiment, the trend for the web cache, the adaptive core and the DBMS is also the same than in experiment 4, but with less smooth values again. As it is expected because of the analysis of the global percentage, the web server increases its weight, in the total percentages of the four processes, as the request rates are increased.

8.3.2 Analysis of the workload generated by MACE

The order of the values of the CPU consumption of the four processes is the same in MACE than in the other approaches. The trend of the results of MACE (Figure 8.5) is more similar to the case of our adaptive core in experiment set 5 than to the cases of experiment set 4. The web server process also increases its CPU consumption as the request rate gets higher, the opposite to the experiment set 4, in which the web server CPU consumption remained



(a) Percentage considering all the processes of the system



(b) Percentage considering only the processes in relation with the CAS system

Fig. 8.4. CPU utilization percentage for *entire*/∅ in experiment set 5.

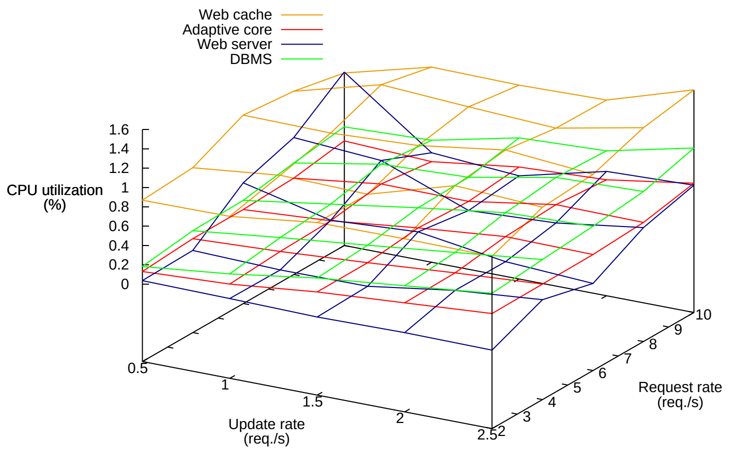
Table 8.7. Particular utilization ratios (%) of the different application processes for *entire/∅* in experiment subset 5A.

Experiment	Web Cache	Adaptive core	Apache	DBMS
`<entire/∅,2.0/0.5/s`$^{-1}$`>`	56.798	11.336	7.282	24.585
`<entire/∅,2.0/1.0/s`$^{-1}$`>`	49.351	12.250	8.940	29.458
`<entire/∅,2.0/1.5/s`$^{-1}$`>`	48.724	15.107	0.833	35.336
`<entire/∅,2.0/2.0/s`$^{-1}$`>`	45.925	15.521	3.451	35.103
`<entire/∅,2.0/2.5/s`$^{-1}$`>`	43.313	17.247	2.199	37.242
`<entire/∅,4.0/0.5/s`$^{-1}$`>`	55.185	13.498	4.643	26.674
`<entire/∅,4.0/1.0/s`$^{-1}$`>`	47.608	13.976	5.574	32.841
`<entire/∅,4.0/1.5/s`$^{-1}$`>`	54.089	0.113	0.084	45.714
`<entire/∅,4.0/2.0/s`$^{-1}$`>`	39.497	12.672	16.147	31.684
`<entire/∅,4.0/2.5/s`$^{-1}$`>`	41.535	13.873	9.543	35.049
`<entire/∅,6.0/0.5/s`$^{-1}$`>`	58.895	9.842	5.434	25.829
`<entire/∅,6.0/1.0/s`$^{-1}$`>`	37.812	13.195	19.590	29.404
`<entire/∅,6.0/1.5/s`$^{-1}$`>`	35.771	11.895	25.368	26.965
`<entire/∅,6.0/2.0/s`$^{-1}$`>`	36.980	14.608	15.742	32.670
`<entire/∅,6.0/2.5/s`$^{-1}$`>`	41.376	16.212	4.913	37.499
`<entire/∅,8.0/0.5/s`$^{-1}$`>`	42.634	12.263	21.232	23.871
`<entire/∅,8.0/1.0/s`$^{-1}$`>`	34.121	13.929	21.463	30.487
`<entire/∅,8.0/1.5/s`$^{-1}$`>`	28.912	11.498	33.086	26.504
`<entire/∅,8.0/2.0/s`$^{-1}$`>`	31.333	13.387	23.847	31.434
`<entire/∅,8.0/2.5/s`$^{-1}$`>`	38.408	16.140	7.621	37.831
`<entire/∅,10.0/0.5/s`$^{-1}$`>`	38.492	11.055	29.605	20.847
`<entire/∅,10.0/1.0/s`$^{-1}$`>`	35.803	9.836	30.590	23.770
`<entire/∅,10.0/1.5/s`$^{-1}$`>`	26.662	12.359	32.584	28.396
`<entire/∅,10.0/2.0/s`$^{-1}$`>`	31.081	11.528	32.649	24.741
`<entire/∅,10.0/2.5/s`$^{-1}$`>`	27.925	12.740	31.044	28.291

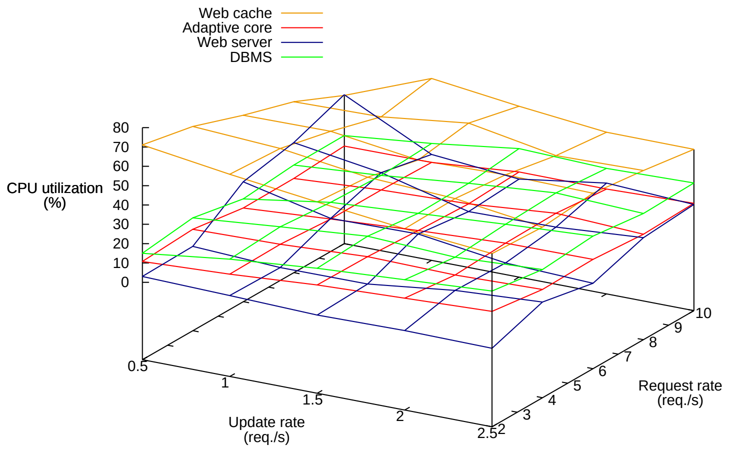
constant. The different behaviour of this process among the scenarios of experiment set 4 and 5 could be explained because of the user behaviour model or the web page model. In any case, the increase of the usage of the web server is smaller in MACE than in the case of using our adaptive core.

8.3.3 Comparison between the adaptive core based on decision trees and MACE

The main issue of the analysis of this experiment set is the comparison between the usages generated by each approach. This comparison has been done comparing the total consumption of the four processes and the consumption of the adaptive core. The extension of the CAS system with a module to adapt the fragment design using a classification algorithm does not only generate the overhead corresponding to the execution of the classification process (the adaptive core process), but also because the changes in the other tiers of the



(a) Percentage considering all the processes of the system



(b) Percentage considering only the processes in relation with the CAS system

Fig. 8.5. CPU utilization percentage for MACE in experiment set 5.

Table 8.8. Particular utilization ratios (%) of the different application processes for MACE in experiment subset 5A.

Experiment	Web Cache	Adaptive core	Apache	DBMS
`<mace,2.0/0.5/s`$^{-1}$`>`	71.079	10.800	3.081	15.041
`<mace,2.0/1.0/s`$^{-1}$`>`	64.536	12.940	1.877	20.647
`<mace,2.0/1.5/s`$^{-1}$`>`	58.894	16.011	0.477	24.619
`<mace,2.0/2.0/s`$^{-1}$`>`	53.729	17.818	1.086	27.367
`<mace,2.0/2.5/s`$^{-1}$`>`	49.502	19.677	0.621	30.200
`<mace,4.0/0.5/s`$^{-1}$`>`	65.584	12.428	3.639	18.350
`<mace,4.0/1.0/s`$^{-1}$`>`	62.883	13.361	1.347	22.409
`<mace,4.0/1.5/s`$^{-1}$`>`	56.699	15.654	1.471	26.176
`<mace,4.0/2.0/s`$^{-1}$`>`	54.302	14.808	6.846	24.044
`<mace,4.0/2.5/s`$^{-1}$`>`	48.230	16.005	9.484	26.282
`<mace,6.0/0.5/s`$^{-1}$`>`	56.387	8.456	21.995	13.162
`<mace,6.0/1.0/s`$^{-1}$`>`	56.759	11.445	11.756	20.041
`<mace,6.0/1.5/s`$^{-1}$`>`	50.563	14.105	12.333	23.000
`<mace,6.0/2.0/s`$^{-1}$`>`	51.619	16.292	6.057	26.032
`<mace,6.0/2.5/s`$^{-1}$`>`	50.716	16.529	4.180	28.575
`<mace,8.0/0.5/s`$^{-1}$`>`	48.403	8.736	27.220	15.632
`<mace,8.0/1.0/s`$^{-1}$`>`	49.258	11.428	20.350	18.958
`<mace,8.0/1.5/s`$^{-1}$`>`	54.622	12.919	8.910	23.548
`<mace,8.0/2.0/s`$^{-1}$`>`	46.501	16.702	9.556	27.241
`<mace,8.0/2.5/s`$^{-1}$`>`	47.475	14.526	12.745	25.253
`<mace,10.0/0.5/s`$^{-1}$`>`	36.638	10.474	37.001	15.887
`<mace,10.0/1.0/s`$^{-1}$`>`	54.064	10.660	14.774	20.502
`<mace,10.0/1.5/s`$^{-1}$`>`	48.381	14.393	10.742	26.484
`<mace,10.0/2.0/s`$^{-1}$`>`	43.583	14.218	17.311	24.888
`<mace,10.0/2.5/s`$^{-1}$`>`	43.449	15.544	14.947	26.059

CAS architecture, for example, the DBMS due to the updates of the fragment designs stored in its database. Thus, we have considered it is interesting to compare the adaptive core in an isolated way, and the total CPU utilization of the processes of the CAS system.

Figure 8.6 shows the total utilization of the four CAS processes for our approach and for MACE. We observe that the utilization values are practically equals for the experiments with update rate equal to 0.5, independently of the request rate. As the update rate is increased, the difference between the utilization of both solutions is exacerbated. On the contrary, the differences between the utilization values as the increases of request rate remain constant. Once again, we observe that the increase of the update rates penalizes the results of our solution.

If we focus in the utilization values of the four processes (Tables 8.5 and 8.6), we can see that the differences between our solution and MACE are mainly concentrated in the process of the web server and the DBMS. This

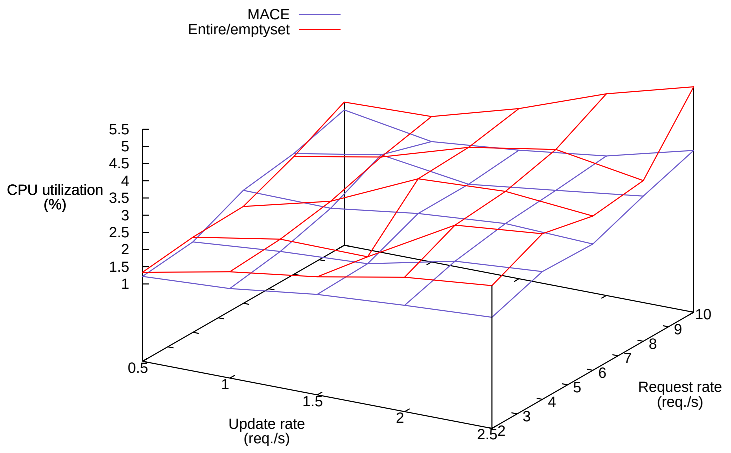
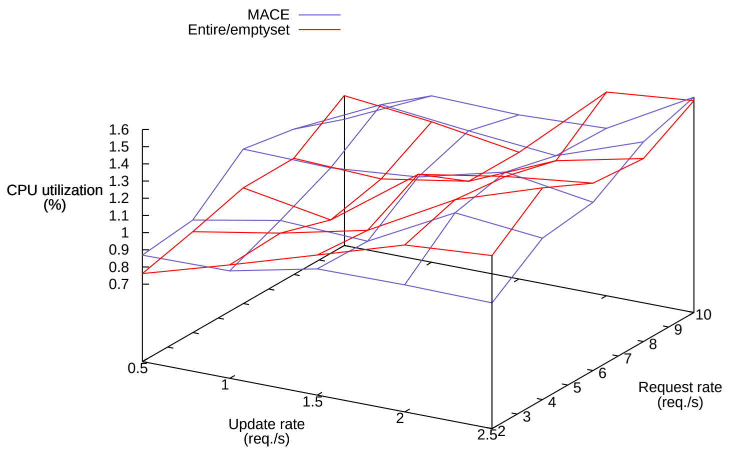


Fig. 8.6. Comparison of the overhead of *entire*/∅ scenario with MACE.

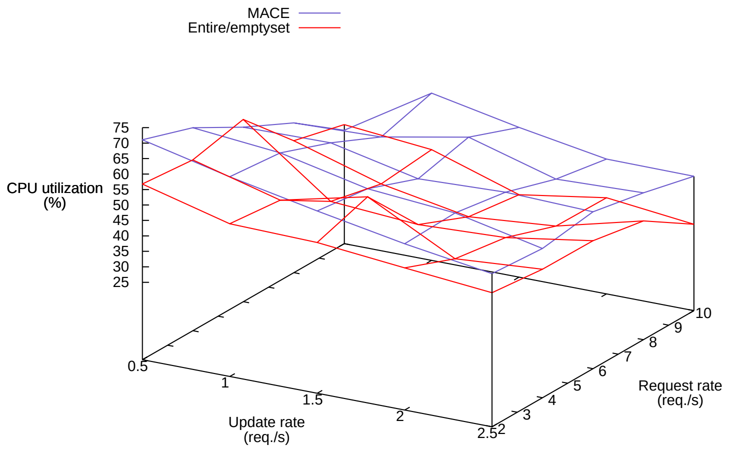
could be explained because the MACE algorithm only creates a cache point for each web page. On the contrary, our solution does not have a limited number of possible cacheable points and the classifications are done for each of the aggregation relationships. This is translated in a smaller number of parameter requests (DBMS read requests) and state updates (DBMS update requests) in the case of using MACE.

Finally, we are going to compare both solutions in terms of the utilization of the adaptive core. There is not a pattern in relation with the utilization values of both approaches, in the case of global percentage (Figure 8.7(a)). Some of the experiments show the lowest utilization in the case of our adaptive core approach and, in other ones, in the case of MACE. We cannot conclude which of the adaptive cores of both experiments generates less CPU consumption.

If we study the plot of the particular percentage values (Figure 8.7(b)), we observe that, in general terms, the core of MACE has a greater weight over the total workload of the CAS processes than in the case of our adaptive core. This can be caused because the adaptive core utilization is higher, or because the utilization of the other three processes is lower. As we have stated in the previous paragraph, there is not a clear difference between the core utilization of both solutions, so MACE generates smaller utilizations in the web cache, web server and DBMS.



(a) Percentage considering all the processes of the system



(b) Percentage considering only the processes in relation with the CAS system

Fig. 8.7. Adaptive core overhead for *entire/∅* scenario and MACE.

8.4 Summary

In this chapter, we have analysed the workload generated by three cache scenarios of CAS system: *split*, the use of the *entire/∅* decision tree and MACE. Several experiments for each scenario have been studied. The variations among the experiments are based on the user load level by the changes in the update and request rates.

After the analysis, we have concluded that the web cache is the software element that generates the most important part of the workload for the three scenarios. But in the case of our solution, the weight of the DBMS increases considerably as the update rate gets higher. We consider that one important improvement for our solution is to study the way of reducing the utilization generated by the DBMS.

The adaptive core has been deployed in *event-driven mode* in all the experiments (Section 5.2). It means a classification process and an update of the fragment designs are done every time that a characterization parameter of a content element changes. Probably, a *periodical mode* would reduce the workload of our solution. It would also reduce the improvement in terms of user-perceived latencies due to the time between a content change and the fragment design adaptation. Interesting future works emerge from these two concepts: the reduction of the number of accesses to the database and the behaviour of a *periodical mode* deployment of the CAS system.

In any case, the overhead of the CPU utilization, experimented in the case of using decision trees, is bigger than the workload generated by the traditional *split* scenario, but this difference is quite small. In comparison with MACE, our solution also generates more CPU consumption in total terms. On the contrary, the workload generated by only the adaptive core of our solution is smaller than in the case of MACE. This is because our approach generates a larger increase in the rest of processes (mainly in the web server and DBMS).

Part IV

Conclusions

9 Conclusions and open problems

¿Qué es la vida? Un frenesí.
¿Qué es la vida? Una ilusión,
una sombra, una ficción,
y el mayor bien es pequeño;
que toda la vida es sueño,
y los sueños, sueños son.
—Calderón de la Barca—

This final chapter summarizes the statements, concepts, contributions, experiments and results of this thesis. It lists main contributions we have identified and opens a discussion about future work and further research lines.

9.1 Thesis summary

This thesis demonstrated that the user-perceived latency of a content aggregation systems can be reduced by adapting the fragment elements that are managed and stored independently in the web cache. Chapter 2 included the introduction to the general architecture for content aggregation web applications and the explanation to the performance limitations that these new applications generate in the web caches. The chapter also included a survey of the fragment-based web caching techniques and another one of the use of data mining in web performance engineering. In Chapter 3, we proposed a formal definition of a model for content aggregated web pages. This model is based on a DAG and it represents the content fragments by indicating the assembly point (web server or web cache) of each pair of aggregated content elements. We also proposed to reduce the user-perceived latency by adapting the content fragments as the characteristics of the contents change. We studied the relationship among a set of content element characterization parameters and the assembly point in which the shortest latency values are observed. These characterization parameters were selected as inputs of the algorithm which classifies the assembly points (states) of each aggregation relationship. Chapter 4 included the details of the core of the proposed solution. We explained the inputs, outputs and process to be done by the adaptive core of the system. The adjective of adaptive is used because the core is in charge of deciding the changes of the content fragments, *i.e.*, the adaptation of the fragment designs to improve the latency of the system. Several implementations of the core were considered, but finally rejected, as the use of ontologies or genetic algorithms. The second part of the chapter was devoted to explain the details of the solution selected to implement the core of the system, data mining and decision trees. The details of the phases of a knowledge discovery process based on data mining, and the details for our particular case, were explained in the chapter. The first part of Chapter 5 included the general definition of a framework to improve the user-perceived latency in content aggregation system using an adaptive core, COFRADIAS framework. The implementation of the adaptive core was addressed as a classification algorithm implemented with decision trees. The knowledge represented by the decision trees was mined in an off-line training phase using a data set obtained from the emulation of synthetic web content page models. The second part of the chapter was devoted to the implementation details of an example of the general framework. We adapted a commercial and wide-used web application, Drupal, to include the management of content aggregations and to include the adaptive core of COFRADIAS. Chapter 6 detailed the design of the experiments to evaluate and to validate our approach. The experiments were designed by defining a set of features. They were executed in different environment conditions in order to evaluate the influences of these features on the benefits obtained by our solution. Five different experiment sets, with a total number of 10 experiment subsets were defined. The details of the test-bed, in which the experiments were executed, were also explained in that chapter. Chapters 7 and 8 were devoted to analyse the results of the experiments. Finally, 158 experiment executions, with their correspondent replicas, were done. Our approach was evaluated in terms of user-perceived latency reduction, web cache hit ratios and produced overhead. Our approach was also compared with other traditional solutions to implement web caches in content aggregation systems and with approaches of other researches (MACE framework). Our solution showed smaller latencies than the other approaches in almost all the experiment configurations. On the contrary, the cache hit ratio was worse than in the other solutions. But the benefits of reducing the assembling times counteracted the losses in the hit ratios. Finally, we concluded that the overhead increase of our solution, in front of other approaches, was small enough to consider our framework as a suitable solution for the problem of web caching in content aggregation systems and to reduce the user-perceived latencies.

9.2 Contributions

We have identified the next contributions:

- *Definition of a methodology to deal with the problem of web caching in content aggregation systems.* In order to solve the initial problem of our research, we have proposed and followed a methodology divided into phases. This methodology could be used again to address similar problems in content aggregation systems or similar systems.

- *Proposal of the adaptation of the content fragments managed by the web cache to solve the cache performance degradation problem in content aggregation systems.* This dissertation proposes a framework to counteract the limitations of cache performance in environments with high update rates and web pages with a high level of user customization. The framework bases the contributions in adapting dynamically the fragment designs using knowledge extracted from synthetic data. These fragment designs define the assembly point of the content elements.
- *Definition of a formal model for content aggregation structure, content element characterization parameters, and content fragment designs.* Content aggregation pages can be modelled by the use of Directed Acyclic Graphs (DAG). We have based our model definition on an existing one, ODG (Object Dependency Graph), but we have extended it (ODGex, extended Object Dependency Graph) to allow the representation of content element characterization parameters —by the use of vertex attributes corresponding to the inputs of our adaptive core— and of content fragment designs —by the use of edge labels corresponding to the outputs of the adaptive core, *i.e.*, the assembly point or state of an edge—.
- *Validation that characterization parameters of content elements can be used to predict the assembly point which generates a shorter user-perceived latency.* We have done several experiments to validate that the user-perceived latencies are affected by changes in the content fragment. These experiments have been also used to demonstrate that the values of the characterization parameters of the content elements are important to determine the assembly point that generates a higher performance.
- *Definition of a set of parameters to be used to define the content fragment designs.* Once the previous contribution was validated, we researched the metrics that have influence on the performance of a fragment design. We have validated that a set of characterization parameters of two aggregated content elements can predict the assembly point for which the latency of the web page is shorter. These attributes are the content size, the content update rate, the content request rate, the number of aggregators (fathers) of the aggregated content element (child) and the number of aggregations (children) of the aggregator content element (father). These characterization parameters have been used as the independent attributes, or inputs, of the classification algorithm of the adaptive core of COFRADIAS.
- *Design of a core in charge of adapting the fragment designs.* We have stated that the adaptation of the fragment designs can solve the problem of caching in content aggregation systems. We have called adaptive core to the element in charge of defining these fragment designs. We have defined the requirements and features of this adaptive core.
- *Study of several techniques for the deployment of the adaptive core.* Ontologies and genetic algorithms have been evaluated as core of our framework. The benefits and drawbacks have been stated, and we have finally

rejected both of them. Some conclusions and results have been obtained from this study.

- *Definition of the process to adapt fragment designs of a content aggregation system with knowledge discovery and data mining techniques.* We have used data mining to deploy the core of the solution. This involves several phases and contributions. Firstly, we have defined a procedure to create content models to be emulated in order to extract data about the assembly points that generate higher performance in relation with the characterization parameters of the content elements in CAS systems. Secondly, we have designed the data instances representations which best include the data obtained from the previous emulation phase. This design includes the definition of the independent attributes and their numerical representation in order to create the training data sets. And, finally, we have studied the algorithms and knowledge representations that best suit in our process. We selected C4.5 and decision trees as the best algorithm and knowledge representation.
- *Design of a general framework to integrate the adaptation of content fragments in web content aggregation systems.* We have called COFRADIAS to this general framework. This framework defines the changes in the tiers and in the modules of the content aggregation systems and the new interfaces among the modules of the CAS system and the adaptive core. The design of the framework is independent of the deployment of the adaptive core. This framework can be used independently of the techniques or solution used to adapt the fragment designs.
- *Development of a COFRADIAS framework example.* We have extended Drupal, a commercial content management system. Thus, it manages content aggregation and it includes COFRADIAS framework. This extension has been done by the implementation of a Drupal module. Two different implementations of the adaptive core have been used into the COFRADIAS implementation: one with our adaptive core solution based on decision trees, and another one with the MACE algorithm based on cost functions.
- *Performance study of COFRADIAS framework.* Several experiments have been presented to evaluate the benefits of our solution. We have compared the results of our approach with the two traditional cache schemes of CAS systems (assembling all the content elements in the web cache or in the web application) and with a similar approach proposed by Hassan *et al.*, MACE framework. The evaluation has been done in terms of user-perceived latency speed-ups and in terms of system CPU overhead. Our solution has shown great latency improvements and it obtained these improvements generating low overheads in the CPU consumption of the servers.

9.3 Future work and open problems

Despite the contributions of this dissertation, there are several open problems that are planned as future work:

- *To investigate the relationship between the characteristics of the decision trees (size and coverage) and the improvement of the performance they achieve when they are used in the adaptive core.* We have not observed any relationship between the size and the coverage of a decision tree, and the improvement it offers to the system. Thus, we can only choose among different trees by executing them in the COFRADIAS framework. This analysis could be based on the size or coverage of the decision trees, or even other tree characteristics.
- *To investigate how to reduce the overhead of the system.* One of the most overloaded parts of the system is the DBMS. We have studied these overheads in an *event-driven mode*, fragments are adapted in each characterization change. It would be interesting to study the overhead and the benefits of the solution in a *periodical mode*, fragment adaptation is done in intervals of times. Also other improvements for the process of updating the fragment designs in the DBMS could be studied.
- *To study in which points in time a training phase should be done.* We have considered that it is enough to train the system —to extract the knowledge and to create the decision trees— when changes in the architecture are done. This is because our training models cover a wide range of content characteristics changes and user behaviour cases. Further research could be done.

The expertise obtained through this research allows us to propose new research challenges, even the use of the results and findings of this thesis in some applications. These applications and challenges include, but they are not limited to:

- *Integration of all the phases of the data mining process in a software suite.* This tool should automatically execute all the phases involves since the selection of content aggregation system and the best decision tree is selected. All the tasks to be executed involve: (a) the crawling of the content of the goal web site; (b) the use of the parameters extracted from the web site to create the synthetic content page model; (c) the emulation of the system using the content page model and the monitoring of the performance metrics; (d) the creation of the data instances of the data training set; (e) the extraction of the knowledge and the creation of the decision trees; (f) the selection of the best decision tree, which generate shortest latencies, by evaluating them in a set of experiments. In fact, some of these tasks have been done. For example, the application of automatic crawling of content web sites is finished, and it have been used in some tasks of our research work.

- *Evaluation of the use of other techniques in the deployment of the adaptive core.* We have considered other techniques to deploy the adaptive core, for example the multi-criteria optimization, but we have not investigated enough in this direction in order to make conclusions. Further research could be done in the use of this or other techniques.
- *Use of the methodology followed in this dissertation to study the inclusion of new input parameters in the adaptive core.* Even, to study the possibility of using this method to achieve the improvement of different metrics to the user-perceived latency.

Part V

Appendixes and references

A

Decision trees specification

This appendix shows the structure of the decision trees for all the training data sets created by the monitoring of the performance for the emulation of the synthetic web content models. For each tree, it is presented the independent attributes and the structure of the decision tree —leave vertices assign the state to the aggregation relationship and the other vertices are evaluations of the values of the independent attributes—.

A.1 Distributed hardware architecture

A.1.1 Entire/∅

```
Attributes:  9
          fupdaterate
          cupdaterate
          frequestrate
          crequestrate
          fatherChildNumber
          childFatherNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------

frequestrate <= 63.7904
|  fatherChildNumber <= 5: J
|  fatherChildNumber > 5
|  |  frequestrate <= 9.69782
|  |  |  crequestrate <= 1.2795: J
|   |   |   crequestrate > 1.2795
|   |   |   |   fatherSize <= 259: S
|   |   |   |   fatherSize > 259
|   |   |   |   |   crequestrate <= 1.33426: S
|   |   |   |   |   crequestrate > 1.33426
|   |   |   |   |   |   childFatherNumber <= 4
|   |   |   |   |   |   |   childSize <= 1550: J
|   |   |   |   |   |   |   childSize > 1550
|   |   |   |   |   |   |   |   childFatherNumber <= 2: S
|   |   |   |   |   |   |   |   childFatherNumber > 2: J
|   |   |   |   |   |   childFatherNumber > 4
|   |   |   |   |   |   |   childFatherNumber <= 8
|   |   |   |   |   |   |   |   childFatherNumber <= 7
|   |   |   |   |   |   |   |   |   childFatherNumber <= 6
|   |   |   |   |   |   |   |   |   |   frequestrate <= 7.88797
|   |   |   |   |   |   |   |   |   |   |   cupdaterate <= 573
|   |   |   |   |   |   |   |   |   |   |   |   fatherChildNumber <= 7
|   |   |   |   |   |   |   |   |   |   |   |   |   childSize <= 992: J
|   |   |   |   |   |   |   |   |   |   |   |   |   childSize > 992: S
|   |   |   |   |   |   |   |   |   |   |   |   fatherChildNumber > 7: S
|   |   |   |   |   |   |   |   |   |   |   cupdaterate > 573: J
|   |   |   |   |   |   |   |   |   |   frequestrate > 7.88797: J
|   |   |   |   |   |   |   |   |   childFatherNumber > 6: S
|   |   |   |   |   |   |   |   childFatherNumber > 7: J
|   |   |   |   |   |   |   childFatherNumber > 8: S
|   |   frequestrate > 9.69782
|   |   |   childFatherNumber <= 4
|   |   |   |   childFatherNumber <= 3
|   |   |   |   |   fatherChildNumber <= 8
|   |   |   |   |   |   childSize <= 576
|   |   |   |   |   |   |   fatherChildNumber <= 6
|   |   |   |   |   |   |   |   childSize <= 466: S
|   |   |   |   |   |   |   |   childSize > 466: J
|   |   |   |   |   |   |   fatherChildNumber > 6
|   |   |   |   |   |   |   |   childSize <= 267: J
|   |   |   |   |   |   |   |   childSize > 267: S
|   |   |   |   |   |   childSize > 576
|   |   |   |   |   |   |   childSize <= 1504: J
|   |   |   |   |   |   |   childSize > 1504
|   |   |   |   |   |   |   |   fupdaterate <= 840
|   |   |   |   |   |   |   |   |   childFatherNumber <= 2
|   |   |   |   |   |   |   |   |   |   childSize <= 1584: J
|   |   |   |   |   |   |   |   |   |   childSize > 1584
|   |   |   |   |   |   |   |   |   |   |   frequestrate <= 36.0784: S
|   |   |   |   |   |   |   |   |   |   |   frequestrate > 36.0784
|  |  |  |  |  |  |  |  |  |  |  |  fupdaterate <= 275: S
|  |  |  |  |  |  |  |  |  |  |  |  fupdaterate > 275: J
|  |  |  |  |  |  |  |  |  childFatherNumber > 2
|  |  |  |  |  |  |  |  |  |  fupdaterate <= 695
|  |  |  |  |  |  |  |  |  |  |  fupdaterate <= 93: S
|  |  |  |  |  |  |  |  |  |  |  fupdaterate > 93
|  |  |  |  |  |  |  |  |  |  |  |  crequestrate <= 16.17: J
|  |  |  |  |  |  |  |  |  |  |  |  crequestrate > 16.17
|  |  |  |  |  |  |  |  |  |  |  |  |  fatherSize <= 595: J
|  |  |  |  |  |  |  |  |  |  |  |  |  fatherSize > 595: S
|  |  |  |  |  |  |  |  |  |  fupdaterate > 695: S
|  |  |  |  |  |  |  |  fupdaterate > 840: J
|  |  |  |  |  fatherChildNumber > 8
|  |  |  |  |  |  childFatherNumber <= 2
|  |  |  |  |  |  |  frequestrate <= 26.5896: J
|  |  |  |  |  |  |  frequestrate > 26.5896
|  |  |  |  |  |  |  |  crequestrate <= 26.4706: S
|  |  |  |  |  |  |  |  crequestrate > 26.4706: J
|  |  |  |  |  |  childFatherNumber > 2
|  |  |  |  |  |  |  crequestrate <= 12.1265
|  |  |  |  |  |  |  |  crequestrate <= 8.18101
|  |  |  |  |  |  |  |  |  cupdaterate <= 269: S
|  |  |  |  |  |  |  |  |  cupdaterate > 269: J
|  |  |  |  |  |  |  |  crequestrate > 8.18101: S
|  |  |  |  |  |  |  crequestrate > 12.1265: J
|  |  |  |  childFatherNumber > 3: J
|  |  |  childFatherNumber > 4: J
frequestrate > 63.7904
|  fatherChildNumber <= 9: J
|  fatherChildNumber > 9
|  |  fupdaterate <= 219
|  |  |  fupdaterate <= 157
|  |  |  |  childFatherNumber <= 5
|  |  |  |  |  cupdaterate <= 80: S
|  |  |  |  |  cupdaterate > 80: J
|  |  |  |  childFatherNumber > 5: S
|  |  |  fupdaterate > 157: J
|  |  fupdaterate > 219
|  |  |  childSize <= 586: J
|  |  |  childSize > 586: S

Number of Leaves  :  46
Size of the tree :  91
```

A.1.2 Entire/updateRate

```
Attributes:  7
          frequestrate
          crequestrate
          fatherChildNumber
          childFatherNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|  crequestrate <= 3.29264: S
|  crequestrate > 3.29264: J

Number of Leaves  :  3
Size of the tree :  5
```

A.1.3 Entire/requestRate

```
Attributes:  7
          fupdaterate
          cupdaterate
          fatherChildNumber
          childFatherNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 10: J
fatherChildNumber > 10
|  fupdaterate <= 455
|  |  fatherChildNumber <= 11
|  |  |  childFatherNumber <= 5
|  |  |  |  fupdaterate <= 206
|  |  |  |  |  cupdaterate <= 80: S
|  |  |  |  |  cupdaterate > 80: J
|  |  |  |  fupdaterate > 206
|  |  |  |  |  childSize <= 586: J
|  |  |  |  |  childSize > 586: S
|  |  |  childFatherNumber > 5: S
|  |  fatherChildNumber > 11
|  |  |  childFatherNumber <= 7
|  |  |  |  childSize <= 1618: S
|  |  |  |  childSize > 1618: J
|  |  |  childFatherNumber > 7: J
|  fupdaterate > 455: J

Number of Leaves  :  10
Size of the tree :  19
```

A.1.4 Entire/size

```
Attributes:  8
          fupdaterate
          cupdaterate
          frequestrate
          crequestrate
          fatherChildNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.5 Entire/childrenNumber

```
Attributes:  7
          fupdaterate
          cupdaterate
          frequestrate
          crequestrate
          fatherChildNumber
          childFatherNumber
          improvingclass

J48 pruned tree
------------------
```

```
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.6 Entire/fatherNumber

```
Attributes:  8
          fupdaterate
          cupdaterate
          frequestrate
          crequestrate
          childFatherNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.7 Ratio/∅

```
Attributes:  6
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 4
|  fatherChildNumber <= 2: J
|  fatherChildNumber > 2
|  |  requestrate <= 7.570455: J
|  |  requestrate > 7.570455
|  |  |  updaterate <= 10.8293
|  |  |  |  requestrate <= 7.906367: S
|  |  |  |  requestrate > 7.906367
|   |   |   |   |   childFatherNumber <= 7: J
|   |   |   |   |   childFatherNumber > 7
|   |   |   |   |   |   updaterate <= 1.4633
|   |   |   |   |   |   |   childFatherNumber <= 9
|   |   |   |   |   |   |   |   updaterate <= 0.7794: J
|   |   |   |   |   |   |   |   updaterate > 0.7794: S
|   |   |   |   |   |   |   childFatherNumber > 9
|   |   |   |   |   |   |   |   Size <= 0.3288: S
|   |   |   |   |   |   |   |   Size > 0.3288: J
|   |   |   |   |   |   updaterate > 1.4633
|   |   |   |   |   |   |   fatherChildNumber <= 3
|   |   |   |   |   |   |   |   requestrate <= 42.803118
|   |   |   |   |   |   |   |   |   childFatherNumber <= 8
|   |   |   |   |   |   |   |   |   |   updaterate <= 1.7745: J
|   |   |   |   |   |   |   |   |   |   updaterate > 1.7745
|   |   |   |   |   |   |   |   |   |   |   requestrate <= 14.555: J
|   |   |   |   |   |   |   |   |   |   |   requestrate > 14.555: S
|   |   |   |   |   |   |   |   |   childFatherNumber > 8: J
|   |   |   |   |   |   |   |   requestrate > 42.803118: J
|   |   |   |   |   |   |   fatherChildNumber > 3
|   |   |   |   |   |   |   |   childFatherNumber <= 8: J
|   |   |   |   |   |   |   |   childFatherNumber > 8
|   |   |   |   |   |   |   |   |   requestrate <= 28.551851
|   |   |   |   |   |   |   |   |   |   Size <= 0.7799
|   |   |   |   |   |   |   |   |   |   |   childFatherNumber <= 9: S
|   |   |   |   |   |   |   |   |   |   |   childFatherNumber > 9: J
|   |   |   |   |   |   |   |   |   |   Size > 0.7799: J
|   |   |   |   |   |   |   |   |   requestrate > 28.551851
|   |   |   |   |   |   |   |   |   |   requestrate <= 179.202936: S
|   |   |   |   |   |   |   |   |   |   requestrate > 179.202936: J
|   |   |   updaterate > 10.8293
|   |   |   |   fatherChildNumber <= 3: S
|   |   |   |   fatherChildNumber > 3
|   |   |   |   |   Size <= 0.74
|   |   |   |   |   |   childFatherNumber <= 4: J
|   |   |   |   |   |   childFatherNumber > 4
|   |   |   |   |   |   |   updaterate <= 11.3636: S
|   |   |   |   |   |   |   updaterate > 11.3636
|   |   |   |   |   |   |   |   Size <= 0.5132: S
|   |   |   |   |   |   |   |   Size > 0.5132: J
|   |   |   |   |   Size > 0.74: S
fatherChildNumber > 4
|   fatherChildNumber <= 10: J
|   fatherChildNumber > 10
|   |   fatherChildNumber <= 11
|  |  |  childFatherNumber <= 5
|  |  |  |  requestrate <= 58.837999: J
|  |  |  |  requestrate > 58.837999: S
|  |  |  childFatherNumber > 5: S
|  |  fatherChildNumber > 11
|  |  |  childFatherNumber <= 7
|  |  |  |  Size <= 0.8286: J
|  |  |  |  Size > 0.8286: S
|  |  |  childFatherNumber > 7: J

Number of Leaves  :  32
Size of the tree :  63
```

A.1.8 Ratio/updateRate

```
Attributes:  5
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 10: J
fatherChildNumber > 10
|  fatherChildNumber <= 11
|  |  childFatherNumber <= 5
|  |  |  requestrate <= 58.837999: J
|  |  |  requestrate > 58.837999: S
|  |  childFatherNumber > 5: S
|  fatherChildNumber > 11
|  |  childFatherNumber <= 7
|  |  |  Size <= 0.8286: J
|  |  |  Size > 0.8286: S
|  |  childFatherNumber > 7: J

Number of Leaves  :  7
Size of the tree :  13
```

A.1.9 Ratio/requestRate

```
Attributes:  5
          updaterate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 10: J
fatherChildNumber > 10
|   fatherChildNumber <= 11: S
|   fatherChildNumber > 11
|   |   childFatherNumber <= 7
|   |   |   Size <= 0.8286: J
|   |   |   Size > 0.8286: S
|   |   childFatherNumber > 7: J

Number of Leaves  :  5
Size of the tree :  9
```

A.1.10 Ratio/size

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 4
|   fatherChildNumber <= 2: J
|   fatherChildNumber > 2
|   |   requestrate <= 7.570455: J
|   |   requestrate > 7.570455
|   |   |   updaterate <= 10.8293
|   |   |   |   requestrate <= 7.906367: S
|   |   |   |   requestrate > 7.906367
|   |   |   |   |   childFatherNumber <= 7: J
|   |   |   |   |   childFatherNumber > 7
|   |   |   |   |   |   updaterate <= 1.4633
|   |   |   |   |   |   |   childFatherNumber <= 9
|   |   |   |   |   |   |   |   updaterate <= 0.7794: J
|  |  |  |  |  |  |  |  updaterate > 0.7794: S
|  |  |  |  |  |  |  childFatherNumber > 9: J
|  |  |  |  |  |  updaterate > 1.4633
|  |  |  |  |  |  |  fatherChildNumber <= 3
|  |  |  |  |  |  |  |  requestrate <= 42.803118
|  |  |  |  |  |  |  |  |  childFatherNumber <= 8
|  |  |  |  |  |  |  |  |  |  updaterate <= 1.7745: J
|  |  |  |  |  |  |  |  |  |  updaterate > 1.7745
|  |  |  |  |  |  |  |  |  |  |  requestrate <= 14.555: J
|  |  |  |  |  |  |  |  |  |  |  requestrate > 14.555: S
|  |  |  |  |  |  |  |  |  childFatherNumber > 8: J
|  |  |  |  |  |  |  |  requestrate > 42.803118: J
|  |  |  |  |  |  |  fatherChildNumber > 3
|  |  |  |  |  |  |  |  childFatherNumber <= 8: J
|  |  |  |  |  |  |  |  childFatherNumber > 8
|  |  |  |  |  |  |  |  |  requestrate <= 28.551851
|  |  |  |  |  |  |  |  |  |  childFatherNumber <= 9
|  |  |  |  |  |  |  |  |  |  |  requestrate <= 13.457064: S
|  |  |  |  |  |  |  |  |  |  |  requestrate > 13.457064: J
|  |  |  |  |  |  |  |  |  |  childFatherNumber > 9: J
|  |  |  |  |  |  |  |  |  requestrate > 28.551851
|  |  |  |  |  |  |  |  |  |  requestrate <= 179.202936: S
|  |  |  |  |  |  |  |  |  |  requestrate > 179.202936: J
|  |  |  updaterate > 10.8293: S
fatherChildNumber > 4
|  fatherChildNumber <= 10: J
|  fatherChildNumber > 10
|  |  fatherChildNumber <= 11
|  |  |  childFatherNumber <= 5
|  |  |  |  requestrate <= 58.837999: J
|  |  |  |  requestrate > 58.837999: S
|  |  |  childFatherNumber > 5: S
|  |  fatherChildNumber > 11
|  |  |  childFatherNumber <= 7: S
|  |  |  childFatherNumber > 7: J

Number of Leaves  :  25
Size of the tree :  49
```

A.1.11 Ratio/childrenNumber

```
Attributes:  5
          updaterate
          requestrate
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.12 Ratio/fatherNumber

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 4
|  fatherChildNumber <= 2: J
|  fatherChildNumber > 2
|  |  requestrate <= 7.570455: J
|  |  requestrate > 7.570455
|  |  |  updaterate <= 10.8293
|  |  |  |  requestrate <= 7.906367
|  |  |  |  |  requestrate <= 7.693931
|  |  |  |  |  |  requestrate <= 7.64: S
|  |  |  |  |  |  requestrate > 7.64: J
|  |  |  |  |  requestrate > 7.693931: S
|  |  |  |  requestrate > 7.906367: J
|  |  |  updaterate > 10.8293
|  |  |  |  Size <= 0.74
|  |  |  |  |  updaterate <= 11.3636: S
|  |  |  |  |  updaterate > 11.3636: J
|  |  |  |  Size > 0.74: S
fatherChildNumber > 4
|  fatherChildNumber <= 10: J
|  fatherChildNumber > 10
|  |  requestrate <= 20.727041: J
|  |  requestrate > 20.727041
|  |  |  Size <= 1.5501: S
|   |   |  Size > 1.5501
|   |   |   |  Size <= 3.083: J
|   |   |   |  Size > 3.083: S

Number of Leaves  :  14
Size of the tree :  27
```

A.1.13 Difference/∅

```
Attributes:  6
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|  fatherChildNumber <= 10: J
|  fatherChildNumber > 10
|   |  requestrate <= 165.55097: J
|   |  requestrate > 165.55097: S

Number of Leaves  :  4
Size of the tree :  7
```

A.1.14 Difference/updateRate

```
Attributes:  5
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|  fatherChildNumber <= 10: J
|  fatherChildNumber > 10
|  |  requestrate <= 165.55097: J
|  |  requestrate > 165.55097: S

Number of Leaves  :  4
Size of the tree :  7
```

A.1.15 Difference/requestRate

```
Attributes:  5
          updaterate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 10: J
fatherChildNumber > 10
|  fatherChildNumber <= 11: S
|  fatherChildNumber > 11
|  |  childFatherNumber <= 7
|  |  |  Size <= -277: J
|  |  |  Size > -277: S
|  |  childFatherNumber > 7: J

Number of Leaves  :  5
Size of the tree :  9
```

A.1.16 Difference/size

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
```

```
|  fatherChildNumber <= 10: J
|  fatherChildNumber > 10
|  |  requestrate <= 165.55097: J
|  |  requestrate > 165.55097: S

Number of Leaves  :  4
Size of the tree :  7
```

A.1.17 Difference/childrenNumber

```
Attributes:  5
          updaterate
          requestrate
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.18 Difference/fatherNumber

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|  fatherChildNumber <= 10: J
|  fatherChildNumber > 10
|  |  requestrate <= 165.55097: J
|  |  requestrate > 165.55097: S

Number of Leaves  :  4
Size of the tree :  7
```

A.1.19 Distance/∅

```
Attributes:  6
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.20 Distance/updateRate

```
Attributes:  5
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.21 Distance/requestRate

```
Attributes:  5
          updaterate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
```

```
fatherChildNumber <= 10: J
fatherChildNumber > 10: S

Number of Leaves  :  2
Size of the tree :  3
```

A.1.22 Distance/size

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.23 Distance/childrenNumber

```
Attributes:  5
          updaterate
          requestrate
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.1.24 Distance/fatherNumber

```
Attributes:  5
          updaterate
          requestrate
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.2 Centralized hardware architecture

A.2.1 Entire/∅

```
Attributes:  9
          fupdaterate
          cupdaterate
          frequestrate
          crequestrate
          fatherChildNumber
          childFatherNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------

frequestrate <= 63.7413: J
frequestrate > 63.7413
|   fatherChildNumber <= 2
|   |   fatherChildNumber <= 1: J
|   |   fatherChildNumber > 1
|   |   |   childFatherNumber <= 6
|   |   |   |   childFatherNumber <= 4
|   |   |   |   |   cupdaterate <= 463
|   |   |   |   |   |   crequestrate <= 33.8097: S
|   |   |   |   |   |   crequestrate > 33.8097: J
|   |   |   |   |   cupdaterate > 463
|   |   |   |   |   |   frequestrate <= 138.462: J
|   |   |   |   |   |   frequestrate > 138.462
|   |   |   |   |   |   |   childFatherNumber <= 2: S
|   |   |   |   |   |   |   childFatherNumber > 2
|   |   |   |   |   |   |   |   crequestrate <= 1.83841: S
|  |  |  |  |  |  |  |  crequestrate > 1.83841: J
|  |  |  |  childFatherNumber > 4
|  |  |  |  |  cupdaterate <= 523: J
|  |  |  |  |  cupdaterate > 523
|  |  |  |  |  |  fupdaterate <= 897
|  |  |  |  |  |  |  crequestrate <= 16.9049: J
|  |  |  |  |  |  |  crequestrate > 16.9049
|  |  |  |  |  |  |  |  crequestrate <= 19.3639: S
|  |  |  |  |  |  |  |  crequestrate > 19.3639: J
|  |  |  |  |  |  fupdaterate > 897: S
|  |  |  childFatherNumber > 6
|  |  |  |  fupdaterate <= 542
|  |  |  |  |  childFatherNumber <= 8
|  |  |  |  |  |  crequestrate <= 8.41121: S
|  |  |  |  |  |  crequestrate > 8.41121
|  |  |  |  |  |  |  frequestrate <= 107.254: S
|  |  |  |  |  |  |  frequestrate > 107.254: J
|  |  |  |  |  childFatherNumber > 8: J
|  |  |  |  fupdaterate > 542
|  |  |  |  |  crequestrate <= 7.55268: J
|  |  |  |  |  crequestrate > 7.55268
|  |  |  |  |  |  fupdaterate <= 907: S
|  |  |  |  |  |  fupdaterate > 907: J
|  fatherChildNumber > 2
|  |  fatherChildNumber <= 8
|  |  |  cupdaterate <= 201
|  |  |  |  cupdaterate <= 9
|  |  |  |  |  crequestrate <= 8.38396: J
|  |  |  |  |  crequestrate > 8.38396
|  |  |  |  |  |  cupdaterate <= 4: S
|  |  |  |  |  |  cupdaterate > 4: J
|  |  |  |  cupdaterate > 9: S
|  |  |  cupdaterate > 201: J
|  |  fatherChildNumber > 8: S

Number of Leaves  :  26
Size of the tree :  51
```

A.2.2 Entire/updateRate

```
Attributes:  7
          frequestrate
          crequestrate
          fatherChildNumber
          childFatherNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.2.3 Entire/requestRate

```
Attributes:  7
          fupdaterate
          cupdaterate
          fatherChildNumber
          childFatherNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9
|  fatherSize <= 53
|  |  fatherSize <= 13
|  |  |  childFatherNumber <= 5
|  |  |  |  childSize <= 1716: J
|  |  |  |  childSize > 1716: S
|  |  |  childFatherNumber > 5: S
|  |  fatherSize > 13: J
|  fatherSize > 53
|  |  fatherChildNumber <= 5: J
|  |  fatherChildNumber > 5
|  |  |  fatherChildNumber <= 6
|  |  |  |  childSize <= 1908: J
|  |  |  |  childSize > 1908: S
|  |  |  fatherChildNumber > 6: J
fatherChildNumber > 9
|  childFatherNumber <= 2: J
|  childFatherNumber > 2
|  |  fatherChildNumber <= 11: S
|  |  fatherChildNumber > 11
|   |   |   childFatherNumber <= 7
|   |   |   |   childSize <= 1618: S
|   |   |   |   childSize > 1618: J
|   |   |   childFatherNumber > 7: J

Number of Leaves  :  13
Size of the tree :  25
```

A.2.4 Entire/size

```
Attributes:  7
          fupdaterate
          cupdaterate
          frequestrate
          crequestrate
          fatherChildNumber
          childFatherNumber
          improvingclass

J48 pruned tree
------------------

frequestrate <= 68.8852: J
frequestrate > 68.8852
|   fatherChildNumber <= 4: J
|   fatherChildNumber > 4
|   |   cupdaterate <= 201: S
|   |   cupdaterate > 201: J

Number of Leaves  :  4
Size of the tree :  7
```

A.2.5 Entire/childrenNumber

```
Attributes:  8
          fupdaterate
          cupdaterate
          frequestrate
          crequestrate
          childFatherNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
```

```
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.2.6 Entire/fatherNumber

```
Attributes:  8
          fupdaterate
          cupdaterate
          frequestrate
          crequestrate
          fatherChildNumber
          fatherSize
          childSize
          improvingclass

J48 pruned tree
------------------

frequestrate <= 68.8852: J
frequestrate > 68.8852
|  fatherChildNumber <= 4: J
|  fatherChildNumber > 4
|  |  cupdaterate <= 201: S
|  |  cupdaterate > 201: J

Number of Leaves  :  4
Size of the tree :  7
```

A.2.7 Ratio/∅

```
Attributes:  6
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9
```

```
|  fatherChildNumber <= 7: J
|  fatherChildNumber > 7
|  |  childFatherNumber <= 4
|  |  |  childFatherNumber <= 2
|  |  |  |  updaterate <= 0.5578: S
|  |  |  |  updaterate > 0.5578: J
|  |  |  childFatherNumber > 2
|  |  |  |  childFatherNumber <= 3
|  |  |  |  |  Size <= 0.2067: S
|  |  |  |  |  Size > 0.2067: J
|  |  |  |  childFatherNumber > 3: J
|  |  childFatherNumber > 4
|  |  |  fatherChildNumber <= 8
|  |  |  |  requestrate <= 45.467104
|  |  |  |  |  childFatherNumber <= 7
|  |  |  |  |  |  requestrate <= 4.402947: J
|  |  |  |  |  |  requestrate > 4.402947: S
|  |  |  |  |  childFatherNumber > 7
|  |  |  |  |  |  Size <= 0.3088: J
|  |  |  |  |  |  Size > 0.3088
|  |  |  |  |  |  |  childFatherNumber <= 8
|  |  |  |  |  |  |  |  Size <= 0.5522: S
|  |  |  |  |  |  |  |  Size > 0.5522
|  |  |  |  |  |  |  |  |  requestrate <= 16.056561: J
|  |  |  |  |  |  |  |  |  requestrate > 16.056561: S
|  |  |  |  |  |  |  childFatherNumber > 8: S
|  |  |  |  requestrate > 45.467104
|  |  |  |  |  childFatherNumber <= 8: J
|  |  |  |  |  childFatherNumber > 8: S
|  |  |  fatherChildNumber > 8: J
fatherChildNumber > 9
|  childFatherNumber <= 2: J
|  childFatherNumber > 2
|  |  fatherChildNumber <= 11
|  |  |  Size <= 2.6814
|  |  |  |  Size <= 1.6844
|  |  |  |  |  requestrate <= 42.424847
|  |  |  |  |  |  childFatherNumber <= 5
|  |  |  |  |  |  |  fatherChildNumber <= 10: S
|  |  |  |  |  |  |  fatherChildNumber > 10: J
|  |  |  |  |  |  childFatherNumber > 5: J
|  |  |  |  |  requestrate > 42.424847: S
|  |  |  |  Size > 1.6844: J
|  |  |  Size > 2.6814: S
|  |  fatherChildNumber > 11
|   |   |   childFatherNumber <= 7
|   |   |   |   Size <= 0.8286: J
|   |   |   |   Size > 0.8286: S
|   |   |   childFatherNumber > 7: J

Number of Leaves  :  26
Size of the tree :  51
```

A.2.8 Ratio/updateRate

```
Attributes:  5
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9
|   fatherChildNumber <= 7: J
|   fatherChildNumber > 7
|   |   childFatherNumber <= 4: J
|   |   childFatherNumber > 4
|   |   |   fatherChildNumber <= 8
|   |   |   |   requestrate <= 45.467104
|   |   |   |   |   childFatherNumber <= 7
|   |   |   |   |   |   requestrate <= 4.402947: J
|   |   |   |   |   |   requestrate > 4.402947: S
|   |   |   |   |   childFatherNumber > 7
|   |   |   |   |   |   Size <= 0.3088: J
|   |   |   |   |   |   Size > 0.3088
|   |   |   |   |   |   |   childFatherNumber <= 8
|   |   |   |   |   |   |   |   Size <= 0.5522: S
|   |   |   |   |   |   |   |   Size > 0.5522
|   |   |   |   |   |   |   |   |   requestrate <= 16.056561: J
|   |   |   |   |   |   |   |   |   requestrate > 16.056561: S
|   |   |   |   |   |   |   childFatherNumber > 8: S
|   |   |   |   requestrate > 45.467104
|   |   |   |   |   childFatherNumber <= 8: J
|   |   |   |   |   childFatherNumber > 8: S
|   |   |   fatherChildNumber > 8: J
fatherChildNumber > 9
|   childFatherNumber <= 2: J
|  childFatherNumber > 2
|  |  fatherChildNumber <= 11
|  |  |  Size <= 2.6814
|  |  |  |  Size <= 1.6844
|  |  |  |  |  requestrate <= 42.424847
|  |  |  |  |  |  childFatherNumber <= 5
|  |  |  |  |  |  |  fatherChildNumber <= 10: S
|  |  |  |  |  |  |  fatherChildNumber > 10: J
|  |  |  |  |  |  childFatherNumber > 5: J
|  |  |  |  |  requestrate > 42.424847: S
|  |  |  |  Size > 1.6844: J
|  |  |  Size > 2.6814: S
|  |  fatherChildNumber > 11
|  |  |  childFatherNumber <= 7
|  |  |  |  Size <= 0.8286: J
|  |  |  |  Size > 0.8286: S
|  |  |  childFatherNumber > 7: J

Number of Leaves  :  22
Size of the tree :  43
```

A.2.9 Ratio/requestRate

```
Attributes:  5
          updaterate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|  childFatherNumber <= 2: J
|  childFatherNumber > 2
|  |  fatherChildNumber <= 11
|  |  |  Size <= 2.6814
|  |  |  |  Size <= 1.6844: S
|  |  |  |  Size > 1.6844: J
|  |  |  Size > 2.6814: S
|  |  fatherChildNumber > 11
|  |  |  childFatherNumber <= 7
|  |  |  |  Size <= 0.8286: J
|   |   |   |   Size > 0.8286: S
|   |   |   childFatherNumber > 7: J

Number of Leaves  :  8
Size of the tree :  15
```

A.2.10 Ratio/size

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9
|   fatherChildNumber <= 7: J
|   fatherChildNumber > 7
|   |   childFatherNumber <= 4
|   |   |   childFatherNumber <= 2
|   |   |   |   updaterate <= 0.5578: S
|   |   |   |   updaterate > 0.5578: J
|   |   |   childFatherNumber > 2: J
|   |   childFatherNumber > 4
|   |   |   fatherChildNumber <= 8
|   |   |   |   requestrate <= 45.467104
|   |   |   |   |   childFatherNumber <= 7
|   |   |   |   |   |   requestrate <= 4.402947: J
|   |   |   |   |   |   requestrate > 4.402947: S
|   |   |   |   |   childFatherNumber > 7: J
|   |   |   |   requestrate > 45.467104
|   |   |   |   |   childFatherNumber <= 8: J
|   |   |   |   |   childFatherNumber > 8: S
|   |   |   fatherChildNumber > 8: J
fatherChildNumber > 9
|   childFatherNumber <= 2: J
|   childFatherNumber > 2
|   |   fatherChildNumber <= 11
|   |   |   fatherChildNumber <= 10
|   |   |   |   updaterate <= 1.6756: S
|   |   |   |   updaterate > 1.6756: J
|   |   |   fatherChildNumber > 10
|   |   |   |   childFatherNumber <= 5
|   |   |   |   |   requestrate <= 58.754758: J
|   |   |   |   |   requestrate > 58.754758: S
|   |   |   |   childFatherNumber > 5: S
|   |   fatherChildNumber > 11
|   |   |   childFatherNumber <= 7: S
|   |   |   childFatherNumber > 7: J

Number of Leaves  :   18
Size of the tree :   35
```

A.2.11 Ratio/childrenNumber

```
Attributes:   5
          updaterate
          requestrate
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :   1
Size of the tree :   1
```

A.2.12 Ratio/fatherNumber

```
Attributes:   5
          updaterate
          requestrate
          fatherChildNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :   1
Size of the tree :   1
```

A.2.13 Differences/∅

```
Attributes:   6
```

```
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9
|   requestrate <= 76.371375: J
|   requestrate > 76.371375
|   |   fatherChildNumber <= 5: J
|   |   fatherChildNumber > 5
|   |   |   updaterate <= 381: J
|   |   |   updaterate > 381: S
fatherChildNumber > 9
|   childFatherNumber <= 2: J
|   childFatherNumber > 2
|   |   fatherChildNumber <= 11: S
|   |   fatherChildNumber > 11
|   |   |   childFatherNumber <= 7
|   |   |   |   Size <= -277: J
|   |   |   |   Size > -277: S
|   |   |   childFatherNumber > 7: J

Number of Leaves  :  9
Size of the tree :  17
```

A.2.14 Differences/updateRate

```
Attributes:  5
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|   childFatherNumber <= 2: J
|  childFatherNumber > 2
|  |  fatherChildNumber <= 11: S
|  |  fatherChildNumber > 11
|  |  |  childFatherNumber <= 7
|  |  |  |  Size <= -277: J
|  |  |  |  Size > -277: S
|  |  |  childFatherNumber > 7: J

Number of Leaves  :  6
Size of the tree :  11
```

A.2.15 Differences/requestRate

```
Attributes:  5
          updaterate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|  childFatherNumber <= 2: J
|  childFatherNumber > 2
|  |  fatherChildNumber <= 11: S
|  |  fatherChildNumber > 11
|  |  |  childFatherNumber <= 7
|  |  |  |  Size <= -277: J
|  |  |  |  Size > -277: S
|  |  |  childFatherNumber > 7: J

Number of Leaves  :  6
Size of the tree :  11
```

A.2.16 Differences/size

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          improvingclass
```

```
J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.2.17 Differences/childrenNumber

```
Attributes:  5
          updaterate
          requestrate
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.2.18 Differences/fatherNumber

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.2.19 Distance/∅

```
Attributes:  6
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|   childFatherNumber <= 2: J
|   childFatherNumber > 2
|   |   fatherChildNumber <= 10
|   |   |   updaterate <= 429: S
|   |   |   updaterate > 429: J
|   |   fatherChildNumber > 10: S

Number of Leaves  :  5
Size of the tree :  9
```

A.2.20 Distance/updateRate

```
Attributes:  5
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|   childFatherNumber <= 2: J
|   childFatherNumber > 2: S

Number of Leaves  :  3
Size of the tree :  5
```

A.2.21 Distance/requestRate

```
Attributes:  5
          requestrate
          fatherChildNumber
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|   childFatherNumber <= 2: J
|   childFatherNumber > 2: S

Number of Leaves  :  3
Size of the tree :  5
```

A.2.22 Distance/size

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          childFatherNumber
          improvingclass

J48 pruned tree
------------------

fatherChildNumber <= 9: J
fatherChildNumber > 9
|   childFatherNumber <= 2: J
|   childFatherNumber > 2
|   |   fatherChildNumber <= 10
|   |   |   updaterate <= 429: S
|   |   |   updaterate > 429: J
|   |   fatherChildNumber > 10: S

Number of Leaves  :  5
Size of the tree :  9
```

A.2.23 Distance/childrenNumber

```
Attributes:  5
          updaterate
          requestrate
          childFatherNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

A.2.24 Distance/fatherNumber

```
Attributes:  5
          updaterate
          requestrate
          fatherChildNumber
          Size
          improvingclass

J48 pruned tree
------------------
: J

Number of Leaves  :  1
Size of the tree :  1
```

B

Synthetic content page model features

Table B.1. Features of the *nytimes* content page model extracted from *The New York Times* web site.

Parameter	Value
min_{size}	138 bytes
max_{size}	97008 bytes
min_{req}	–
max_{req}	–
min_{upd}	–
max_{upd}	–
$total_{ce}$	3082
$total_{ag}$	13979
web pages number	482

Table B.2. Features of the *pageflakes* content page model extracted from *PageFlakes* web site.

Parameter	Value
min_{size}	391 bytes
max_{size}	3633 bytes
min_{req}	–
max_{req}	–
min_{upd}	–
max_{upd}	–
$total_{ce}$	16803
$total_{ag}$	24771
web pages number	2000

Table B.3. Features of the *yahoopipes* content page model extracted from *Yahoo Pipes!* web site.

Parameter	Value
min_{size}	1008 bytes
max_{size}	62093 bytes
min_{req}	–
max_{req}	–
min_{upd}	–
max_{upd}	–
$total_{ce}$	9182
$total_{ag}$	14000
web pages number	2000

Table B.4. Parametrizations used for the creation of the synthetic web page model.

Parameter	Value
min_{size}	138 bytes
max_{size}	97008 bytes
min_{req}	2.0 req./s
max_{req}	10.0 req./s
min_{upd}	0.5 req./s
max_{upd}	2.5 req./s
$total_{ce}$	29067
$total_{ag}$	52755

List of Figures

2.1 Architecture of a content aggregation system in which the content assemblies take place in the application server. 9
2.2 Architecture of a content aggregation system in which the content assemblies take place in the cache server. 10

3.1 Research methodology of the dissertation. 24
3.2 Architecture for adaptive content fragment framework. 25
3.3 Content designs for the benchmarked web site. 26
3.4 Example of content aggregation system web page (iGoogle [32]). 37
3.5 Example of content aggregation graph structure. 37
3.6 Example of extended content aggregation graph structure. 39

4.1 Example of the inputs and outputs of the adaptive core. 44
4.2 An overview of the steps that composes the Knowledge Discovery process (source: [27]). 49
4.3 Example of a multivariate scatter plot representation. 56
4.4 Graphical representation of the association rules for the preliminary test. 57

5.1 Architecture of the adaptive content fragment framework. 65
5.2 Sequence diagrams for the communication between the content aggregation system tiers and the adaptive system. 66
5.3 Sequence diagram for an ESI-enabled proxy cache. 70
5.4 Example of fragment element identification. 71
5.5 Architecture of Drupal with the modules for COFRADIAS. 73

6.1 Architecture of the test-bed. 91

7.1 Decision tree corresponding to the *ratio/∅* of the *distributed* hardware architecture. 97
7.2 Speed-up of the user perceived latency in experiment subset 1A. 102

7.3 User-perceived latencies of the experiment subset 1A for the experiment executions *split* and *entire/∅*. 104
7.4 Speed-up of the user perceived latency in experiment subset 1B. 106
7.5 Speed-up of the user perceived latency in experiment subset 1C. 108
7.6 Speed-up of the user perceived latency in experiment subset 2A. 110
7.7 Speed-up of the user perceived latency in experiment subset 2B. 113
7.8 Speed-up of the user perceived latency in experiment subset 2C. 115
7.9 Speed-up of the user perceived latency in experiment subset 3A. 119
7.10 Speed-up of the user perceived latency in experiment subset 3B. 121
7.11 Speed-up of the user perceived latency in experiment subset 4A. 124
7.12 Cache performance for the executions of the experiment subset 4A. 127
7.13 Speed-up of the user perceived latency in experiment subset 5A. 128
7.14 Cache performance for the executions of the experiment subset 5A. 131

8.1 CPU utilization percentage for *entire/∅* in experiment set 4. . . . 146
8.2 CPU utilization percentage for the *split* scenario in experiment set 4. 147
8.3 Comparison of the overhead of *entire/∅* with the *split* scenario in experiment set 4. 148
8.4 CPU utilization percentage for *entire/∅* in experiment set 5. . . . 151
8.5 CPU utilization percentage for MACE in experiment set 5. 153
8.6 Comparison of the overhead of *entire/∅* scenario with MACE. . . 155
8.7 Adaptive core overhead for *entire/∅* scenario and MACE. 156

List of Tables

3.1 Summarized latencies for the benchmark to study the assembly points with the highest performance. 27
3.2 Statistical distributions for the characterization of a web fragment cache. 31
3.3 Correlation analysis for the results obtained in the simulator. . . 34
3.4 Correlation analysis for the results obtained in the test-bed executions. 35

4.1 Parametrization of the process to create the training data set. . . 53
4.2 Data instances representation for the training data sets. 59
4.3 Comparative analysis among data mining algorithms. 60

6.1 Hardware characteristics of the *centralized* hardware architecture. 82
6.2 Hardware characteristics of the *distributed* hardware architecture. 82
6.3 Features of the experiment design. 87
6.4 Design of the experiment set 1. 87
6.5 Design of the experiment set 2. 88
6.6 Design of the experiment set 3. 88
6.7 Design of the experiment set 4. 89
6.8 Design of the experiment set 5. 89

7.1 Decision trees obtained from the *distributed* architecture. 96
7.2 Decision trees obtained from the *centralized* architecture. 98
7.3 Speed-ups of the user-perceived latency for experiment subset 1A. 103
7.4 Cache performance for the executions of the experiment subset 1A. 105
7.5 Speed-ups of the user-perceived latency for experiment subset 1B. 105

7.6 Cache performance for the executions of the experiment subset 1B. 107
7.7 Speed-ups of the user-perceived latency for experiment subset 1C. 107
7.8 Cache performance for the executions of the experiment subset 1C. 109
7.9 Speed-ups of the user-perceived latency for experiment subset 2A. 111
7.10 Cache performance for the executions of the experiment subset 2A. 112
7.11 Speed-ups of the user-perceived latency for experiment subset 2B. 114
7.12 Cache performance for the executions of the experiment subset 2B. 114
7.13 Speed-ups of the user-perceived latency for experiment subset 2C. 116
7.14 Cache performance for the executions of the experiment subset 2C. 117
7.15 Speed-ups of the user-perceived latency for experiment subset 3A. 118
7.16 Cache performance for the executions of the experiment subset 3A. 120
7.17 Speed-ups of the user-perceived latency for experiment subset 3B. 122
7.18 Cache performance for the executions of the experiment subset 3B. 123
7.19 Speed-ups of the user-perceived latency for experiment subset 4A. 125
7.20 Speed-ups of the user-perceived latency for experiment subset 5A. 129
7.21 Cache performance for the executions of the experiment subset 4A (*entire/∅* experiments). 134
7.22 Cache performance for the executions of the experiment subset 4A (*split* experiments). 135
7.23 Cache performance for the executions of the experiment subset 5A (*entire/∅* experiments). 136
7.24 Cache performance for the executions of the experiment subset 5A (MACE experiments). 137

8.1 Global utilization ratios (%) of the different application process for *entire/∅* in experiment subset 4A. 141
8.2 Global utilization ratios (%) of the different application processes for the *split* scenario in experiment subset 4A. 142
8.3 Particular utilization ratios (%) of the different application processes for *entire/∅* in experiment subset 4A. 143

8.4 Particular Utilization ratios (%) of the different application processes for the *split* scenario in experiment subset 4A. 144
8.5 Global utilization ratios (%) of the different application processes for *entire/∅* in experiment subset 5A. 149
8.6 Global utilization ratios (%) of the different application processes for MACE in experiment subset 5A. 150
8.7 Particular utilization ratios (%) of the different application processes for *entire/∅* in experiment subset 5A. 152
8.8 Particular utilization ratios (%) of the different application processes for MACE in experiment subset 5A. 154

B.1 Features of the *nytimes* content page model extracted from *The New York Times* web site. 201
B.2 Features of the *pageflakes* content page model extracted from *PageFlakes* web site. 201
B.3 Features of the *yahoopipes* content page model extracted from *Yahoo Pipes!* web site. 202
B.4 Parametrizations used for the creation of the synthetic web page model. 202

References

1. S. Allamaraju. *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. O'Reilly Series. O'Reilly Media, 2010.
2. Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *SIGMETRICS Perform. Eval. Rev.*, 27:3–11, March 2000.
3. Daniel A. Ashlock. *Evolutionary computation for modeling and optimization*. Springer, 2006.
4. A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.
5. Fabrício Benevenuto, Adriano Pereira, Tiago Rodrigues, Virgílio Almeida, Jussara Almeida, and Marcos Gonalves. Characterization and analysis of user profiles in online video sharing systems. *Journal of Information and Data Management*, 1(2):115–129, June 2010.
6. Fabricio Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgilio Almeida. Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 49–62, New York, NY, USA, 2009. ACM.
7. B. Bollobas. *Random Graphs*. Cambridge University Press, 2001.
8. Francesco Bonchi, Fosca Giannotti, Giuseppe Manco, Mirco Nanni, Dino Pedreschi, Chiara Renso, and Salvatore Ruggieri. Data mining for intelligent web caching. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, pages 599–603, Washington, DC, USA, 2001. IEEE Computer Society.
9. J.A. Bondy and U.S.R Murty. *Graph Theory*. Springer Publishing Company, Incorporated, 1st edition, 2008.
10. A. Broder. Graph structure in the Web. *Computer Networks*, 33(1-6):309–320, June 2000.
11. Daniel Brodie, Amrish Gupta, and Weisong Shi. Accelerating dynamic web content delivery using keyword-based fragment detection. *J. Web Eng.*, 4(1):79–100, 2005.
12. Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technology and Systems*, pages 193–206, 1997.

13. L.G. Cardenas, Julio Sahuquillo, A. Pont, and J.A. Gil. The multikey Web cache simulator: a platform for designing proxy cache management techniques. In *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pages 390 – 397, feb. 2004.
14. Emiliano Casalicchio. *Cluster-based Web Systems: Paradigms and Dispatching Algorithms.* PhD dissertation, Università degli Studi di Rome Tor Vergata, Rome,IT, 2002.
15. Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 1–14, New York, NY, USA, 2007. ACM.
16. Jim Challenger, Paul Dantzig, Arun Iyengar, and Karen Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions Internet Technologies*, 5:359–389, May 2005.
17. Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, HPCN Europe 2001, pages 114–123, London, UK, 2001. Springer-Verlag.
18. B. Clifton. *Advanced Web metrics with Google Analytics.* Serious skills. Wiley Pub., 2008.
19. World Wide Web Consortium. Esi language specification 1.0. Technical report, http://www.w3.org/TR/esi-lang, 2001.
20. Oracle Corp. Oracle web cache. Technical report, http://www.oracle.com/technetwork/middleware/ias/index-089317.html, January 2011.
21. Anindya Datta, Kaushik Dutta, Helen Thomas, Debra VanderMeer, Suresha, and Krithi Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 97–108, New York, NY, USA, 2002. ACM.
22. Brian D. Davison. Ncs: Network and cache simulator – an introduction. Technical report, 2001.
23. Fernando Duarte, Bernardo Mattos, Azer Bestavros, Virgilio Almeida, and Jussara Almeida. Traffic characteristics and communication patterns in blogosphere. In *Proceedings of the International AAAI Conference on Weblogs and Social Media*, 2007.
24. P. Erdös and A. Rényi. On random graphs, I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.
25. P. Erdös and A. Rényi. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.
26. Joan Espina. *Extension development to allow Drupal to manage content aggregations and to adapt content fragment designs.* Master dissertation, Universitat de les Illes Balears, Palma, SPAIN, 2010.
27. Usama Fayyad, Gregory Piatetsky-shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17:37–54, 1996.
28. Yingjie Fu, Haohuan Fu, and Pui on Au. An integration approach of data mining with web cache pre-fetching. In Jiannong Cao, Laurence Tianruo Yang,

Minyi Guo, and Francis Chi-Moon Lau, editors, *Parallel and Distributed Processing and Applications, Second InternationalSymposium, ISPA 2004, Hong Kong, China, December 13-15, 2004, Proceedings*, volume 3358 of *Lecture Notes in Computer Science*, pages 59–63. Springer, 2004.

29. Syam Gadde. Proxycizer. http://www.cs.duke.edu/ari/cisi/proxycizer/. Technical report, Duke University Computer Science Department, 2001.
30. Jose Maria Gago, Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Web mining service (wms), a public and free service for web data mining. *Internet and Web Applications and Services, International Conference on*, 0:351–356, 2009.
31. Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28, New York, NY, USA, 2007. ACM.
32. Google Corp. igoogle. URL http://www.google.com/ig.
33. William Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2001.
34. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Estudio de viabilidad de esi en aplicaciones web dinámicas. In *Conferencia IADIS Ibero-Americana WWW/Internet 2006*, 2006.
35. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Dynamic web fragment architecture. In J. L. lvarez, J. L. Arjona, R. Corchuelo, and D. Ruiz, editors, *Actas de los Talleres de las Jornadas de Ingeniera del Software y Bases de Datos (TJISBD)*, volume 1. Sistedes, 2007.
36. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. The Applicability of Balanced ESI for Web Caching - A proposed algorithm and a case of study. In *WEBIST 2007 Third International Conference on Web Information Systems and Technologies*, pages 197–203. 2007.
37. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Using Ontologies to Improve Performance in a Web System. A Web Caching System Case of Study. In *Proceedings of the Fourth International Conference on Web Information Systems and Technologies WEBIST 2008*, pages 117–122. 2008.
38. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Web performance and behavior ontology. *Complex, Intelligent and Software Intensive Systems, International Conference*, 0:219–225, 2008.
39. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Web performance engineering based on ontological languages and semantic web. *Int. J. Comput. Appl. Technol.*, 33:300–311, January 2008.
40. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Rule-based system to improve performance on mash-up web applications. In *DCAI*, volume 79 of *Advances in Soft Computing*, pages 577–584. Springer, 2010.
41. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Web cache performance correlation with content characterization parameters in content aggregation systems. In *Proceedings of the XXXVIth Latin American Informatics Conference*, 2010.
42. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Evaluation of a fragment-optimized content aggregation web system. In *The Fourth International Conference on Internet Technologies and Applications (ITA 2011)*, pages 48–55. Glyndwr University, April 2011.

43. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Improving web cache performance via adaptive content fragmentation design. In *Proceedings of The Tenth IEEE International Symposium on Networking Computing and Applications, NCA 2011*, pages 310–313. IEEE Computer Society, 2011.
44. Carlos Guerrero, Carlos Juiz, and Ramon Puigjaner. Mining web usage and content structure data to improve web cache performance in content aggregation systems. In *The Sixth International Conference on Digital Telecommunications (ICDT 2011)*, pages 123–130. IARIA, April 2011.
45. Lei Guo, Enhua Tan, Songqing Chen, Xiaodong Zhang, and Yihong (Eric) Zhao. Analyzing patterns of user content generation in online social networks. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 369–378, New York, NY, USA, 2009. ACM.
46. Emily Halili. *Apache JMeter*. Packt Publishing, 2008.
47. David J. Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. The MIT Press, August 2001.
48. Osama Al-Haj Hassan, L Ramaswamy, and J A Miller. The mace approach for caching mashups. *International Journal of Web Services Research*, 7(4):64–88, 2010.
49. Osama Al-Haj Hassan, Lakshmish Ramaswamy, and John A. Miller. Mace: A dynamic caching framework for mashups. In *Proceedings of the 2009 IEEE International Conference on Web Services*, ICWS '09, pages 75–82, Washington, DC, USA, 2009. IEEE Computer Society.
50. Osama Al-Haj Hassan, Lakshmish Ramaswamy, and John A Miller. Mace: A dynamic caching framework for mashups. *2009 IEEE International Conference on Web Services*, pages 75–82, 2009.
51. Osama Al-Haj Hassan, Lakshmish Ramaswamy, and John A. Miller. The mace approach for caching mashups. *International Journal of Web Services Research*, 7(4):64–88, 2010.
52. Yin-Fu Huang and Jhao-Min Hsu. Mining web logs to improve hit ratios of prefetching and caching. *Know.-Based Syst.*, 21:62–69, February 2008.
53. Aye Aye Khaing and Ni Lar Thein. Efficiently creating dynamic web content: A fragment based approach. In *Proceedingsof the 6th Asia-Pacific Symposium on Information and Telecommunication Technologies*, pages 154 – 159, 2005.
54. Patrick Killelea. *Web Performance Tuning*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2002.
55. Patrick Killelea. *Web performance tuning - speeding up the web (2. ed.)*. O'Reilly, 2002.
56. Chetan Kumar and John B. Norris. A new approach for a proxy-level web caching mechanism. *Decis. Support Syst.*, 46:52–60, December 2008.
57. R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic models for the Web graph. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 57–65. IEEE Computer Society, 2000.
58. Rasmus Lerdorf, Kevin Tatroe, and Peter MacIntyre. *Programming PHP*. O'Reilly Media, Inc., 2 edition, April 2006.
59. Tjen-Sien Lim, Wei-Yin Loh, and Yu-Shan Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Mach. Learn.*, 40:203–228, September 2000.

60. Xiang Liu. Developing high performance applications with oracle9ias web cache and esi. URL http://www.oracle.com/technetwork/middleware/ias/9iaswebcache-esi-twp-133927.pdf.
61. Priyanka Makkar and Payal Gulati. A novel approach for predicting user behavior for improving web performance. *International Journal on Computer Science and Engineering*, 2:1233–1236, July 2010.
62. Johann Marquez, Josep Domenech, Jose Gil, and Ana Pont. A Web Caching and Prefetching Simulator. In *16th IEEE International Conference on Software, Telecommunications and Computer Networks, SoftCOM*, pages 346–350, Split, Croacia, September 2008. FESB.
63. Daniel Menascé and Virgilio Almeida. *Capacity planning for web services.* Prentice Hall, 2002.
64. E. Morton. Start-page smackdown: Netvibes, Pageflakes, iGoogle and Live.com. Online, 2008.
65. Mor Naaman, Hector Garcia-Molina, and Andreas Paepcke. Evaluation of esi and class-based delta encoding. In *8th International Workshop on Web Content Caching and Distribution (IWCW 2003)*, September 2003.
66. Pageflakes Inc. LiveUniverse. Pageflakes. URL http://www.pageflakes.com/.
67. George Pallis, Athena Vakali, and Jaroslav Pokorny. A clustering-based prefetching scheme on a web cache environment. *Comput. Electr. Eng.*, 34:309–323, July 2008.
68. G. Shiva Prasad, N. V. Subba Reddy, and U. Dinesh Acharya. Knowledge discovery from web usage data: A survey of web usage pre-processing techniques. In *Information Processing and Management*, volume 70 of *Communications in Computer and Information Science*, pages 505–507. Springer Berlin Heidelberg, 2010.
69. Konstantinos Psounis. Class-based delta-encoding: A scalable scheme for caching dynamic web content. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCSW '02, pages 799–805, Washington, DC, USA, 2002. IEEE Computer Society.
70. J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.
71. Michael Rabinovich, Zhen Xiao, Fred Douglis, and Chuck Kalmanek. Moving edge-side includes to the real edge: the clients. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.
72. Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic detection of fragments in dynamically generated web pages. In *In International World Wide Web Conference (WWW*, pages 443–454. ACM Press, 2004.
73. Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic fragment detection in dynamic web pages and its impact on caching. *IEEE Transactions on Knowledge and Data Engineering*, 17:859–874, 2005.
74. Arao Ramos, John Baxter Rollins, and David Giddens Wilhite. Smart data caching using data mining, 07 2009.
75. Pablo Rodriguez Roca. Lenguajes canónicos para la descripción de grafos: Estudio y transformación entre esquemas de distintos modelos de datos. Technical Report 0308 ISSN: 07976410, Instituto de Computación de la Facultad de Ingeniería y del Área Informática de la Universidad de la República Uruguay, 2002.

76. Fabian Schneider, Sachin Agarwal, Tansu Alpcan, and Anja Feldmann. The new web: Characterizing ajax traffic. In Mark Claypool and Steve Uhlig, editors, *PAM*, volume 4979 of *Lecture Notes in Computer Science*, pages 31–40. Springer, 2008.
77. Ric Shreves. 2008 Open Source CMS Market Share Survey. Technical report, 2008.
78. S. Sulaiman, S.M. Shamsuddin, and A. Abraham. Intelligent web caching using adaptive regression trees, splines, random forests and tree net. In *Data Mining and Optimization (DMO), 2011 3rd Conference on*, pages 108 –114, june 2011.
79. Suresha, Jayant, and Jayant R. Haritsa. On reducing dynamic web page construction times. In *Proceedings of 6th Asia-Pacific Web Conference*, pages 722–731, 2004.
80. The New York Times Company. The new york times. breaking news, world news and multimedia. URL http://www.nytimes.com/.
81. Alexander Totok. *Exploiting Service Usage Information for Optimizing Server Resource Management*. PhD dissertation, New York University, New York,USA, 2006.
82. Thomas Tullis and William Albert. *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics*. Morgan Kaufmann, Burlington, MA, USA, 2008.
83. Carsten Ullrich, Paul Libbrecht, Stefan Winterstein, and Martin Muhlenbrock. A flexible and efficient presentation-architecture for adaptive hypermedia: Description and technical evaluation. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies*, pages 21–25, Washington, DC, USA, 2004. IEEE Computer Society.
84. Weka Machine Learning Project. Weka. URL http://www.cs.waikato.ac.nz/˜ml/weka.
85. L Welicki and SanJuan Martinez. Improving performance and server resource usage with page fragment caching in distributed web servers. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA 2007, 2007.
86. Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
87. Roland P. Wooster and Marc Abrams. Proxy caching that estimates page load delays. In *Selected papers from the sixth international conference on World Wide Web*, pages 977–986, Essex, UK, 1997. Elsevier Science Publishers Ltd.
88. Yahoo! Inc. Pipes, rewrite the web. URL http://pipes.yahoo.com/pipes/.
89. Qiang Yang and Haining Henry Zhang. Web-log mining for predictive web caching. *IEEE Trans. on Knowl. and Data Eng.*, 15:1050–1053, July 2003.
90. Qiang Yang, Haining Henry Zhang, and Tianyi Li. Mining web logs for prediction models in www caching and prefetching. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, pages 473–478, New York, NY, USA, 2001. ACM.
91. Chun Yuan, Yu Chen, and Zheng Zhang. Evaluation of edge caching/offloading for dynamic content delivery. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 461–471, New York, NY, USA, 2003. ACM.

92. Chun Yuan, Zhigang Hua, and Zheng Zhang. Web content caching and distribution. chapter PROXY+: simple proxy augmentation for dynamic content processing, pages 91–108. Kluwer Academic Publishers, Norwell, MA, USA, 2004.