WPI-CS-TR-01-03 March 2001

Change and Relationship-Driven Content Caching, Distribution and Assembly[1]

by

Mikhail Mikhailov
Craig E. Wills

[1]This work is partially supported by the National Science Foundation Grant CCR-9988250.

Abstract

Web caching and Content Distribution Networks (CDNs) seek to reduce retrieval latency, lower bandwidth usage and reduce load on the origin servers by moving the offered content closer to the end users. Historically, the content replicated to the edges of the network in this way has been mostly static, such as images. As more dynamic and personalized content is being offered on the Web, there is a growing need for mechanisms capable of caching and replicating content that changes frequently and is user-specific. In addition, existing caches heuristically estimate object freshness lifetimes, which results in unnecessary validation traffic between caches and origin servers, and does not prevent caches from serving stale objects to their clients.

In this work we describe a mechanism that makes caching of Web objects more deterministic, reduces validation traffic, increases the amount of content that may be cached, and reduces the amount of dynamic content that must be retrieved from the origin server. Our mechanism uses a number of techniques in concert. First, objects that constitute a page are classified based on their change characteristics. Next, servers analyze relationships between objects in conjunction with object change characteristics and compile them into *Content Control Commands*. Caches and servers then use these commands to manage objects. Finally, caching and CDN servers construct frequently changing, personalized, or any other pages from individual components using a powerful *Content Assembly* technique.

Keywords: Web Caching, Content Distribution, Change Characteristics, Object Relationships, Object Composition, Content Assembly

1 Introduction

The sheer size and explosive growth of the World Wide Web demands powerful techniques to scale the Web and improve its performance. Since the early days of the Web organizations successfully deployed caching proxy servers to lower the usage of bandwidth on their Internet connections and decrease the response time for their internal users. Academic and industrial efforts to advance the state-of-the-art in Web performance led to the introduction of caching hierarchies, interception proxies, surrogate servers and more recently to the deployment of CDNs.

While caching and content delivery infrastructures are integral parts of the Web, they lack a mechanism for caching frequently changing and personalized content. Servers compute that content upon request and cannot assign it a useful expiration or last modification time—the two parameters currently used by caches in determining content freshness lifetime. Caching and CDN servers store objects that change infrequently, many of which are images. However, even these cacheable objects are managed heuristically because there is no efficient cache consistency mechanism.

In our previous work [16] we proposed an alternative approach to caching of Web objects that addresses these problems. The idea is to classify objects based on their types and change characteristics, compose Web pages from such objects, and expose the page structure to caches. Our approach also combines the relationships between objects with object change characteristics to better manage individual objects. In our previous work we evaluated potential gains of our approach and reported encouraging results. In this paper we describe elements of the design of a system implementing our approach.

The paper is organized as follows. We introduce our classification of object change characteristics and show how servers better manage their objects by controlling caches with object-specific *Content Control Commands* (*CCCs*). Then we describe a powerful *Content Assembly* (*CA*) technique that enables caching and CDN servers to construct frequently changing, personalized, or other pages from individual components. We illustrate how the addition of Content Assembly increases the amount of content that can be cached and also show specific examples of how CA-enabled caches can personalize pages. We report on the status of a prototype system implementing our approach, and also compare our ideas to those proposed by others, concluding the paper with a summary and a discussion of directions for future work.

2 Current Practice

Let us consider an example of a Web page, shown in Figure 1. This example is motivated by home pages of popular news portals. The container object *CO* is changing frequently—every few minutes—because content designers update the top story, add and remove links leading to the major news articles. Irrespective of the manual updates, every request for *CO* results in a different response because the origin server dynamically generates *CO*, changing which ad banner image to display and where on the page to place it. Embedded objects *EO1*—*EO3* change only occasionally. Such changes are applied by a human, occur at unpredictable points in time, and are saved under the same name as the previous version of the object.

Objects *EO4* and *EO5* never change. If changes are required, they are saved under a different name, effectively resulting in a new object being created.

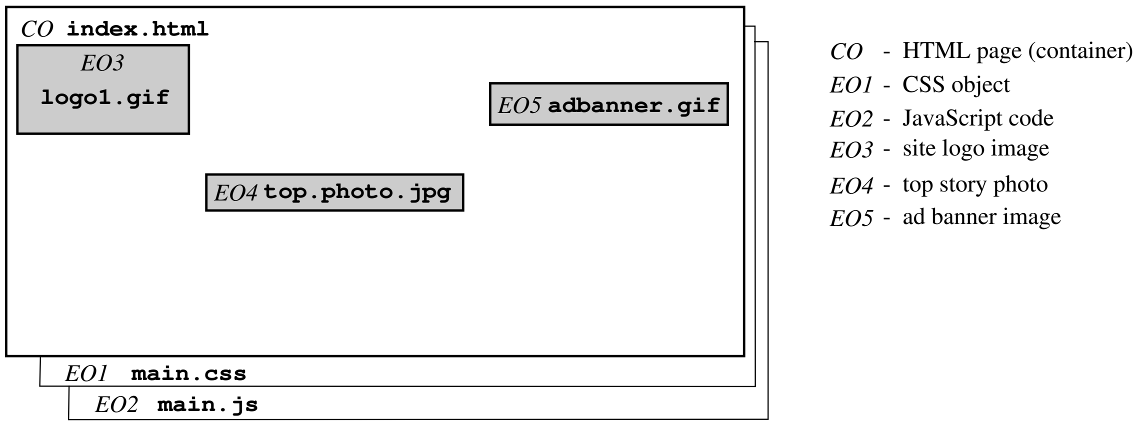
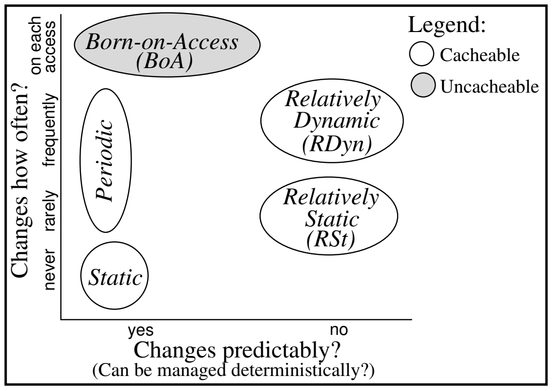


Figure 1: Home Page of a Popular News Site

Let us now consider how these objects are currently managed within a proxy-style caching architecture. Caches cannot store the container *CO* because it changes frequently and either carries an explicit indication that it is uncacheable or has no cache control meta information associated with it. The five embedded objects *EO1*—*EO5* change rarely or never and may be cached. Currently, the more cache-friendly origin servers assign cacheable objects an expiration or a last modification time, via `Expires` and `Last-Modified` HTTP response headers respectively. When an explicit expiration time is not available, caches, such as Squid [14], consider a configurable fraction of object's age to be a reasonable estimate for the freshness lifetime for that object. Age of an object is the difference between current time and that object's last modification time. The heuristic used here, referred to as Alex protocol [4], suggests that the younger files are likely to change sooner than the older files. It is impossible for caches to deterministically know when cached objects *EO1*—*EO5* become stale because servers can not accurately predict object expiration times and heuristic TTLs are imprecise by definition. Also, servers can inadvertantly provide misleading expiration and last modification times [15]. As a result, caches may serve stale objects to their clients. Caches also generate unnecessary traffic and place additional load on the origin servers when they validate objects that have expired in the cache but are unchanged at the origin server. Studies show that such validation requests represent 15-18% [9], 30% [12] and 37% [2] of all requests served by origin servers.

3 Object Change Characteristics

As illustrated in Figure 1, Web objects not only have different content types, but also exhibit distinct change characteristics. Our classification of object change characteristics is given in Figure 2. The three categories on the left—Static, Periodic and BoA—represent *predictable* changes. Objects in these predictable categories can be managed *deterministically*: cached and never validated, cached for a predetermined period, and always retrieved from the origin server respectively. The two categories on the right—RDyn and RSt—cover objects that can be cached but change unpredictably and, therefore, cannot be managed deterministically. Caches consider RSt objects, which change rarely, fresh until a heuristically assigned TTL expires, and validate RDyn objects, which change frequently, on each request.



Class	Example
Periodic	periodically updated weather map
BoA	container *CO* in Figure 1
Static	objects *EO4* and *EO5* in Figure 1
RSt	objects *EO1*—*EO3* in Figure 1
RDyn	list of latest news stories

Figure 2: Classification of Object Change Characteristics

Ideally, parts of BoA content that do not change on every access should be factored out of BoA into any other category so that they can be cached, whether deterministically or heuristically. Non-BoA content that changes predictably should be classified as such so that it can be cached deterministically. Also, portions of RDyn content that change rarely should be factored out and classified as RSt to avoid validations on every access. Finally, to minimize the reliance on heuristics and reduce validation traffic while managing objects in the unpredictable categories, RDyn and RSt, we propose to exploit the relationships that these objects have with objects in the predictable categories.

We believe unpredictably changing content can be managed more deterministically and with fewer transferred messages and bytes than currently if servers use object change characteristics in deciding which management strategies to use. A key question is whether it is possible and feasible to classify a large number of objects at a Web site into these categories. Our viewpoint is that many objects are already automatically generated based on measurable events or at regular intervals. It is a trivial addition to these automated tasks to mark the resulting objects with appropriate change characteristics. In addition, the type of an object may define its change characteristic. For example, a manually created image may be marked as static. Another observation, that we believe makes this approach possible, is that only the most popular objects require classification, with others being classified as RSt by default.

4 Object Management

In this section we use the example in Figure 1 to illustrate our approach to the problem of managing Web objects more deterministically and efficiently. First, objects at a site are classified based on their change characteristics. Second, the server compiles the relationships between objects constituting a page in conjunction with change characteristics of these objects into *Content Control Commands* (*CCCs*), which it then associates with the appropriate objects.

In our example, servers notify caches that objects *EO4* and *EO5* are Static. Caches store Static objects for as long as they deem necessary, without ever generating validation requests. Objects *EO1—EO3* belong to the RSt category and would be managed heuristically if they were standalone. In our case, however, these RSt objects are embedded in the container that is retrieved on every access. Servers use the deterministic retrieval of the container *CO* to invalidate the RSt objects embedded in it so that caches can store these RSt objects and treat them as fresh until the server sends an invalidation. Servers treat objects *EO1—EO3* as a volume [9, 7] and assign each of them a CCC, an example of which is shown in Figure 3. The CCC associated with the container *CO* is shown in Figure 4, it instructs caches that they may keep only meta information about the *CO*. If servers did not use the container to manage RSt objects, they would instruct caches not to cache it.

```
HTTP/1.1 200 OK
Server: Apache/1.3.12 (Unix)
ETag: 4321
CCC: MAY cache until INV
Date: Sun, 04 Mar 2001 10:01:00 GMT
```

Figure 3: Server Response with CCC for *EO1—EO3*

```
HTTP/1.1 200 OK
Server: Apache/1.3.12 (Unix)
ETag: 1234
CCC: MAY cache METAINFO
Date: Sun, 04 Mar 2001 10:00:00 GMT
```

Figure 4: Server Response with CCC for container *CO*

The server reply to a request for the container *CO* looks slightly different if the server sends an invalidation. Suppose a client requested and cached the objects shown in Figure 1. Object *EO* was later updated, and the same client is requesting the container *CO* again, indicating to the server that it had previously seen version `1234` of the *CO*. The server determines which objects the container *CO* embedded at the time of the first request and invalidates those RSt objects that have been updated—*EO2*. For efficiency, the server piggybacks invalidation information onto its response to the client [9]:

```
HTTP/1.1 200 OK
Server: Apache/1.3.12 (Unix)
ETag: 5678
CCC: MAY cache METAINFO; INV main.js
Date: Mon, 05 Mar 2001 10:00:00 GMT
```

The client invalidates object *EO2* in its cache, retrieves a new copy of *EO2*, and reuses cached copies of those RSt objects that the server did not explicitly invalidate: *EO1* and *EO3*. Since RSt objects change infrequently, we expect few invalidations. If the client does not indicate to the server that it had previously seen the *CO* object, the server can either provide no invalidation information at all, as is done currently, or inform the client about the current versions of the RSt objects embedded in the *CO*.

Our approach to object management allows us to deterministically manage all objects in Figure 1, even the RSt ones. Except for the container that is changing on every access, caches store all objects and re-retrieve them only when they truly change. In our example, caches generate no validation requests—a significant improvement over current practice.

Evaluation of our approach on proxy logs in our earlier work, where we had to make simplifying assumptions regarding object relationships, showed that 75-89% of all validation requests could be eliminated if servers sent invalidations for objects that have not changed while serving requests for related objects that have changed [16].

5 Content Assembly

Page construction is a two-step process. The first step is content generation. Designers often use multiple heterogeneous data sources with different change characteristics to generate timely and personalized content. Web site building tools, such as PHP [13] and Mason [10], assist designers in placing content that changes differently from the rest or content shared between multiple pages into separate files or *components*. These tools even provide server-side caching mechanisms that cache the results of component execution. The second step is assembly of the final page from a set of components. Currently origin servers perform both steps *before* serving pages to clients. Clients receive monolithic pages that inherit the most dominant change characteristic of all of their original constituent parts. When one underlying data source changes the entire page changes and must be retrieved anew. Furthermore, currently *only* origin servers are equipped to perform both steps of page construction.

We propose to enable other entities, such as L7 switches, caching and content distribution servers, browsers, in addition to origin servers, to carry out page assembly, while keeping content generation an origin server's prerogative. Most of the components can be cached, and page assembly can be done as close to the end users as possible, with only frequently changing components obtained from the origin server as needed.

Content designers expose the structure of their pages to authorized clients by placing environment-specific placeholders within objects where components must be inserted. In the context of HTML, a placeholder could be a new tag, such as `INSERT`. We attempted to utilize the existing HTML tags, `ilayer` and `iframe`, but discovered that they both treat the inserted content as structurally separate from the container. Currently browsers do not understand the `INSERT` tag, and may not be allowed to receive raw components in the future. We envision that trusted intermediate devices, such as certain caching proxy servers and CDN servers, are customized to perform *Content Assembly* (*CA*) and deliver fully assembled pages to browsers.

Content Assembly is the process of parsing components and replacing all occurrences of `INSERT` tags with the contents of the components that they include. CA also involves combining cache control information associated with each component to produce cache control information for the assembled object. When serving requests from CA-unaware clients, origin servers can also assemble pages from components or generate content using existing mechanisms. In general, any CA-enabled server may forward unassembled components only to clients that are known to have the CA capability and are authorized to receive raw components from a particular content provider. While the issues of security and authorization are important, we have not focused on them thus far.

Components can recursively embed other components via `INSERT` tags, and `INSERT` tags can recursively embed other `INSERT` tags. In practice there should be a limit to the number of recursion levels. CA servers may choose to perform either *full* or *partial* assembly. This decision could depend on a number of conditions. An overloaded caching server could skip the assembly or assemble only those components that are present in the cache and are fresh. In a CA-enabled caching hierarchy, only servers at the client-side edge of the network may be allowed to assemble content. A CA server may cache the result of the assembly and reuse it on the subsequent requests, thus amortizing the cost of the assembly over multiple requests.

6 Object Management with Components

Let us modify the example in Figure 1 so that the container is not a monolithic object, but a composition of multiple components (objects). The modified page is shown in Figure 5. We placed parts of the original BoA container *CO* that change frequently or on every access in separate components: *CMP1* and *CMP2* are BoA, and are the only objects in our modified page that must be retrieved from the server on every access; *CMP4* and *CMP5* are RDyn, they can be cached, but must be validated on every access. We relocated cacheable and deterministically manageable content—a daily opinion poll on a controversial question—into Periodic component *CMP6*. We also moved RSt content that is shared between many pages at the site into components *CMP3* and *CMP7*. This content, as well as the new RSt container *CO'*, can now be cached.

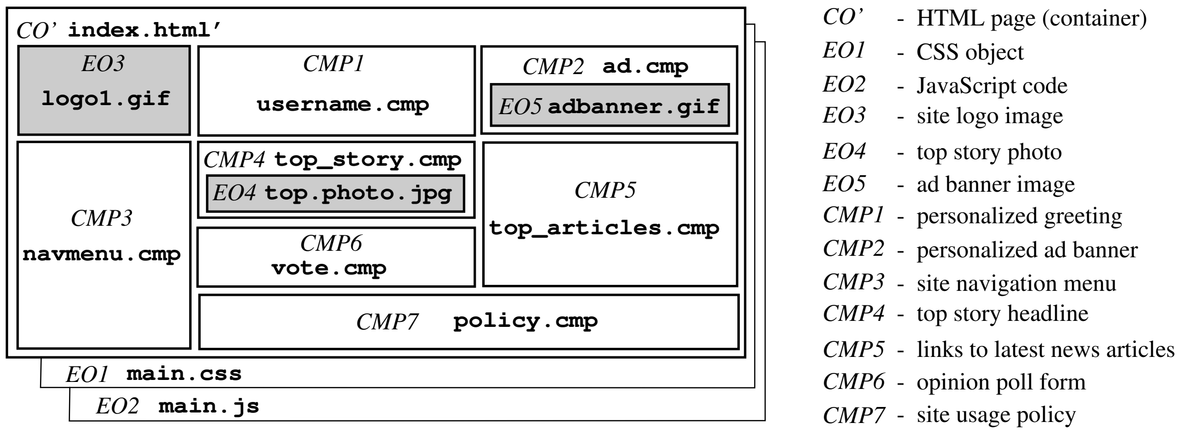


Figure 5: Objects and Components Composing Home Page of a Popular News Site

The transition of the container object from BoA to RSt fundamentally affects how we manage our set of objects. Since there is no need to retrieve *CO'* on every access, caches can not expect the origin server to invalidate RSt objects contained or embedded in the container. The server selects one of the two objects that are retrieved on every access to assist caches in managing the container *CO'* and other non-deterministically changing objects contained or embedded in *CO'*. The server associates the following CCC with *CO'*, instructing caches that they may cache the container but should always satisfy the provided *precondition*—retrieval of *CMP2* in this case—before reusing the cached copy of the container:

```
HTTP/1.1 200 OK
Server: Apache/1.3.12 (Unix)
ETag: 1234
CCC: MAY cache, PRECOND=GET ad.cmp
Date: Sun, 04 Mar 2001 10:00:00 GMT
```

The server also associates a CCC with the *CMP2* component. Upon receiving a request for *CMP2*, referred from the RSt container *CO'*, the server attempts to invalidate the container and the RSt objects embedded in it. The server may also *validate* RDyn objects embedded in the *CO'*. This last rule is useful because RDyn objects are expected to change frequently, but not necessarily on every access, and validating the ones that have not changed avoids an unnecessary freshness inquiry that the cache must send to the server.

Extending our basic object management approach, discussed in Section 4, with the page composition technique not only preserves all the advantages of the basic approach over the current practice—deterministic management of objects composing a page and elimination of validations generated by caches—but also allows more page content to be cached. Evaluation of our extended approach on home pages of popular Web sites showed that the page composition technique increases the amount of content that can be cached by 50% [16]. Furthermore, since some of our RSt components, such as the site navigation menu *CMP3*, are likely to be used on many pages at the site, benefits are derived from retrieving and caching these components only once.

Additional benefits of the page composition technique, however, come at a cost. The componentized page in Figure 5 contains two, not one, BoA objects, and also two RDyn objects that change frequently and must be retrieved anew. How a site's content is decomposed into components in practice is up to the content designers. For example, *CMP1* and *CMP2* could be merged into a single component. If BoA components are interspersed throughout an RSt container so that they can not be merged content designers could advertize a single *aggregate* BoA object that is used to generate all the required BoA objects at once. Once a client receives such an aggregate object, or a *bundle* [17], it recovers all individual BoA objects encapsulated in it. Also, in the next section we show how to transform personalized and customized content from BoA into a cacheable category.

7 Assembling Customized Content

Personalized objects are currently treated as BoA because origin servers generate them based on the information in the client request, such as a cookie identifying a particular user, parameters that follow a '?' in the requested URL, or an HTTP request header. Personalized objects form the *Input Dependent* (*InpDep*) category of change characteristics, which is separate from the BoA category. Objects in the InpDep category also belong to one of the categories depicted in Figure 2. For example, an object can be both RSt and InpDep. That object should be treated as an RSt object as long as requests supply the same input parameter. This property of InpDep objects allows us to remove the dependency on input parameters at the origin server, cache the resulting non-InpDep object, and later on have the cache reintroduce the dependency on input during the content assembly process. The three examples that follow illustrate this approach.

7.1 Personalization

Consider the personalized greeting component *CMP1* in Figure 5. Instead of having the server generate this component on every request, content designers could remove the dependency on input and make the new *CMP1'* component of the category RSt and cacheable by changing `greeting.cmp` to:

```
Welcome, <INSERT SRC=userprofile/<INSERT ISRC=userid DEFAULT=new-user-id>#Name>!
```

As a CA-enabled cache assembles a page with the *CMP1'* component it replaces the inner `INSERT` construct with the value of the `userid` object that the current request supplies in the cookie. For example, if a request contains the "`Cookie: userid=ID1`" HTTP request header, the assembler replaces the inner `INSERT` tag with `ID1`. Trusted caches and origin servers manage the resulting `userprofile/ID1` as any other object. If the client request contains no `userid` object then the origin server provides the cache with an id for a new user.

User profiles may contain a number of distinct fields, not all of which may be required for a particular component. To access only the required field we propose to use a mechanism similar to that used by Web browsers to navigate to a specific named part of a page. We append '`#`' and the name of the required field to the name of the object. After obtaining `userprofile/ID1` from the origin server, the assembler replaces the outer `INSERT` tag with the content of the `userprofile/ID1` that is located between the `Name` and the next field, finishing the assembly. To prevent input parameters from masking those objects that should be retrieved from the origin server we use the `ISRC` attribute of the `INSERT` tag, instead of `SRC`, to explicitly inform assemblers that the included object may be available in the request.

The real strength of treating an entire user profile as any other object is that caches can avoid contacting the origin server when the same client requests pages that depend on other fields in the profile, such as address, e-mail or company name. Unless the profile changes at the origin server, caches continue using it to personalize pages that depend on it.

7.2 URL Re-Writing

Many Web sites wish to identify unique visitors and track paths that they follow through the site. Cookies often can not be used because many users turn cookies off in their browsers, and Web crawlers do not always support cookies. A more robust technique of differentiating between clients is URL re-writing, where for each client the server generates a unique identifier and dynamically appends it to each traversal link on each page that it serves to that client. Pages with re-written URLs are uncacheable even though their content does not change on every request.

Caches enhanced with our CA mechanism can re-write traversal links in cached pages by performing simple substitutions and can propagate unique IDs between cached pages without contacting the origin server. Content designers change traversal links in their pages from

`<a href="link.html">`

to

`<a href="link.html?sessionid=<INSERT ISRC=sessionid>">` , effectively decoupling the InpDep part from the rest of the page. The modified page now belongs to one of the categories in Figure 2, and can be cached if the category is not BoA. Upon receiving a request for such a page, a cache replaces the entire `INSERT` tag with the value of the `sessionid` found in the client's request. For example, a request for `page.html?sessionid=ID1` results in the re-written `page.html` containing the link `<a href="link.html?sessionid=ID1">`. If the client follows re-written links within that page, the cache re-writes those pages as well, assuming it has them cached, using the same `sessionid`. If another client requests the same pages, the cache re-writes them with the `sessionid` taken from that client's request. If a request does not contain a `sessionid` object, the cache obtains a new ID from the origin server. Cache can also prefetch a block of new IDs from the origin server in advance.

The reason sites deploy URL-re-writing is to log all requests and differentiate between unique visitors. CA-enabled caches annul the usefulness of the URL-re-writing to servers by shielding them from client requests. We propose to decouple serving cached content from propagating requests to the origin servers. Servers inform caches via a CCC command whether they wants to see each request for a given object right away, at some later point in time or never. If real-time feedback is not required, caches can aggregate requests for a given object and notify the server later, perhaps during off peak hours.

7.3 Input-Based Object Selection

When a Web site is offering content in more than one language encoding, the server decides on the correct encoding at the time of the access by examining the `Accept-Language` HTTP request header. The default installation of the Apache Web server [1], for example, comes with the default home page in a few languages `index.html.de`, `index.html.en`, `index.html.fr`, etc. Currently, such language-specific server responses can be cached with the addition of the `Vary` field that a subsequent client request must satisfy to receive the cached page.

Using our CA mechanism, caches can cache all objects that may result from a server making a selection from a finite set of choices, while the `Vary` field supports caching of only one object. Content providers add another object, `index.html`, that contains a single line:

```
<INSERT SRC=index.html.<INSERT ISRC=Accept-Language DEFAULT=en>>
```

Upon receiving a request for `index.html`, caches either use the language preference of the browser or the default value of `en` and construct the name of the object with the language-specific content.

8 Current Status

We are building a system based on the approach discussed in this paper. We have built a prototype *Web Object Cache Compiler* (*WOCC*). For each page at a Web site WOCC builds a graph representing relationships between objects composing a page. It then analyzes each graph and assigns appropriate CCCs to Static and Periodic objects, if any. WOCC also determines which object, the *manager*, should be used to manage non-deterministically changing objects on the page. Then it consults the matrix of all possible combinations of object change characteristics for parent-child relationship and extracts a template for the correct CCC for the manager and for each non-deterministically changing object on the page. WOCC expands the templates and assigns the resulting CCCs to the objects. WOCC generates CCCs for all objects as needed.

We have also implemented a prototype *Content Assembler* (*CASM*). CASM is a stand-alone proxy server capable of serving client requests by fetching the top-level component from the origin server, parsing it, and then recursively fetching all the included components and assembling the final page. CASM performs simple HTTP response header adjustment so that Web browsers can handle the assembled pages as regular HTML pages. We have also added basic in-memory caching of components and embedded objects. CASM behaves as a regular proxy server when a page requested by a client does not require assembly: it fetches the page and forwards it to the client. Initial testing shows that CASM does not introduce a noticeable delay.

9 Related Work

Challenger et. al. [5, 6] in their work on the IBM 1998 Olympics Web site developed Data Update Propagation mechanism to automatically update cache contents at the origin severs when underlying data changes. The same authors also proposed constructing pages from *fragments* that change at different rates and encapsulate semantically coherent content. These ideas are similar to ours. However, while the authors postulated that their work can be extended to proxy server environment, the main focus of their research was on improving Web server performance for serving frequently changing content to a large number of users. Our focus is on deterministic and efficient management of objects close to the users.

Authors of the Cachuma caching system [18] advocate grouping dynamic pages into classes based on URL patterns and exploiting coarse-grain dependencies between the resulting groups and underlying data. Servers invalidate a group of dynamic pages when underlying data changes. While Cachuma approach may require servers to maintain less state than our finer-grain approach, we believe invalidating entire pages when only a portion of underlying data changes is inefficient. Our approach in such situations invalidates only a single changing component, shared between all pages in a group.

Web site building tools, such as PHP [13] and Mason [10], allow designers to place content shared between multiple pages or content that changes differently from the rest into separate files or *components*. These tools also provide server-side caching mechanisms that cache the results of component execution. We are not aware of any work, however, that explicitly explored combining object change characteristics with object relationships for more deterministic object management.

Two main works on caching frequently changing Web pages are HPP and Active Cache. Douglis et. al. in their HPP work [8] proposed to treat rarely changing part of a page as a cacheable *template* and separate frequently changing portions of the page into a set of *bindings* that must be retrieved on every access and then expanded within the template by a modified browser of proxy. Our work suggests using a finer granularity of object change characteristics than just static and dynamic, giving caches better control over which content needs to be retrieved and which can be taken from the cache. Splitting each page in two parts does not allow the reuse of shared content between two pages. Also, if only one value in a set of bindings changes, the entire set must be retrieved. We propose to use a simple substitution for content assembly, while HPP handles custom macro language complete with conditional statements and loops.

Cao et. al. [3] in their Active Cache work and more recently Myers et.al in their Gemini work [11] proposed a more general method of handling small changes to cached objects. Content providers, in addition to content, provide specialized code that intermediate caching servers execute to produce a new version of the cached object. A possibility of malicious or badly written code supplied by origin servers raises security concerns. Java implementations of both Active Cache and Gemini also incurred non-trivial computational costs. Unlike the idea of having caches execute code provided by publishers, our CA mechanism requires intermediate nodes to perform only a single pre-defined task of tag substitution and avoids security and performance issues introduced by external code.

10 Summary and Future Work

In this paper we have presented the techniques that constitute our approach to Web object management and illustrate the improvements over current practice that each technique brings. First, we discuss the mechanics of how servers combine the relationships between objects composing Web pages with object change characteristics to produce object-specific CCCs. We show how the use of appropriate CCCs leads to more deterministic management of objects and eliminates unnecessary validation requests that are currently due to inefficient cache consistency mechanisms and have to be handled by servers. These are significant improvements over current practice.

Second, our powerful Content Assembly technique enables caches to construct pages from individual components. Each component encapsulates homogeneously changing and semantically coherent content, and only those components that change on every access must always be retrieved from the server. Servers and caches treat components just like objects and deterministically manage them via CCCs. Relocation of page assembly away from the servers and closer to the users allows more content to be cached.

Third, we demonstrated how CA-enabled caches can cache components that are currently uncacheable due to being personalized or customized based on the input parameters. CA-enabled caches perform the required customization upon a client request. This capability further increases the amount of currently uncacheble content that can now be cached.

Our Content Assembly technique is not only powerful, but also simple. Our content assemblers perform only a single pre-defined task of tag substitution thus avoiding the security and performance issues raised by alternative proposals advocating more complex methods of page construction by caches.

Current and future work on the prototype implementation includes extending CASM to support CCCs, integrating CASM with an existing Web proxy server, such as Squid [14], and integrating WOCC with an existing Web server, such as Apache [1]. We will also continue to evaluate performance of our system.

References

[1] The Apache Server Project. http://www.apache.org.

[2] Martin Arlitt and Tai Jin. A Workload Characterization Study of the 1998 World Cup Web Site. *IEEE Network*, May/June 2000.

[3] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching dynamic contents (objects) on the Web. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, September 1998.

[4] Vincent Cate. Alex - a Global Filesystem. In *Proceedings of the USENIX File Systems Workshop*, pages 1–12, May 1992.

[5] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of the IEEE Infocom '99 Conference*, New York, NY, March 1999. IEEE.

[6] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic web content. In *Proceedings of the IEEE Infocom 2000 Conference*, Tel Aviv, Israel, March 2000. IEEE.

[7] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *ACM SIGCOMM'98 Conference*, September 1998.

[8] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, December 1997.

[9] Balachander Krishnamurthy and Craig E. Wills. Piggyback server invalidation for proxy cache coherency. In *Seventh International World Wide Web Conference*, pages 185–193, Brisbane, Australia, April 1998.

[10] Mason. http://www.masonhq.com.

[11] Andy Myers, John Chuang, Urs Hengartner, Yinglian Xie, Weiqiang Zhuang, and Hui Zhang. A Secure and Publisher-Centric Web Caching Infrastructure. In *Proceedings of the IEEE Infocom 2001 Conference*, Anchorage, Alaska USA, April 2001.

[12] Erich M. Nahum. WWW Workload characterization work at IBM Research. In *Web Characterization Workshop*, Cambridge, MA, November 1998. World Wide Web Consortium. http://www.w3.org/1998/11/05/WC-workshop/Papers/nahum.html.

[13] PHP Hypertext Preprocessor. http://www.php.net.

[14] Duane Wessels. Squid Internet Object Cache. http://squid.nlanr.net/Squid/.

[15] Craig E. Wills and Mikhail Mikhailov. Towards a better understanding of Web resources and server responses for improved caching. In *Eighth International World Wide Web Conference*, Toronto, Canada, May 1999.

[16] Craig E. Wills and Mikhail Mikhailov. Studying the Impact of More Complete Server Information on Web Caching. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[17] Craig E. Wills, Mikhail Mikhailov, and Hao Shang. *N* for the Price of 1: Bundling Web Objects for More Efficient Content Delivery. In *Tenth International World Wide Web Conference*, Hong Kong, May 2001.

[18] Huican Zhu and Tao Yang. Class-Based Cache Management for Dynamic Web Content. In *Proceedings of the IEEE Infocom 2001 Conference*, Anchorage, Alaska USA, April 2001.