RC23423 (W0411-087) November 10, 2004
Computer Science

IBM Research Report

Techniques for Efficiently Serving and Caching Dynamic Web Content

Arun Iyengar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 703
Yorktown Heights, NY 10598

Lakshmish Ramaswamy
College of Computing
Georgia Tech
Atlanta, GA 30332

Bianca Schroeder
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Chapter 1

TECHNIQUES FOR EFFICIENTLY SERVING AND CACHING DYNAMIC WEB CONTENT

Arun Iyengar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
aruni@us.ibm.com

Lakshmish Ramaswamy
College of Computing, Georgia Tech
Atlanta, GA 30332
laks@cc.gatech.edu

Bianca Schroeder
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
bianca@cs.cmu.edu

Abstract This chapter presents an overview of techniques for efficiently serving and caching dynamic web data. We describe techniques for invoking server programs and architectures for serving dynamic web content. Caching is crucially important for improving the performance of Web sites generating significant dynamic data. We discuss techniques for caching dynamic Web data consistently. Fragment-based web publication and can significantly improve performance and increase the cacheability of dynamic web data. These techniques assume the existence of mechanisms for creating fragments. We discuss techniques for automatically detecting fragments in web pages.

It is often desirable to provide content with quality-of-service (QoS) guarantees. We examine techniques for providing QoS under overload conditions. We also look at techniques for providing differentiated QoS.

Keywords: caching, dynamic Web data, fragment detection, quality of service

1. Introduction

Dynamic data refers to content which changes frequently. Dynamic web content is created by programs which execute at the time a request is made. By contrast, static data is usually comprised of already existing files. Dynamic data requires more overhead to serve than static data; a request for a dynamic Web page might require orders of magnitude more CPU cycles to satisfy than a request for a static page. For web sites which generate significant amounts of dynamic data, the overhead for serving dynamic requests is often the performance bottleneck.

Another key problem presented by dynamic data is maintaining updated content. If a dynamic web page is changing frequently, then copies of the web page must be updated frequently to prevent stale data from being served. This chapter will discuss a number of techniques for maintaining consistent replicas of dynamic content. The dual problems of serving content efficiently and maintaining updated consistent data are what make dynamic data so challenging.

One method for improving performance which allows at least parts of dynamic web content to be cached is to generate web pages from fragments. A fragment is a portion of a web page which might recursively embed smaller fragments. Using the fragment-based approach, personalized, secure, or rapidly changing data can be encapsulated within fragments, allowing other parts of the web page to be stored in caches which have the ability to assemble web pages from fragments. This chapter discusses fragment-based web publication as well as techniques for automatically detecting fragments in web pages.

It is often in the economic interest of Web sites to offer some guarantee on the quality of service they deliver. Moreover, in many situations, it is desirable to provide different levels of quality of service, since requests vary in their importance and their tolerance to delays. We discuss techniques for both achieving system-wide performance guarantees and class-based service differentiation.

2. Architectures for Serving Dynamic Content

The most common method of generating dynamic content used to be by invoking server programs through the Common Gateway Interface (CGI). This method is inefficient because a new process needs to be invoked for each dynamic request. More sophisticated methods for creating dynamic content are now commonplace. Web servers generally provide interfaces for invoking server programs which incur significantly less overhead than CGI. One approach to implementing faster interfaces is for a server program to execute as part of a web server's process. This can be done by dynamically loading the server program the first time that it is invoked. Alternatively, a server program can be statically linked as part of the web server's executable image.

There are drawbacks to running a server program as part of a web server process, however. Using this approach, a server program could crash a web server or leak resources such as memory. In addition, the server program should be thread-safe so that another thread within the server process cannot affect the state of the server program in unpredictable ways; this can make writing server programs more difficult. Another approach which alleviates these problems to a large degree at the cost of slight overhead is for server programs to run as long-running processes independent from a web server's process. The server communicates with these long-running processes to invoke server programs.

One commonly used method for creating dynamic web content is via JavaServer Pages (JSP) technology. JSP pages consist of special tags in addition to standard HTML or XML tags. A JSP engine interprets the special tags and generates the content required. Subsequently, the results are sent back to the browser in the form of an HTML or XML page.

JSP pages may be compiled into Java platform servlet classes (A servlet is a program that runs on the server, compared with an applet which runs on a client browser. Servlets are handled by special threads within a web server process and run as part of the web server process.). A JSP page needs to be compiled the first time the page is accessed. The resulting Java servlet class can remain in memory so that subsequent requests for the page do not incur compilation overhead. A JSP page typically has the extension .jsp or .jspx to indicate to the web server that the JSP engine should be invoked to handle the page.

Microsoft has a similar approach for handling dynamic web content known as Active Server Pages (ASP). However, ASP technology is restricted to Microsoft platforms. JSP technology was developed using the Java Community Process and is available on a much wider variety of platforms than ASP technology.

Figure 1.1 shows an end-to-end flow of how a web request might proceed from a client to a server. After the request is made by a browser, it may go through a proxy server which is shared by several clients. Web content may be cached at several places within the network as shown in the figure. Caching can significantly reduce the latency for accessing remote data.

A proxy cache will store static data on behalf of several clients which share the use of the proxy. By contrast a content distribution network will cache data on behalf of content providers; content providers pay a content distribution network such as Akamai to cache content in geographically distributed locations so that a client can obtain a copy from a cache which is not too far away.

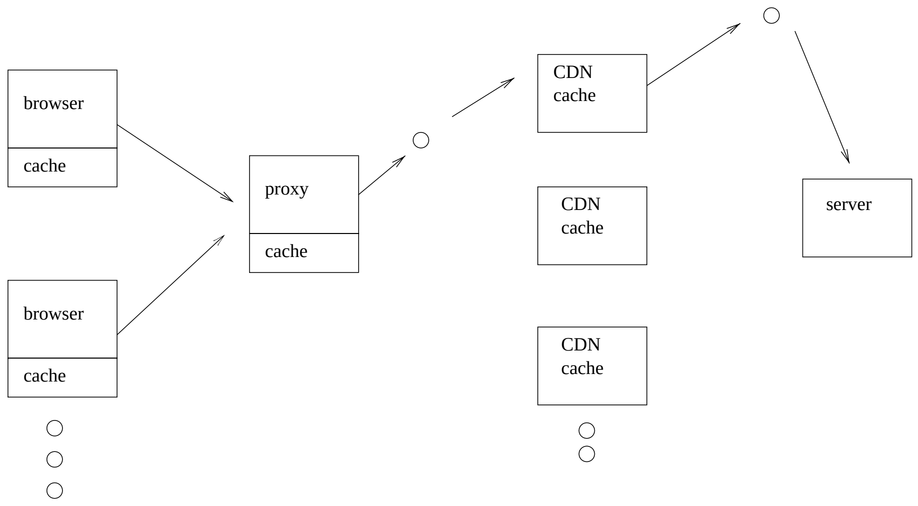


Figure 1.1. Path of a request from a client browser to a server. *CDN* stands for content distribution network.

A key problem with caching web data is maintaining consistency among multiple copies. Later in the chapter, we will discuss different cache consistency methods which can be used for caching dynamic data. Proxy caches and CDN's typically only cache static data.

A web site which serves dynamic content to a large number of clients will need several servers. Figure 1.2 depicts a three-tiered system for serving dynamic data at a web site. Requests come into a load balancer which sends requests to Tier 1 consisting of web servers. Static requests are handled by the web servers in Tier 1. Dynamic requests are handled by the application servers in Tier 2. Application servers may incur significant overhead in satisfying dynamic requests. Therefore, there should be enough application servers to prevent Tier 2 from becoming a bottleneck during periods of high request rates for dynamic data. Balancing resources among the tiers is important for preventing one of the tiers from becoming a bottleneck while not wasting money on excessive resources in a tier which are underutilized.

High availability is a critically important requirement for commercial web sites which should be robust in the presence of failures. Multiple servers improve availability as well as performance. If one server fails, requests can be directed to another server.

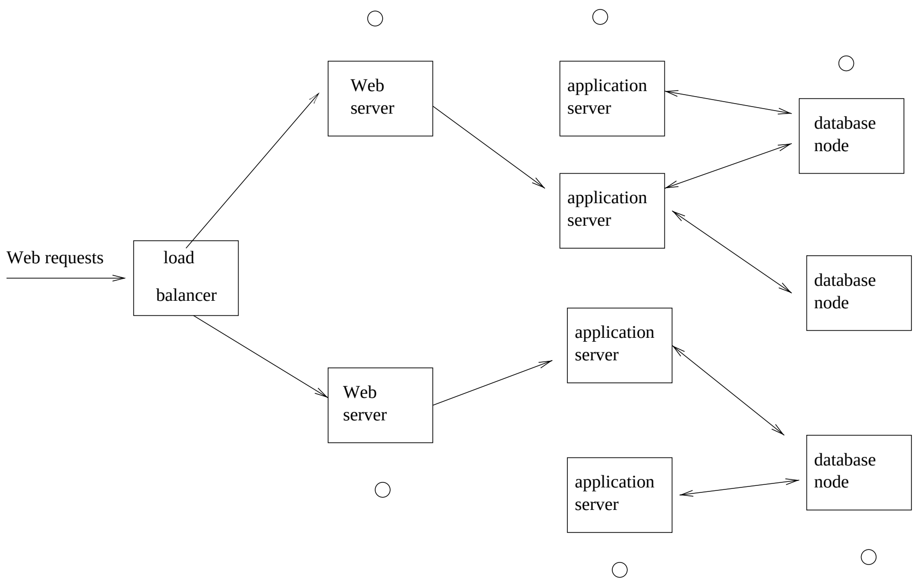


Figure 1.2. A three-tiered system for serving dynamic web content.

3. Consistently Caching Dynamic Web Data

Caching may take place at any of the tiers in Figure 1.2. For example, many databases employ some form of caching. A web site might also have one or more reverse proxy caches [37]. We now discuss techniques for achieving consistency among multiple cached copies.

Web objects which are cached may have expiration times associated with them. A cache continues to serve an object before its expiration time has elapsed. If a request is received for an object which has expired, the cache sends a get-if-modified-since request to the server; the server then either returns an updated copy of the object or indicates to the cache that the previous version is still valid. In either case, a new expiration time should be assigned to the object.

Expiration times require relatively low overhead for consistency maintenance. They only provide weak consistency, however. An application needs to know in advance when an object will expire, and this is not always possible. If the application overestimates the lifetime of an object, caches may serve obsolete copies. If the application underestimates the lifetime of an object, the cache will send extraneous authentication messages to the server which add overhead and increase latency for satisfying client requests.

There are a number of approaches for achieving stronger forms of cache consistency. In strong cache consistency, any cached copy of an object must be current; even a slight degree of inconsistency is not acceptable. A key problem with strong consistency is that updates require considerable overhead. Before performing an update, all cached copies of the object need to be invalidated first. Considerable delays can occur in making sure that all cached copies have been invalidated.

Strong consistency is often not feasible for web data because of the high overhead it entails. For much of the data on the web, slight degrees of inconsistency are tolerable. Therefore, true strong consistency is overkill for most web data. Other forms of consistency exist which offer stronger degrees of consistency than expiration times but don't have the overhead of strong consistency. These schemes generally make use of server-driven consistency or client polling.

In server-driven consistency, servers must notify caches when an object has changed. The notification message may either be a message to simply invalidate the object or it could contain a copy of the new object (prefetch). Prefetching is advantageous for hot objects which are highly likely to be accessed before they are updated again. For objects which are not accessed all that frequently relative to the update rate, prefetching is not a good idea because of the added bandwidth consumed by the prefetch; if the object is not accessed before the next update, then the prefetch will not have achieved anything useful.

Server-driven consistency has overhead when an update to a cached object is made. All caches storing the object need to be notified, and the number of update messages grows linearly with the number of caches which need to be notified. The server also needs to maintain information about which caches are storing which objects; this adds storage overhead.

In client polling, caches are responsible for contacting the server in order to determine if a cached object is still valid or not. Cache consistency managed via expiration times is a form of client polling in which the expiration time reduces the frequency of polling. If expiration times are very short, more frequent polling is required. In the worst case, polling is required on each request to a cache. The overhead for these polling message can be quite significant.

One method which can reduce the overhead for maintaining cache consistency is *leases*. In the lease-based approach, a server grants a lease to a cache for a duration. During the lease duration, the server must continue to send update messages to the cache. After the lease duration has expired, the server is no longer required to send update messages to the cache. If the cache wants to continue to receive update messages for an object, it must renew the lease. Leases combine elements of server-driven consistency and client polling. Note that if the lease duration is zero, the cache consistency scheme degenerates into pure client polling. If, on the other hand, the lease duration is infinite, the cache consistency scheme degenerates into pure server-driven consistency. Leases were used for distributed file cache consistency before they were applied to the web [23].

Leases provide a number of advantages. They bound the length of time that a server needs to provide updates to a cache. This is important because a cache might become unresponsive to a server or be taken off the network. The server might not know which caches are responsive and which ones are not. Leases provide an upper bound on how stale a validly cached object can be. In the worst case, a cached object is updated multiple times but a cache fails to receive any invalidation messages for the object due to an event such as a network failure. In this worst case scenario, the cache will still not serve a copy of the object obsolete by more than the lease duration.

There are a number of variations on leases which have been proposed. A *volume lease* is a single lease which is granted to a collection of several objects [40]. *Cooperative leases* have been proposed for CDN's and involve cooperation between multiple caches to reduce the overhead of consistency maintenance [31].

Determining how Changes to Underlying Data Affect Cached Objects

A key problem with generating and caching dynamic web data is determining which cached pages have changed when changes occur. Web pages are often constructed from underlying data in a nonobvious fashion. When the underlying data changes, several web pages may be affected. Determining the precise ones which change can be nontrivial. For example, a news web site might have information about the latest news stories, stock quotes, sporting events, and other similar items. Suppose the web pages are constructed from databases which store the latest information received. When new information is received, the database tables are updated. The problem then becomes how to determine which web pages need to be updated as a result of the new information.

Data update propagation (DUP) provides an effective solution to this problem. In data update propagation, correspondences between web pages and underlying data are maintained in an object dependence graph. When underlying data changes, graph traversal algorithms are applied to determine which web pages are affected by the change [13].

Figure 1.3 depicts a simple object dependence graph in which none of the nodes have both an incoming and outgoing edge. If underlying data $u1$ changes, then web pages $W1$, $W3$, and $W4$ also change. If $u2$ changes, then $W2$ also changes, while if $u3$ changes, then $W4$ is affected.

Figure 1.3 depicts a more general object dependence graph in which paths of length longer than one exist. If either $u1$ or $u3$ change, then both web pages $W1$ and $W2$ are affected.

The relationships between web pages and underlying data can change quite frequently. Object dependence graphs can thus be quite dynamic.

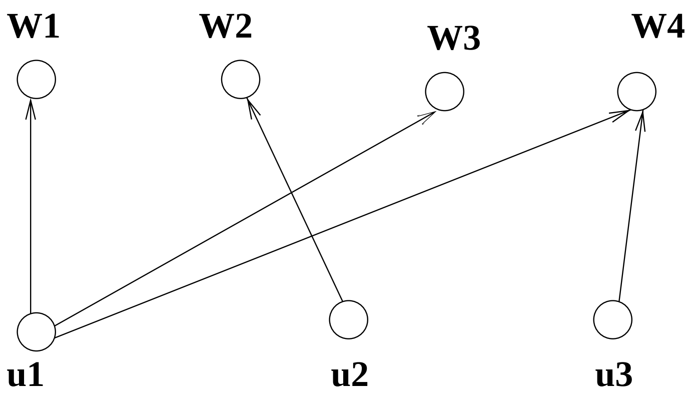


Figure 1.3. A simple object dependence graph representing data dependencies between web pages and underlying data.

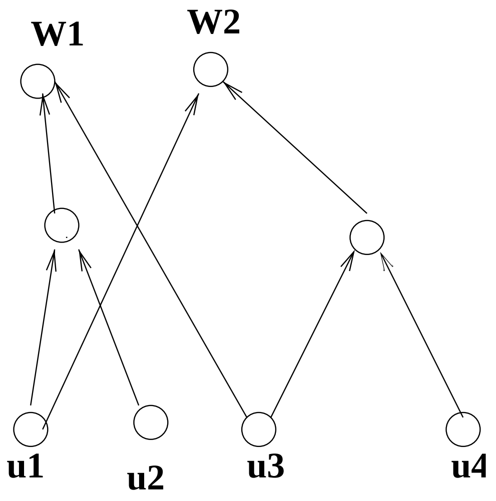


Figure 1.4. A more general object dependence graph representing data dependencies between web pages and underlying data.

4. Fragment-based Web Caching

The primary motivation for fragment-based web caching comes from some of the recent trends in web publishing. A considerable fraction of dynamic web pages exhibit the following distinct properties.

- Web pages rarely have a single theme or functionality. Most web pages have several document segments which differ in the information they provide or the functionality they encapsulate. For example, a web page from a news provider web site, in addition to containing an article about a news event, may also have links and synopses of other headlines of the day. Recent stock market data might also be listed on the web page. Further, the web page might also have a welcome bar containing personalized greetings to each registered user. These segments provide very different information, but are still present in the web page whose predominant theme is the news item with which it is associated.

- Most dynamic web pages contain a considerable amount of static content. The dynamic contents are often embedded in static web page segments. Similarly web pages are usually not completely personalized; Web pages generated for different users often share significant amounts of information.

- These different kinds of information may exhibit different lifetime and personalization characteristics. For example the stock market information in a web page might expire every few minutes whereas the synopses of headlines might change every few hours.

- Web pages hosted by the same web site tend to have similar structure and may exhibit considerable overlap in terms of HTML segments.

Fragment-based publishing, delivery and caching of web pages are designed with the aim of utilizing these characteristics to improve the cacheable content on the web and to reduce data invalidations at caches. In fragment-based dynamic web pages, various information segments are clearly identified and demarcated from one another.

Conceptually, a fragment is a portion of a web page which has a distinct theme or functionality associated with it and is distinguishable from the other parts of the page. In the fragment-based model, each fragment is an independent information entity. The web pages have reference to these fragments, which are served independently from the server and stored as such in the caches.

Figure 1.5 provides an example of a fragment-based web page. This was one of the web pages on a web site hosted by IBM for a major sporting event. The figure shows four fragments in the web page. Fragment-A lists some of

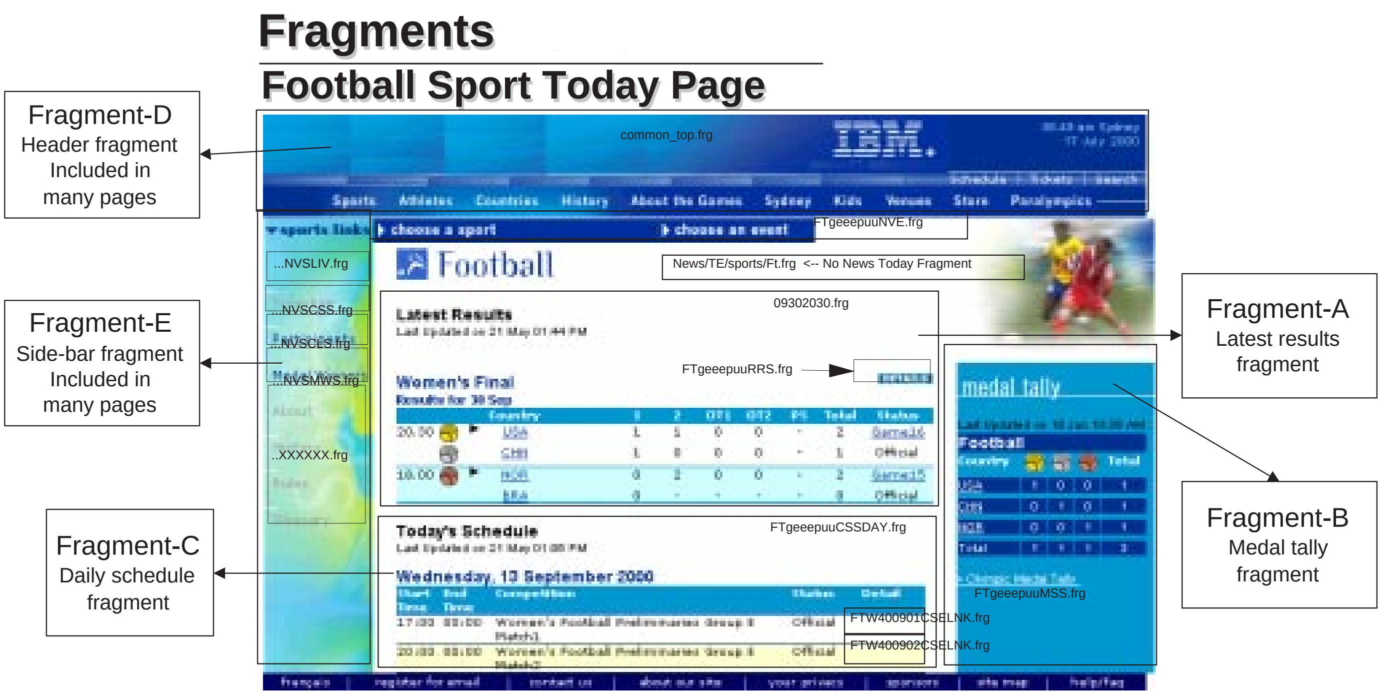


Figure 1.5. Fragments in a Web Page

the recent results of the event. Fragment-B indicates the current medal tally. Fragment-C shows the day's schedule of events. Fragment-D and Fragment-E are navigation bars aiding the users to navigate the web site. Note that the fragments shown in the figure have distinct themes, whereas the web page itself is a collage of various information pertaining to the sporting event.

As the fragments differ from one another in terms of the information and the functionality they provide, the properties associated with them are also likely to vary from one another. Important properties associated with a fragment include the time for which the information remains fresh, personalization characteristics such as whether the generated information is client-specific (based on cookies), and information sharing behavior exhibited by them. Fragment-based caching schemes recognize the variability of these fragment-specific properties. They try to improve cache performance by allowing the web page designer to specify cache directives such as cacheability and lifetime at the granularity of a fragment rather than requiring them to be specified at the web-page level. Specifying the cache directives at fragment-level provides a number of distinct advantages:

1 **Increases Cacheable Web Content:** A significant percentage of web pages contain some kind of personalized information. The personalized

information may be as simple as a welcome bar, or may be highly sensitive information like credit card or bank account details. This personalized information should not be cached for reasons of privacy and security, and hence has to be marked as non-cacheable. However, the pages that contain them usually also have non-personalized information. By specifying the properties at fragment level, only the personalized information need be marked as non-cacheable thereby permitting the caches to store the non-personalized content; this increases the cacheable content.

2 **Decreases Data Invalidations:** Fragments in the same web page might exhibit very different lifetime characteristics. In our example, Fragment-1 and Fragment-2 are likely to change more frequently than Fragment-3 which in turn changes less frequently than Fragments 4 or 5. When lifetimes are specified at the page level, the entire page gets invalidated when any of its fragments change. Therefore, the lifetime of a web page is dictated by the most frequently changing fragment contained in it. With fragment-based caching, the lifetime information can be specified at fragment-level. This allows the caches to invalidate only the fragment that has expired, and to fetch only the invalidated fragment from the origin server. The rest of the web page remains in the cache and need not be re-fetched from the server.

3 **Aids Efficient Disk Space Utilization:** Web pages from the same web site often share content. It might be in the form of same information being replicated on multiple web pages or it might be structural content such as navigation bars, headers and footers. In fragment-based publication, content which is shared across multiple web pages would typically be generated as fragments. These fragments are stored only once rather being replicated with each web page, thus reducing redundancy of information at the caches.

Researchers have proposed several flavors of fragment-based publishing, delivery and caching of web data. A fragment-based publishing system for dynamic web data is presented in [14]. This was one of the first works on the fragment-based web document model. Their system not only improves the performance of web-page construction, but also simplifies designing of web sites. The system allows embedding smaller and simpler fragments into larger and more complex fragments. This permits reuse of generated fragments, thereby avoiding unnecessary regeneration of the same information multiple times. Fragment-based publishing also facilitates designing of multiple web pages having a common look and feel, thereby simplifying the task of constructing large web sites. Further, if the information corresponding to a fragment changes, only that particular fragment needs to be updated rather than updating every single web page containing that fragment. This significantly reduces the overhead of maintaining complex web sites.

Datta et al. [20] argue that the dynamic nature of a web page is exhibited along two mutually orthogonal dimensions, namely, the layout of the web page and its content. They propose a fragment-based web caching scheme wherein the dynamic content is cached at proxy caches, whereas the layout of the web page is fetched from the server on each access to the web page.

When a request for a web page reaches a proxy cache, it is routed to the origin server. The origin server executes a script associated with the request, which generates the entire web page. A background process called back-end monitor (BEM) constantly monitors the script generating the web page. This process creates a layout for the page being generated. The background process creates the layout by removing those contents that are available at the proxy cache from the web page being generated by the application script. A reference containing an identifier of the removed fragment replaces the removed content. This document, which contains the layout information along with the fragments not available in the proxy cache, is sent to the proxy cache. The proxy cache, on receiving this document, parses it and introduces any missing fragments to generate the complete web page, which is then communicated to the user. While parsing the document received from the server, the proxy also stores fragments that are not available locally. This approach reduces the amount of data communicated from the server to the cache by transferring only the layout and the fragments that are not present in the cache.

Mohapatra and Chen [30] apply the concept of fragments for providing efficient QoS support and security mechanisms for web documents. They propose a system called WebGraph, which is a graphical representation of the containment relationship among weblets, which are analogous to fragments. The nodes of the graph correspond to weblets (fragments), and the edges represent the containment relationships. In the proposed system the QoS and security attributes can be specified at the granularity of a weblet. For example fragments may carry attributes such as delay sensitive, throughput sensitive, loss sensitive, etc. The edges of the WebGraph can also carry attributes indicating whether the edge can be dropped under certain circumstances like server overload or inability to support the QoS requirements of the associated fragments. If an edge is dropped, the corresponding fragment is not included in the web page.

Edge Side Includes: A Standard for Fragment-based Caching

Although researchers have shown the benefits of fragment-based caching, adopting it in commercial systems presents additional challenges. One of the major challenges is to evolve a common standard for fragment-based caching so that various proxies and servers implemented by different vendors become interoperable. Recognizing the need for standardizing fragment-based publishing, caching and delivery of web data, the Internet and Web communities have evolved a standard called the **Edge Side Includes (ESI)** [2]. ESI is an XML-based markup language that can be used by content providers to publish their web pages through fragments. The content providers can specify the fragments to be included in a web page. A cache supporting ESI understands that the specified fragment has to be included in the web page. It constructs the web page by inserting those fragments either from the cache, or by fetching them from the server.

ESI has been endorsed by companies like IBM, Akamai, Oracle and Digital Island. One of the goals of its design was to make it mark-up language and programming-model independent. The key functionalities provided by ESI are:

1 **Inclusion** The primary functionality of ESI is to provide support for specifying fragments to be inserted in a web page. The include element in the markup language is provided for this purpose. The content providers use the include element to instruct the caches to insert the indicated fragment in the web page. The caches which encounter an include statement check to see whether it is available locally. In case the fragment is not available at the cache, the fragment is fetched from the server. In addition to identifying the fragment, the include element also provides variable support. It can be used to specify the parameters to the script generating the web page.

2 **Conditional Inclusion/Exclusion:** Web page designers may want to include certain fragments when one or more conditions are satisfied, and may want to exclude them, or provide alternate fragments otherwise. The *choose*, *when* and *otherwise* clauses in the ESI specification provide support for such conditional inclusions and exclusion of fragments.

3 **Handling Exceptions:** At certain times, a particular fragment may become unavailable due to failure of the server or the network. ESI includes mechanisms to counter such exceptions through the *try*, *attempt* and *except* clauses. Web page designers can use these clauses to specify alternate resources, or default fragments to be used by the caches when such exceptions occur.

4 **Fragment Invalidation Support:** An important issue with caching dynamic web data is maintaining consistency of the cached copies. Since such documents may become stale when the data on the backend databases change, caching such documents needs stronger consistency mechanisms than the commonly used Time-to-Live schemes. ESI supports stronger consistency through explicit invalidation of cached fragments. ESI's fragment invalidation includes 2 messages: (1) Invalidation request, and

(2) Invalidation response. A server instructs the caches to invalidate one or more fragments through the invalidation response. A cache receiving an invalidation request sends the invalidation response to the server informing it of the result of the invalidation.

While fragment assembly in proxy or CDN caches has been shown to be effective in reducing the loads on the backbone networks and origin servers, it does not reduce the load of the network link connecting the end clients to the cache. However, for clients which are connected through dial-ups, the dial-up links from the clients to the reverse proxy caches located at the ISPs (or the so called last-mile) often become bottlenecks, and the latency involved in transferring documents over the dial-up links forms a significant fraction of the total latency experienced by dial-up clients. Therefore, it is important to reduce the load on the last-mile links, especially for dial-up clients. Rabinovich et al. [32] address this problem by taking fragment assembly one step further. They propose the *Client Side Includes* scheme, wherein the composition of web pages from fragments is done at the client itself, rather than within a network cache. The CSI mechanism enables the browsers to assemble the web pages from the individual fragments.

The paper also describes a JavaScript-based implementation of the CSI mechanism. In their implementation, the origin server returns a wrapper on receiving a request for a web page from a CSI-enabled client. This wrapper invokes a JavaScript-based page assembler at the client's browser. This assembler fetches individual fragments from the origin server and composes the web page, which is then displayed to the user. The clients can cache the wrapper-code of a web page in order to avoid the overhead of fetching it from the server on each access to the web page.

Another important question that needs to be addressed is how dynamic web pages can be fragmented so that servers and caches provide optimal performance. One obvious solution for this problem is to require that the web pages be fragmented by either the web page designer or the web site administrator. Manual fragmentation of dynamic web pages in this fashion is both labor intensive and error prone. Further, manual markup of fragments in web pages does not scale, and it becomes unmanageable for edge caches serving web data from multiple content providers. Ramaswamy et al. [34] present a scheme to automatically detect fragments in web pages. This scheme automatically detects and flags fragments at a given web site which exhibit potential benefits as potential cache units. Their approach depends upon a careful analysis of the dynamic web pages with respect to their information sharing behavior, personalization characteristics and change patterns.

Automatic Fragmentation of Dynamic Web Pages

Automatically fragmenting dynamic web pages presents a unique challenge. The conceptual definition of a fragment says that it is a part of a web page having distinct theme or functionality. While a human with prior knowledge of the web page's domain can easily and unambiguously identify fragments with different themes, it is not straightforward to build a system that can do the same. Not only should these systems be able to identify the themes associated with the fragments, but also they should efficiently detect fragments in web sites with thousands of web pages.

The proposed scheme detects fragments that form cost-effective cache units [34]. These fragments are referred to as *candidate fragments*. A candidate fragment is recursively defined as follows:

- Each Web page of a web site is a candidate fragment.
- A part of a candidate fragment is itself a candidate fragment if any one of the two conditions is satisfied:
  - The part is shared among "M" already existing candidate fragments, where $M > 1$.
  - The part has different personalization and lifetime characteristics than those of its encompassing (parent or ancestor) candidate fragment.

This paper distinguishes between two types of fragments. The fragments that satisfy the first condition are called the *shared fragments* and those satisfying the second condition are referred to as the *Lifetime-Personalization based fragments* or the L-P fragments. The two kinds of fragments benefit the caching application in two different ways. Incorporating shared fragments into web pages avoids unnecessary duplication of information at the caches, thereby reducing disk space usage at caches. The L-P fragments, on the other hand, improve cacheable content and reduce the amount of data that gets invalidated.

The scheme has two algorithms: one to detect shared fragments and another to detect L-P fragments. The shared fragment detection algorithm works on a collection of different dynamic pages generated from the same web site and detects fragments that are approximately shared among multiple web pages. In contrast the L-P fragment detection algorithm detects fragments by comparing different versions of the same web page.

As both fragment detection algorithms compare several web pages, they need a data structure that permits efficient comparison of web pages. The automatic fragmentation scheme uses a web page model for this purpose which is called the *Augmented Fragment Tree* model, or the AF tree model. An AF tree model is a hierarchical model for web documents similar to the document object model

(DOM) [1], with three distinct characteristics: First, it is a compact DOM tree with all the text-formatting tags (e.g., <Big>, <Bold>, <I>) removed. Second, the content of each node is fingerprinted with Shingles encoding [8]. Third, each node is augmented with additional information for efficient comparison of different documents and different fragments of documents.

Shingles are fingerprints of strings or text documents with the property that when the string changes by a small amount its shingles also change by a small amount. The similarity between two text documents can be estimated by computing the overlap between their shingles. Shared and L-P fragment detection algorithms use shingles to estimate similarity of web pages.

Shared Fragment Detection Algorithm. The shared fragment detection algorithm compares different web pages from the same web site and detects maximal fragments that are approximately shared among at least $ShareFactor$ distinct web pages, where $ShareFactor$ is a configurable parameter.

Let us consider the two web page parts shown in Figure 1.6a. The first web page part appeared in the Americas page of BBC's web site, while the second one is taken from the World News web page from the same web site. These two web page parts essentially contain the same information, which makes them prime candidates for shared fragment detection. However, detecting these fragments automatically presents several challenges. First, although the two web page parts are similar, they are not exactly the same. For example, the order of the bullet points is different, and the text appearing in one of the bullet points is slightly different. The shared fragment detection algorithm should be able to detect the entire web page part as a single fragment, or detect individual items (like the heading, the text, the bullet points) as fragments depending on the web site's preference. Second, similar html segments in two web pages might appear in completely different positions (note that the bullet points have changed their relative positions in the two web page parts). Third, if two or more web page parts are deemed to be similar, and hence detected as a shared fragment, a large fraction of their sub-parts (like the heading, the text and the bullet points in the example) would also be similar to one another. However, these sub-parts are *trivial* fragments, and hence the algorithm should avoid detecting them as fragments.

The shared fragment detection algorithm addresses these challenges. It works in two steps as shown in Figure 1.6b. In the first step, the algorithm sorts the nodes of AF trees of the different web pages based on their sizes. In the second step the algorithm detects maximally shared fragments by grouping nodes that are similar to one another based on the overlap of their shingles.

The algorithm has three tunable parameters which can be used to control the quality of the detected fragments. The first parameter, $ShareFactor$, specifies the minimum number of web pages (or more generally other fragments) that

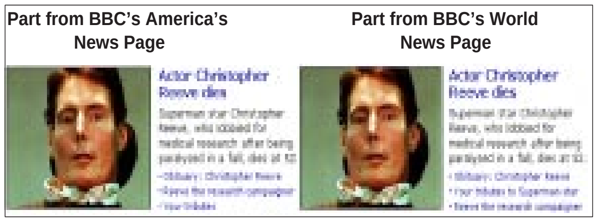

Part from BBC's America's News Page
Part from BBC's World News Page
Actor Christopher Reeve dies

Figure 1.6a. Example of Shared Fragments

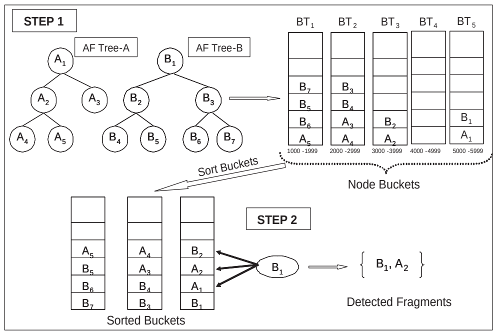
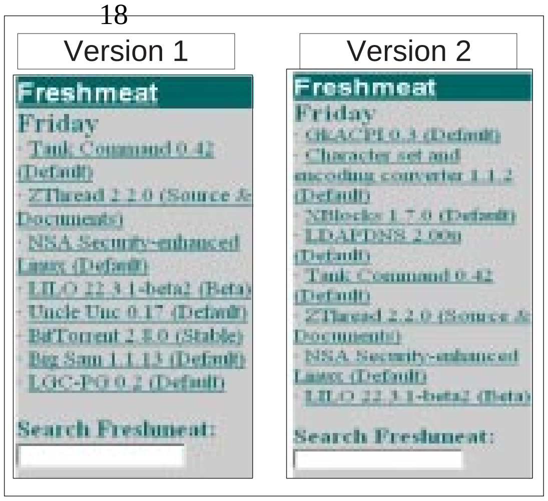


Figure 1.6b. Illustration of Shared Fragment Detection Algorithm

should share a web-page part in order for it to be detected as a fragment. The web site administrator can use this parameter to avoid detecting fragments that are shared across a very minute fraction of the web pages. The second parameter, $MinMatchFactor$, which can vary between 0.0, and 1.0, is used to specify the similarity threshold among the AF tree nodes that form a shared fragment. When this parameter is set to higher values, the algorithm looks for more perfect matches, thereby detecting larger numbers of small-sized fragments. For example, in the Figure 1.6a, if $MinMatchFactor$ is set to 0.6, the entire web page part is detected as a single fragment. If $MinMatchFactor$ is set to 0.9, the algorithm detects four fragments (a heading fragment, a fragment corresponding to the text, and two fragments corresponding to two bullet points). $MinFragSize$ is the third parameter, which specifies the minimum size of the detected fragments. One can use this parameter to avoid very small fragments being detected. If a fragment is very small, the advantages in incorporating it are limited, whereas the overheads of composing the web page from these very tiny fragments would be considerably high. A web site administrator can choose an appropriate value for these parameters based on the capability of the infrastructure, the user request patterns, and the invalidation rates.

L-P Fragment Detection. The L-P fragment detection algorithm can be used to detect fragments that have different lifetime or personalization characteristics than their encompassing fragments. The input to this algorithm is a set of different versions of the same web page. The different versions might either be time-spaced, or be obtained by sending in different cookies.

The L-P fragment detection algorithm compares different versions of the same web page and detects portions of the web page that have changed over



Freshmeat
Friday
· Tank Command 0.42 (Default)
· ZThread 2.2.0 (Source & Documents)
· NSA Security-enhanced Linux (Default)
· LILO 22.3.1-beta2 (Beta)
· Uncle Unc 0.17 (Default)
· BitTorrent 2.8.0 (Stable)
· Big Sam 1.1.13 (Default)
· LOC-PG 0.2 (Default)

Search Freshmeat:

Freshmeat
Friday
· OkACPI 0.3 (Default)
· Character set and encoding converter 1.1.2 (Default)
· XBlocks 1.7.0 (Default)
· LDAPDNS 2.00s (Default)
· Tank Command 0.42 (Default)
· ZThread 2.2.0 (Source & Documents)
· NSA Security-enhanced Linux (Default)
· LILO 22.3.1-beta2 (Beta)

Search Freshmeat:

Figure 1.7a. Examples of L-P Fragments

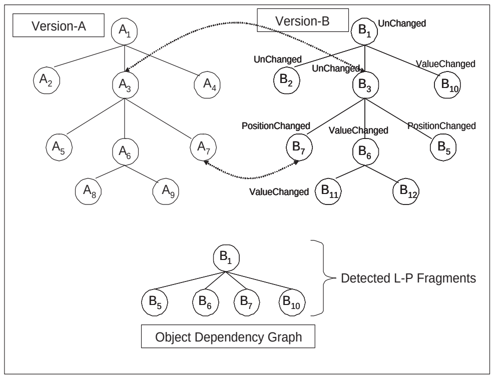


Figure 1.7b. Illustration of L-P Fragment Detection Algorithm

different versions. A web page might undergo several different types of changes: contents might be added, deleted or they might change their relative position in the web page. Figure 1.7a shows two versions of a web page part appearing in two versions of the same web page from Slashdot's web site. This figure demonstrates the various kinds of changes web pages might undergo. Therefore, the algorithm should be sensitive to all these kinds of web page changes. Further, the algorithm should also detect fragments that are most beneficial to the caching application.

The L-P fragment detection algorithm discussed in the paper installs the AF tree of the first version available (in chronological order) as the base version. The AF tree of each subsequent version is compared against this base version. For each AF tree node of a subsequent version, the algorithm checks whether the node has changed its value or position when compared with an equivalent node from the base version and marks the node's status accordingly. In the second phase the algorithm traverses the AF trees and detects the L-P fragments based on its and its children's status, such that the detected fragments are most beneficial to caching. Figure 1.7b depicts the execution of the algorithm. The detected fragments and the object dependency graph are also shown in the figure.

Similar to the shared fragment detection algorithm, the L-P fragment detection algorithm also provides parameters which can be used by the web administrators to tune the performance of the algorithm. The first parameter called the $MinFragSize$ specifies the minimum size of the detected fragments. As in shared fragment detection, this parameter can be used to preclude very small fragments. By doing so one can detect only those fragments that are cost effective cache units. $ChildChangeThreshold$ is the second tunable parameter of the algorithm. This parameter in some sense quantifies the amount of change an html-segment has to undergo before being detected as a fragment. When $ChildChangeThreshold$ is set to high values, fragments are detected at a finer granularity leading to larger numbers of small fragments. While finer granularities of fragmentation reduce the amount of data invalidations at caches, it also increases the page assembly costs at the caches. The web site administrator can decide the appropriate value for this parameter depending upon the available infrastructure and the web-site specific requirements.

The fragment detection scheme which includes these two algorithms outputs a set of fragments. These fragments are served as recommendations to the web site administrator.

5. Providing quality of service for dynamic web content

Providing good quality of service (QoS) is crucial in serving dynamic content for several reasons. First, one of the main applications of dynamic content is in e-business web sites. For an e-business web site, the competitor is only one click away making it very easy for dissatisfied customers to take their business to a competitor. Second, poor quality of service affects the image of the company as a whole. Studies have shown that users equate slow web download times with poor product quality or even fear that the security of their purchases might be compromised.

In addition, it is often important for a site to offer different levels of service, since requests vary in how important they are and users vary in how tolerant they are to delays. For example, it is in the economic interest of an e-commerce retailer to differentiate between high-volume customer and low-volume customers. Moreover, some customers may have payed for service-level agreements (SLAs) which guarantee certain levels of service. Finally, studies [7, 5] indicate that customers' patience levels decline in proportion to the time spent at the site. Patience levels are also tied to the type of request users submit. Customers tend to be more tolerant of long search times and less tolerant of long purchase times.

How users perceive the quality of service they experience when accessing web content depends on several factors:

- The latency experienced by users
  When users rate the quality of service received at a web site as poor, the most common reason is high latency. Latency, or response time, is defined as the period of time between the moment a user makes a request and the time the user receives the response in its entirety. In order for a response to be perceived as immediate, the latency needs to be on the order of 0.1 seconds. For a user to be satisfied, delays should not exceed 5 seconds.

Delays that are longer than 10 seconds are considered intolerable and lead users to assume that errors have occurred in processing their requests.

- The predictability of system performance
  Frequent customers at a site are used to a certain level of service. If the quality of service is not delivered as expected, users will not be happy. Unpredictable service compromises customers' opinions of the company.

- The overall system availability
  The worst quality of service is no service. Even short service outages can cost an e-business huge amounts in lost revenues. One common source of service outages is transient overload at the web site due to an unexpected surge of requests. Examples of this phenomenon include the overload of the Firestone web site after a tire recall, the outage of several large e-tailers during the holiday shopping season in 2000, and overload of the official Florida election site, which became overwhelmed after the presidential election of 2000.

- The freshness of data returned to the user
  Freshness of data is a quality of service aspect that is particular to dynamic content. It arises when web sites, in an attempt to serve a dynamic request more efficiently, rely on cached data from an earlier execution of the dynamic request. This approach is for example commonly used when serving dynamic requests that require access to a database back-end. While using cached data reduces work for the server, users might receive outdated data. Consider, for example, an e-commerce site that reports product availability based on cached values of stock levels. The site might realize only after accepting an order that the item is not actually in stock.

The network community has long studied the efficacy of providing quality of service guarantees and providing class-based service differentiation. However, much of this work relies on the assumption that the network is the typical bottleneck in web transfers. Serving dynamic content can require orders of magnitudes more processing power at a web site compared with serving purely static data [12]. For web sites that are dominated by dynamic content the bottleneck tends to shift from the network to the server, making it necessary to provide QoS mechanisms at the server.

In the remainder of this section, we will survey recent research on providing quality of service at a web server. We first discuss approaches for achieving system-wide QoS goals, i.e. achieving the same set of QoS goals across all requests served by a site. We then examine solutions for providing class-based QoS, where different classes of requests have different QoS goals.

Achieving system-wide QoS goals

The system load a site experiences is the main factor affecting the quality of service. The resources at a web site are limited, so providing a certain level of quality of service requires keeping the system load below some threshold. The main approach for guaranteeing a certain level of QoS is therefore load control in order to avoid *overload* conditions. Overload in this context refers not only to load that exceeds the system's capacity, but also to load that is too high to ensure the system's QoS specifications.

Load control typically consists of two steps; detecting when a system reaches overload and reacting by taking measures to reduce the load. The approaches suggested for load control vary depending on where they are implemented. Load control can be integrated into the operating system of the web server, it can be implemented at the application level, or it can be moved to the database back-end. Below we first describe different approaches for detecting overload and then discuss possible solutions to the problem.

Detecting Overload. Approaches for overload detection at the *operating system level* fall into two categories. Approaches in the first category focus on the network stack and typically monitor the occupancy of the SYN queue or the TCP listen queue [38]. While the occupancy of these queues provides only a very limited view of the overall state of the server, the back-pressure caused by over-utilization of resources in one of the higher system layers will eventually lead to a backlog in those kernel queues.

Another approach for overload detection at the operating system level is based on measuring the utilization of server resources. One of the motivations for considering server resource utilization is that some QoS algorithms can be theoretically proven to guarantee a certain level of service, assuming that the server utilization is below some threshold [4].

When using information on the utilization level of the system in load control decisions, it is important to distinguish between high utilization caused by a short, transient burst of new traffic, versus high utilization resulting from a persistent increase in traffic that calls for load control. Cherkasova et al. [18] therefore propose to use a predictive admission control strategy that is based on the weighted moving average of the utilization level observed in previous measurement periods, rather than the instantaneous utilization level.

Methods for detecting overload at the *application level* are either based on monitoring the occupancy of application internal queues, or on developing an estimate of the work involved in processing the currently active request.

The prior approach is taken by [33] and [6]. Both employ mechanisms for rapidly draining the TCP listen queue and managing the outstanding requests in an internal system of queues. The lengths of the internal queues can then indicate when to apply load control. Moving load control from the kernel queues to an application level queue helps to avoid TCP timeouts experienced by requests dropped in the kernel queues. Moving the load control to the application level also allows for the use of application-level information in the load control process.

Chen et al. [17] and Elnikety et al. [22] follow the latter approach, i.e. they approximate the current system load based on estimates of the work imposed by each request in progress. Chen et al. experiment with the CGI scripts included in the WebStone benchmark. They measure the CPU usage for each CGI script and use it as an estimate for the work associated with serving the corresponding dynamic request. Elnikety et al. consider java servlets that communicate with a database back-end and find that estimates of per-servlet service time converge relatively quickly. Hence the per-servlet estimates can indicate the load a given dynamic request introduces to the system. Both studies then approximate the system load at any given time by summing up the per-request service time estimates for the requests in progress. Load control is triggered if the estimated load in the system is close to the system capacity, which is determined in off-line experiments.

Research on integrating load control into the *database server* has mostly focused on the avoidance of lock thrashing caused by data contention. A database-integrated approach offers the possibility of utilizing more detailed information on the transactions in progress. Rather than simply basing decisions on the number of transactions in progress, the load control can utilize knowledge of the state of the individual transactions (running vs blocked waiting for a lock) and the progress the transactions have made.

Choosing the right approach for implementing overload control involves several trade-offs. Integration into the operating system allows overload control by immediately dropping new requests before occupying any system resources. On the other hand, the advantage of application-level load control is that application-level information, e.g. the expected work to process a given request, can be taken into account. For complex applications like database systems, the application level and the operating system have only limited information of the state of the system, potentially allowing only for coarse-grained load detection and control. However, the more fine-grained approach of integrating QoS mechanisms into the database system comes at the price of modifying a complex piece of software.

Reacting to Overload. After detecting an overload situation, measures must be taken to reduce the server load. One common approach is to reduce the number of requests at the server by employing admission control, i.e. selectively rejecting incoming requests. An alternative approach is to reduce the work required for each request, rather than reducing the number of requests, by serving lower quality content that requires less resources. The premise for using content adaptation rather than admission control is that clients prefer receiving lower quality content over being denied service completely.

- **Dealing with overload through content adaptation**
  Content adaptation to control overload has first been suggested by Bhatti et al [3]. Some of the proposed mechanisms apply mostly to static content, e.g. the suggestion to replace large, high resolution images by small, low resolution images to reduce the required bandwidth. Their approaches can also help reduce high load that is due to dynamic content. For example, the authors propose reducing the number of local links in each page, e.g. by limiting the web site's content tree to a specific depth. A smaller number of links in a page affects user behavior in a way that tends to decrease the load on the server.

  The work in [15] shows how to apply the concept of service degradation to dynamic content that is generated by accessing a back-end information systems, such as a database server. They propose to generate a less resource intensive, lower quality version of this type of content, by compromising the freshness of the served data by using cached or replicated versions of the original content.

  Chen et al. implement this notion of service degradation by complementing the high-end database back-end server at a web site with a set of low-end servers that maintain replicated versions of the original data with varying update rates. If the load at the high-end server gets too high, traffic can be offloaded to the low-end servers at the price of serving more outdated data.

  Li et al. [26] propose a similar approach for database content that is replicated in data centers across a wide area network. They characterize the dependency between request response times and the frequency of invalidating cached content (and hence the freshness of the cached content) and exploit this relationship by dynamically adjusting the caching policy based on the observed response times.

  A famous example for the application of service degradation in practice includes the way CNN handled the traffic surge at its site on Sept 11, 2001 [25]. CNN's homepage, which usually features a complex page design and extensive use of dynamic content, was reduced to static content with one page containing 1247 bytes of text, the logo, and one image.

- **Dealing with overload through admission control**:
  The simplest form of admission control would be to reject all incoming requests, either in the network stack of the operating system or at the application level, until the load drops below some threshold. There are

at least two problems caused by the indiscriminate dropping of requests in this naive approach.

1 The decision to drop a request does not take into account whether the request is from a new user or part of a session that has been lasting for a while. As a result, long sessions are much more likely to experience rejected requests at some point during their lifetime than short sessions. However, it is often the long sessions at an e-commerce server that finally lead to a purchase.

2 The decision to drop a request does not take the resource requirements of the request into account. To effectively shed load through admission control, one ideally wants to drop incoming requests with high resource requirements. On the other hand, despite high load, the server might want to accept a request if this request has very small resource requirements or requires only little service at the bottleneck resource.

The work of Cherkasova et al. [19] and Chen et al. [16] addresses the first problem by taking session characteristics into account when making admission control decisions. Simply put, if the load at the server exceeds a specified threshold, only requests that are part of an active session are accepted, while requests starting new sessions are rejected. In practice, this approach can be implemented by using cookies to distinguish whether an incoming request starts a new session or is part of a session in progress.

[22] and [17] base admission control decisions on estimates of the service requirements of incoming requests and estimates of the server capacity (determined as described above). The system load at any given time is computed as the sum of the service requirements of all requests in progress. Elnikety et al. admit a new request to the system only if adding its service requirement to the current load does not increase the load beyond the server capacity. Requests that would increase the server load beyond its capacity are stored in a backup queue. Requests are only dropped after the backup queue fills up.

Orthogonal to the above approaches, Welsh et al. [39] propose a whole new design paradigm for architecting internet services with better load control that they call SEDA (Staged-event-driven-architecture). In SEDA, Internet services are decomposed into a set of event-driven stages connected with request queues. Each stage can control the rate at which to admit requests from its incoming request queue and decide which requests to drop in the case of excessive load. This approach allows fine-grained admission control. Moreover, combating overload can be focused on those requests that actually lead to a bottleneck (since requests that never enter an overloaded stage are not affected by the admission control).

As mentioned earlier, work for load control in the database community is mostly concerned with avoiding thrashing due to data contention. Database internal methods reduce data contention not only by employing admission control, but also by canceling transactions that are already in progress. The conditions for triggering admission control and the canceling of transactions depends on the state of the transactions in progress. One option is to trigger admission control and transaction cancellation once the ratio of the number of locks held by all transactions to the number of locks held by active transactions exceeds a critical threshold [29]. Another possibility is to apply admission control and cancel an existing transaction if more than half of all transactions are blocked waiting for a lock after already having acquired a large fraction of the locks they require for their execution [11]. In both cases, the transaction to be canceled is one that is blocked waiting for a lock and at the same time is blocking other transactions.

Differentiated quality of service

There are two different types of differentiated quality of service.

- Best effort performance differentiation
  Different classes with different priorities, where higher priority classes should receive better service than lower priority classes.
- Absolute guarantees
  Each class has a concrete QoS goal that it needs to meet, e.g. a latency target specified in number of seconds.

Best effort performance differentiation. Methods for providing best effort performance differentiation typically follow one of the following three approaches:

1 providing different quality of content depending on the priority of a request.

2 changing the order in which requests are processed based on request priorities.

3 adapting the rate at which a request is served according to the request priority.

We first discuss how to implement these approaches for non-database driven requests, and we then turn to the question of how to achieve performance differentiation at a database back-end server.

For non-database driven requests the first two approaches can be implemented as extensions of methods discussed previously for achieving system-wide QoS goals. The first approach can be implemented by applying the methods for content degradation described above, where the degree of content degradation is chosen according to request priorities. The second approach can be implemented as a straightforward extension of any of the mechanisms presented earlier that involve queues. More precisely, the processing order of requests can be changed based on request priorities, by either prioritizing the kernel SYN and listen queue [38], or by prioritizing application internal queues such as those used in the work by [6] and by [33]. The former approach is limited to the case where class priorities can be determined based on the client IP address, since at this point no application level information is available.

Mechanisms for changing the rate at which requests are processed (approach 3 in the above list) include limiting the number of server processes available to low priority requests and adjusting the operating system priorities of server processes [21]. The applicability of both of these mechanisms is limited in that they assume a process-based server architecture.

When it comes to database-driven requests, most commercial databases ship with tools that allow the database administrator to assign priorities to transactions in order to affect the rate at which transactions is processed. The implementation of those tools usually relies on CPU scheduling [35, 24] and is therefore most effective when applied to CPU bound workloads.

For database driven workloads whose performance is limited by data contention, effective service differentiation requires prioritization applied at the lock queues. McWherter et al. [28, 27] evaluate different lock scheduling strategies and find that simple reordering of lock queues according to priorities provides a relatively good level of service differentiation. However, to allow for optimal performance of high priority transactions, preemptive lock scheduling policies are necessary, i.e. scheduling policies that allow high priority transactions to abort and restart low priority transactions in case of a lock conflict. McWherter et al. show that naive preemptive policies impose harsh performance penalties onto the low priority transactions due to the large amounts of wasted work introduced by transaction aborts and restarts. They go on to propose a new preemptive lock scheduling policy that is able to balance the cost (in terms of wasted work due to rollbacks) and the benefit of preemptive lock scheduling.

Achieving absolute guarantees. One approach for achieving absolute delay guarantees is by adapting the approaches for best-effort differentiation (described above) through feedback control. For example, admission control could be used in combination with one of the approaches that maintains internal request queues to adjust the drop rate of low priority requests if the response times of high priority requests are higher or lower than their target. This approach has been shown to work sufficiently well in the presence of only two priority classes, where one is best-effort and one has a delay bound [33]. However, multi-class latency targets call for more complex methods.

[17] propose the use of priority scheduling in combination with admission control guided by queuing theory to ensure multi-class latency targets. More precisely, requests are scheduled from an application internal queue in strict priority order. The priority of a request is determined by the latency target of its class: requests from classes with low latency targets always have priority over requests from classes with higher latency targets. This scheduling policy implies that the share of the system capacity received by a class equals the total system capacity minus the capacity used up by higher priority classes. The per-class share of the system capacity can be determined based on estimates for request service requirements, the per-class arrival rate, and the overall system capacity. Chen et al. then use known queuing formulas to compute the maximum arrival rate a class can sustain while staying in its delay target and apply admission control to ensure that a class stays within this maximum rate.

The above work focuses only on requests that are not database driven. There is relatively little work in the area of database systems for supporting per class QoS targets. Most work that is concerned with providing performance guarantees is in the large domain of real-time database systems (RTDBMS). The goal in RTDBMS is not improvement of mean execution times for high priority classes of transactions, but rather meeting (usually hard) deadlines associated with each transaction. In achieving this goal RTDBMS often rely on their specialized architecture with features such as optimistic concurrency control mechanisms, which are not available in the general purpose database systems used as web server back-ends.

The existing work in the area of general purpose databases systems for providing multi-class response time goals relies on buffer management strategies [9, 10, 36]. However, as of now these, there has been little work evaluating these approaches for web-driven database workloads.

References

[1] Document Object Model - W3C Recommendation. http://www.w3.org/DOM.

[2] Edge Side Includes - Standard Specification. http://www.esi.org.

[3] Tarek F. Abdelzaher and Nina Bhatti. Web content adaptation to improve server overload behavior. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 31(11–16):1563–1577, 1999.

[4] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[5] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. In *Proceedings of the 9th International World Wide Web Conference*, 2000.

[6] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, 1999.

[7] Anna Bouch, Allan Kuchinski, and Nina Bhatti. Quality is in the eye of the beholder: meeting users requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2000.

[8] Andrei Broder. On resemblance and Containment of Documents. In *Proceedings of SEQUENCES-97*, 1997.

[9] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing memory to meet multiclass workload response time goals. In *Proceedings of the Very Large Database Conference*, pages 328–341, 1993.

[10] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 353–346, 1996.

[11] Michael J. Carey, Sanjey Krishnamurthy, and Miron Livny. Load control for locking: The 'half-and-half' approach. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1990.

[12] Jim Challenger, Paul Dantzig, Arun Iyengar, Mark Squillante, and Li Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, 12(2), 2004.

[13] Jim Challenger, Arun Iyengar, and Paul Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM'99*, March 1999.

[14] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic Web content. In *Proceedings of IEEE INFOCOM*, March 2000.

[15] Huamin Chen and Arun Iyengar. A tiered system for serving differentiated content. *World Wide Web*, 6(4), 2003.

[16] Huamin Chen and Prasant Mohapatra. Overload control in qos-aware web servers. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 42(1):119–133, 2003.

[17] Xiangping Chen, Prasant Mohapatra, and Huamin Chen. An admission control scheme for predictable server response time for web accesses. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 545–554, 2001.

[18] Ludmila Cherkasova and Peter Phaal. Predictive admission control strategy for overloaded commercial web server. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 500, 2000.

[19] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[20] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-Based Accelaration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In *Proceedings of SIGMOD-2002*, June 2002.

[21] Lars Eggert and John S. Heidemann. Application-level differentiated services for web servers. *World Wide Web*, 2(3):133–142, 1999.

[22] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web*, pages 276–286, 2004.

[23] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.

[24] IBM DB2. Technical support knowledge base; Chapter 28: Using the governor. `http://www-3.ibm.com/cgi-bin/db2www/data/db2/udb/winos2unix/support/document.d2w/report?fn=db2v7d0frm3toc.htm`.

[25] William LeFebvre. CNN.com: Facing a world crisis. Invited talk at the USENIX Technical Conference, June 2002.

[26] Wen-Syan Li, Oliver Po, Wang-Pin Hsiung, K. Selcuk Candan, and Divyakant Agrawal. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proceedings of the twelfth international conference on World Wide Web*, pages 587–598. ACM Press, 2003.

[27] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of the 21th IEEE Conference on Data Engineering (ICDE'2005)*, 2005.

[28] David T. McWherter, Bianca Schroeder, Annastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of the 20th IEEE Conference on Data Engineering (ICDE'2004)*, 2004.

[29] Axel Moenkeberg and Gerhard Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In *Proceedings of the Very Large Database Conference*, pages 432–443, 1992.

[30] Prasant Mohapatra and Huamin Chen. A Framework for Managing QoS and Improving Performance of Dynamic Web Content. In *Proceedings of GLOBECOM-2001*, November 2001.

[31] Anoop Ninan, Purushottam Kulkarni, Prashant Shenoy, Krithi Ramamritham, and Renu Tewari. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *Proceedings of the Eleventh International World Wide Web Conference (WWW2002)*, May 2002.

[32] Michael Rabinovich, Zhen Xiao, Fred Douglis, and Charles R. Kalman. Moving Edge-Side Includes to the Real Edge - the Clients. In *Proceedings of Usenix Symposium on Internet Technologies and Systems*, March 2003.

[33] P. Bhoj S Ramanathan and S. Singhal. Web2K: Bringing qos to web servers. Technical Report HPL-2000-61, HP Laboratories, 2000.

[34] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic Detection of Fragments in Dynamically Generated Web Pages. In *Proceedings of* 13^{th} *World Wide Web Conference*, May 2004.

[35] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The Oracle Database Resource Manager: Scheduling CPU resources at the application level. 2001.

[36] Markus Sinnwell and Arnd C. Koenig. Managing distributed memory to meet multiclass workload response time goals. In *Proceedings of the 15th IEEE Conference on Data Engineering (ICDE'99)*, 1997.

[37] Junehua Song, Arun Iyengar, Eric Levy, and Daniel Dias. Architecture of a Web server accelerator. *Computer Networks*, 38(1), 2002.

[38] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[39] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 2003 USENIX Symposium on Internet Technologies and Systems*, 2003.

[40] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):563–576, 1999.