Class-based Cache Management for Dynamic Web Content

Huican Zhu and Tao Yang
Department of Computer Science, University of California
Santa Barbara, CA 93106
{hczhu, tyang}@cs.ucsb.edu

Abstract

Caching dynamic pages at a server site is beneficial in reducing server resource demands and it also helps dynamic page caching at proxy sites. Previous work has used fine-grain dependence graphs among individual dynamic pages and underlying data sets to enforce result consistency. Such an approach can be cumbersome or inefficient in dealing with an arbitrarily large number of dynamic pages. This paper studies partitioning dynamic pages into classes based on URL patterns and the proposed scheme allows an application to specify page identification, cachability and data dependence and invoke invalidation for a class of dynamic pages. To make this scheme time-efficient with small space requirement, lazy invalidation is used to minimize slow disk accesses when identifications of dynamic pages are stored in memory with a digest format. Selective precomputing is further proposed to regenerate stale pages and smoothen load peaks. A data structure is developed for efficient URL class searching during lazy or eager invalidation. This paper also presents design and implementation of a caching software called Cachuma which integrates the above techniques, runs in tandem with standard Web servers, and allows Web sites to add dynamic page caching capability with minimal changes. The experimental results show that the proposed techniques can efficiently handle class-based page invalidation and are effective in reducing server response times for tested applications.

1 Introduction

Dynamic page generation based on user profiles and request parameters has become popular at many Web sites since it enables new types of services such as electronic commerce and personalized information presentation. A number of techniques have been proposed to speed up dynamic page delivery, including efficient OS support [5, 24], server clustering and load balancing [2, 11, 15, 31], content preprocessing [12], and dynamic content caching at server and proxy sites [17, 18, 26, 27]. This paper focuses on caching and invalidating dynamic pages at the server sites. Previous work on Web caching and invalidation [8, 16] at proxy servers mainly deals with static pages.

Caching dynamic pages at server sites has several advantages: 1) it reduces server computing resource demands; 2) it simplifies page invalidation processes since it is easy for a Web site to control its cache; 3) it also helps proxies to cache dynamic pages for network traffic reduction because the creation time of a dynamic page cached by a server can be used to answer an "if-modified-since" query from a proxy. The work in the IBM Olympics Web server [10, 18] and in the Alexandria Digital Library system [3, 17] demonstrated the feasibility and significance of caching dynamic content.

A major problem for dynamic content caching is to ensure that strong consistency be maintained between cached pages and underlying data objects. Holmedehl et al. [17] uses the standard time-to-live (TTL) technique which is applicable to limited applications such as online searches. Vahdat et al. [27] proposes the idea of intercepting file operation related system calls to monitor change of source files. The weakness is that system call monitoring is limited to local file systems and can slow down other operations. For more complicated servers which involve queries on distributed database systems, Iyengar et al. [10, 18] proposes an approach which requires applications to explicitly issue invalidation messages to a cache. While this approach is powerful for many Web applications, it relies on maintaining a fine-grain dependence graph to specify dependence among individual Web pages and underlying data sets (e.g. database tables). For many sites which host an arbitrarily large number of dynamic pages (e.g. pages generated using individual user profiles), the size of such a fine-grain graph can grow enormously and enumerating and maintaining dependence in such cases can be cumbersome. It can also become inefficient if the dependence of some pages on a data object is not registered to the cache in advance because its change will result in an invalidation message being sent for every page depending on this object.

We propose a more comprehensive and flexible caching scheme which allows users to specify coarse-grain dependence among underlying datasets and a set of dynamic pages called URL classes which share common URL patterns or client information. As a consequence, application programmers do not need to enumerate dependence structures for individual pages. This scheme also supports group-based invalidation of pages which share common patterns and these patterns can be unknown to the cache in advance.

To make the above group-based scheme efficient, we propose several techniques. First, since URLs of cached pages are normally stored in main memory in a digest format, problems arise when some pages matching a URL pattern need to be invalidated but such a pattern is not pre-registered to the cache. Determining what pages match such a pattern becomes virtually impossible without knowledge of page URLs. We use a lazy invalidation strategy that accumulates invalidation messages and only performs necessary page invalidation when a client access to a page is initiated. Second, to support dynamic page prefetching for increasing cache hit ratios, we further study eager invalidation for a subset of pages to allow for selective page precomputing. Selective page precomputing balances server load along the time line and minimizes chances of server load spike. We have also developed a data structure for efficient URL class searching during lazy or eager invalidation. We have designed and implemented a caching system called Cachuma which runs in tandem with a standard Web server such as the Netscape Enterprise Server and the Apache server without modifying server source code. This allows Web sites to add caching capability with minimal changes (applications still need to provide cache-control specification and send invalidation messages).

The rest of this paper is organized as follows. Section 2 describes class-oriented caching specification and invalidation. Section 3 presents lazy and eager page invalidation with selective precomputing. Section 4 discusses interactions among the server, cache and applications and presents an efficient data structure for searching URL classes. Section 5 describes our simulation and experimental results. Section 6 is related work. Section 7 concludes the paper.

2 URL Classes and Cache Control Specification

In this section, we first discuss how dynamic pages are partitioned into groups based on their URL patterns or client information. We then discuss how an application specifies identification, dependence and invalidation for a group of pages.

2.1 URL Classes

We group Web pages into classes based on page URLs and requesting client information. Page grouping allows applications to cache and invalidate pages in a group basis rather than individually. We call pages that share the same patterns as *URL classes*. The URL information used for classifying pages includes net paths of application scripts that generate pages, application script names, and parameter values. Additional information for page classification includes client domain names, client IP addresses, and cookie values. The BNF specification of a URL class is shown in Figure 1. The keywords "**__pathinfo**" and "**__Cookie:**$< ID >$" deserve more elaboration. Some Web programming frameworks, including CGI and Java servet, allow extra information to be supplied in a URL as file paths following an application name. Instead of treating the extra path information as part of a URL net path, we treat it as value of the special parameter **__pathinfo**. Though cookies do not appear in page URLs, their tremendous importance for dynamic contents justifies special treatment. We use the keyword **__Cookie:** to identify cookies appearing in user request headers. A cookie with name "foo" and value "bar" is included in a URL class as a parameter in the format of "**__Cookie:**foo=bar".

$< URL\ class >$ **:=** $<net\ path\ directory\ name > \mid <application\ name >$**'?'**$< parameter - condition >$
$< parameter - condition >$ **:=** $< arg > = < value > \mid < arg > = < value >$**'&'**$< parameter - condition >$
$< arg >$ **:=** **__pathinfo** | **__client-domain** | **__client-IPaddress** | **__Cookie:**$<ID>$ | $< ArgID >$

Figure 1: The BNF specification of a URL class.

Some example URL classes and their meanings are given as follows:

- `/cgi-bin`. It covers all dynamic pages which are generated by scripts with /cgi-bin as their net path prefix.
- `/cgi-bin/news?topic=sports&country=USA`. It groups pages which are generated by script `/cgi-bin/news` with parameters `topics=sports` and `country=USA`. This script may have other parameters, but they are not used by this class.
- `/cgi-bin/transaction?_cookie:Session=1234AB`. It covers all pages which are generated by /cgi-bin/transaction and have cookie `Session=1234AB`.

As to be explained in next section, an application specifies cachability and invalidation of pages through their URL classes and we need to define how a page matches a class. We say that a page *matches* a URL class if: (1) the path of the URL class is a prefix of the URL path of this page; and (2) parameters of this page satisfy parameter conditions of this class. Because one URL class may be a *subclass* of another URL class or two URL classes may *intersect*, a page may match multiple URL classes. If cachability and dependence specification for multiple URL classes have a conflict, we need to resolve such a conflict. We will use a concept called *minimal* class which is defined as follows. Let G be a set of URL classes, a URL class $\mathcal{C}$ is a minimal class of a page p with respect to G if: (1) p matches $\mathcal{C}$; (2) there does not exist any class $\mathcal{D}$ in G such that p matches $\mathcal{D}$ and $\mathcal{D}$ is a subclass of $\mathcal{C}$. Notice that a page can still have multiple minimal classes.

Figure 2 demonstrates four URL classes and two minimal classes of a page. The four URL-classes are $\mathcal{C}_1$=/cgi-bin, $\mathcal{C}_2$=/cgi-bin/sports, $\mathcal{C}_3$=/cgi-bin/sports?country=USA, and $\mathcal{C}_4$=/cgi-bin/sports?year=1999. Page p with URL=/cgi-bin/sports?country=USA&year=1999&category=tennis matches all the four classes, but only $\mathcal{C}_3$ and $\mathcal{C}_4$ are its minimal classes.

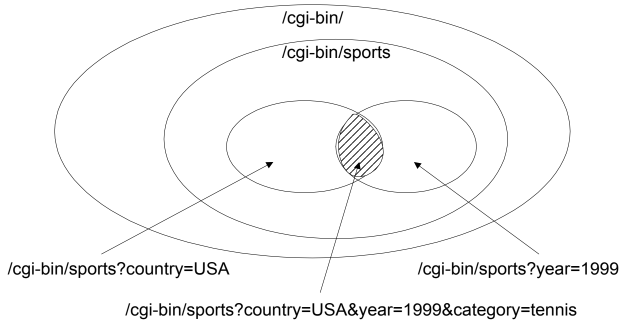


Figure 2: Four URL classes and one page are listed. There are two minimal classes for this page.

2.2 Class-based Cache Control Specification

We use URL classes to manage the three indispensable facets of page caching: (i) page cachability and dependence specification, (ii) cached page identification, and (iii) page invalidation.

Page cachability and dependence information specifies whether some pages are cachable, and if cachable, what data objects they depend on. In our scheme, dynamic pages are non-cachable by default and applications need to explicitly register that a set of URL classes are cachable. Subclasses of cachable classes may be registered as non-cachable, and vice versa.

Page identification specifies that besides URL, what additional information is necessary to differentiate cached pages. The additional information includes cookies and client IP addresses. Such information is necessary because for a URL such as "http://my.yahoo.com/", the same URL is used by millions of users to retrieve their customized pages and cookie values in request headers are used to differentiate them.

Figure 3 illustrates cachability and object dependence specification for two URL classes. The first example specifies that pages in URL class `/cgi-bin/news?topic=stock` are non-cachable. The second example says that a page generated by a script with net path `/cgi-bin/news` is cachable and such a page should be invalidated when data object "news-table" changes. Note that here "news-table" is just an identifier. It could represent any data object, e.g. a database table. The "Page-ID" keyword in the second example further specifies that the value of the cookie "username" is used to construct page IDs for pages matching this URL class.

```
URL-Class:  /cgi-bin/news?topic=stock
Cachable:  No
-------------------------------------
URL-Class:  /cgi-bin/news
Cachable:  Yes
Dependence:  news-table
Page-ID: __cookie:username
```

Figure 3: Cachability and dependence specification for two classes.

The difference should be noted between cookies appearing in a URL class specification and those appearing after "Page-ID" keyword. The former specify what cookies are used to group pages for collective control. The latter specify that for pages matching the same URL class, what cookie values should be used to differentiate them. Cookies appearing in URL class specifications are also used for constructing page IDs.

If a page matches multiple classes and specifications for those classes conflict, we adopt a conservative rule to resolve such a conflict: (1): A page is non-cachable if any of its minimal classes is non-cachable. (2): A page depends on an object if any of the classes it *matches* depends on this object.

Cached pages are invalidated during runtime by messages sent from applications that directly or indirectly trigger invalidation of URL classes. There are three types of invalidation messages:

- **Object-Change**: indirect class invalidation by specifying object change.
  For example: "`Object-Change: news-table`". When such a message arrives, all pages *matching* URL classes depending on object `news-table` should be invalidated.
- **Invalidate-Class:** direct invalidation of a URL class.
  For example: "`Invalidate-Class: /cgi-bin/news?topics=sports`". All the pages *matching* the URL class given in the invalidation message should be invalidated.
- **Invalidate-Page**: direct invalidation of an individual page.
  For example: "`Invalidate-Page: /cgi-bin/txt2ps?file=README.txt`".

The work in [10] uses object changes to trigger invalidation of individual pages. Our scheme expands that work by allowing class based page invalidation. Furthermore, in out scheme, pages can be invalidated not only by notifying underlying object changes, but also by specifying a targeting URL class. One thing covered in [10] is that pages may be invalidated due to dependence between cached pages. Since the authors in [10] indicate that in practice inter-page dependence is rarely used, we have not considered inter-class dependence. If needed, inter-class dependence can be transformed into object dependence.

Conceptually, our page caching and invalidation mechanism works as follows: There is a caching system (or *cache*, for short) which manages page caching and invalidation. Each accessed dynamic page is stored in the cache and is identified by a unique page ID derived from the page URL and possibly other information such as cookies. When a request for a dynamic page arrives, the URL and the related information are extracted to construct an ID for locating a cache entry. If a cached version is found and is fresh, its content is sent to the client. If the requested page is not cached, the server invokes the application to generate a response and sends it back to the client. Meanwhile, cachability of this page is checked. If cachable, a new cache entry is created in the cache. When an invalidation message arrives, it triggers invalidation of an individual page or a group of cached pages.

The above procedure looks simple, however, its detailed design and implementation is complicated due to performance concerns. The following two sections present key problems we encounter and our solutions.

3 Page Invalidation and Precomputing Techniques

When a URL class is to be invalidated, we need to locate all the cached fresh pages which match the URL class. If the cache system is aware of URL classes which are frequently invalidated, it can maintain a mapping between those URL classes and their matched pages to speed up page searching. But the searching process can be expensive if the cache is not "aware" of the targeted URL class. For this reason, we identify two types of URL classes: predictable and unpredictable. For URL classes which are frequently invalidated, they can be made aware to the cache (e.g. during cache bootstrap time) before pages matching them are cached. We call these classes as *predictable URL classes*. All other URL classes are called *unpredictable URL classes*.

A Web site may be tempted to register as many predictable classes as possible. However, If there are too many predictable classes, maintaining all dynamic links can become expensive. Moreover, there could be an efficiency concern if, when a page is created, we have to find all the URL classes that match this page.

In this section, we address the above issues. We first propose a technique called lazy page invalidation to deal with unpredictable URL classes, which can also be applied to predictable URL classes. We then present precomputing techniques for pages matching predictable URL classes. Finally we discuss techniques for selective precomputing.

3.1 Lazy Page Invalidation for Unpredictable URL Classes

When an invalidation message targeting an unpredictable URL class arrives, we need to compare the URLs of cached pages with this URL class. For efficiency, page identifications (URL and cookies) should be available in the main memory. Due to space concern, A typical memory-efficient way to manage a Web cache (e.g. in the Squid proxy server [28]) is to store page content in the secondary storage and only keep in memory key meta information such as page ID, last access time, and creation time. Squid uses the 16-byte MD5 digests [25] computed from page URLs as page IDs and the space saving by using MD5 key could be tremendous because the length of a URL can be several hundred bytes. For example, in the Alexandria Digital Library [3], the length of URLs related to map retrievals can exceed 1KB. We also use the MD5 encoding method in our cache to save memory space. As a result, actual page URLs are not readily available in memory for comparison with an unpredictable URL class because the inverse of an MD5 digest is not tractable. An apparent solution is to store all complete page URLs in a URL file and store a location index for a URL in the in-memory meta-information entry of each page. Then that index can be used to retrieve the corresponding URL for a cached page. This is actually the technique we used, but for page precomputing purpose, as to be discussed later. However this technique is not applicable for page invalidation because we will have to read in hundreds or thousands of URLs and compare them with an unpredictable URL class one by one. Such a process could be very time consuming.

Our approach is to use lazy invalidation: *invalidation is not performed until a cached page is accessed.* To support lazy invalidation, we maintain a set of invalidated URL classes annotated with the latest invalidation time and call this set *I-set*. Lazy invalidation works as follows:

1. When an invalidation message is received, if it is of type "Invalidate-Page", the targeted page is located and invalidated directly. If the invalidation message is of type "Invalidate-Class", the targeted URL class is recorded in the I-set and is marked with the new invalidation time. If the invalidation message is of type "Object-Change", all the URL classes depending on this object become stale and are recorded in the I-set. Notice that only the latest invalidation time needs to be maintained for each invalidated URL class.

2. When a request for a dynamic page is received and a fresh cache entry is found, the I-set is searched first to check if it contains any URL class which matches the cached page and has invalidation time after the page's creation time. If such a class is found, this page is stale and should be regenerated.

Lazy invalidation incurs I-set searching overhead for each page access and we will present an efficient data structure for searching in Section 4.2. The size of the I-set should not grow too big in practice because: (1) For invalidation messages targeting the same URL class, we only need to keep the latest copy; (2) For invalidation messages targeting single pages, they are are not maintained in the I-set. In the worst case, we impose a limit for the size of the I-set. When this limit is reached, pages generated before a certain time are invalidated and invalidation messages with arrival times earlier than this time stamp are removed from the I-set.

3.2 Page precomputing for predictable URL classes

For predictable URL classes, we can maintain links between them and cached pages matching them. The matched pages can then be invalidated immediately (or "eagerly") once such a URL class is invalidated. Certainly lazy invalidation can also be used for handling predictable URL classes and this is especially useful when there are too many predictable URL classes registered because lazy invalidation does not need to maintain and store links between classes and pages. However eager invalidation for predictable URL classes allows us to benefit from another performance enhancement technique: page prefetching.

Prefetching static pages has been studied in previous research and has shown modest performance improvement in reducing page delivery latency [14, 20]. Because dynamic pages are usually short-lived, prefetching dynamic pages can be more effective in improving cache hit ratios. In fact, the IBM 1998 Olympic Web site recomputed every cached page immediately after its invalidation and as a result, a 20% cache hit ratio improvement was observed [10]. As we show in Section 3.3, by prefetching dynamic pages when server load is low, a busy Web site can handle more concurrent requests gracefully at its peak load. To emphasize the computation involved for dynamic page prefetching, we refer to this process as *page precomputing.*

Page precomputing requires knowledge of stale pages, but lazy invalidation prevents us from locating them because the cache is not aware if a page is stale until this page is accessed by some client. Therefore we use eager invalidation for predictable URL classes. In addition to maintaining the I-set, we build a class-page link set (CP-set) which maintains pointer lists from predictable URL classes to cached pages. The CP-set contains a set of URL classes which are frequently invalidated. When an accessed page is cached, a pointer to in-memory meta information of this page is added to URL classes with which this page matches. When a URL class in the CP-set is invalidated, all cached pages matching this class are invalidated immediately and become candidates for precomputing. When a page is to be precomputed, we follow its in-memory meta information to retrieve ID of the page from secondary storage and then issue a precomputing request for this page.

There are three issues that need to be addressed: 1) How can the system know which URL classes are predictable and should appear in the CP-set? 2) How is the size of a CP-set controlled? 3) Is it still necessary to check the I-set for each fresh page access? Our answers are as follows:

1. Our system provides an API for applications to pre-register predictable URL classes which are frequently invalidated. URL classes which are used in cachability and dependence specification are also considered predictable and are maintained in the CP-set. Therefore, the CP-set also provides information for page cachability and dependence specification, and page Id construction.

2. To reduce space usage, the link to a page is only added to its minimal classes with respect to the CP-set. The complication introduced is that, to eagerly invalidate pages matching a URL class, we also need to traverse class-page links of its subclasses. Our data structure for organizing URL classes allow us to locate those subclasses efficiently. The system also provides a parameter for restricting the memory limit of a CP set and drops links to least-frequently-accessed pages when the limit is exceeded.

3. The system still needs to maintain the I-set and verify page freshness upon page accesses. There are two reasons. First, since a class appearing in an invalidation message might not be in the CP-set (e.g. due to space limit, or because it is an unpredictable class) or links to pages matching this class might not be maintained (e.g. due to space limit), this class needs to be added to the I-set. Second, since a class in the CP-set could link a large number of cached pages, invalidation process for a class can take time. Race condition arises if during invalidation, requests for pages matching the class being invalidated arrive. Some pages which should be invalidated may be regarded as fresh and sent out. To avoid such a case, the I-set should contain all invalidated classes. Certainly freshness verification for most of cached pages is redundant. However as we will discuss in Section 4.2, our data structure allows fast searching of URL classes and this checkup overhead is negligible.

The following is the sketch of our page caching and precomputing algorithm.

1. When a request for a dynamic page is received, if this page is not cached but is cachable, a new cache entry is created and a pointer to this entry is added to all of its minimal URL classes in the CP-set. Otherwise, if a fresh cache entry for this page is found, the I-set is searched to see if any URL class matching this page exists which invalidates this page. The page is returned if it is still fresh. Otherwise it is re-generated.
2. When an invalidation message is received, if it invalidates a single page, that page is invalidated immediately; if it targets at URL classes, the targeted URL classes are recorded in the I-set first, and then all the identifiable pages pointed to by the targeted URL-classes and their subclasses in the CP-set are invalidated.
3. After invalidation, URLs of stale pages are saved in a special list for precomputing purpose. The details on precomputing will be discussed in the next subsection.

3.3 Instant precomputing vs. selective precomputing

At the 1998 Olympic Games Web site [10], dynamic page precomputing is conducted whenever a page becomes stale. We call this *instant precomputing* and it is effective when the number of dynamic pages is relatively small. However instant precomputing is not feasible for Web sites hosting a large number of dynamic pages which require a fair amount of system resources because it can create significant resource contention for processing user-initiated requests.

Our strategy is selective precomputing: *precomputing is conducted only when the system resource usage is below a certain threshold in terms of a composite load index.* Let ω be the average percentage of time spent on CPU for computing a dynamic page on a machine. Let x be the percentage of CPU idle time on this machine during last t seconds (we use $t = 5$ in our tests) and y be the percentage of I/O idle time (x and y are reported by "rstat" utility). And let H be load threshold (we use $H = 40\%$ in our experiments). We use the following condition to check the feasibility of computing such a dynamic page on this machine: $\omega\frac{1}{x}+(1-\omega)\frac{1}{y} \leq \frac{1}{H}$ This load index actually approximates the expected stretch factor of processing such a page under current load. A threshold of 40% means that a page should not be precomputed if its processing time will be lengthened by a factor of 2.5. Currently this composite load index only covers I/O and CPU load on machines where content-generation applications are executed. While the load model is effective for working with the Alexandria digital library server and a few other content-generation applications [32], further studies may be needed to model other resource requirement such as local network load and multiprocessing.

When cachability and dependence information of a URL-class is registered, its resource demands can also be specified as "`Resource:<machine-name>, <weight>` ". `<machine-name>` is the machine on which pages matching this URL-class are generated. It is not necessarily the machine running the Web server or the cache system. The `<weight>` field is the percentage of time spent on the CPU of the corresponding machine. It should be noted that in an advanced system which may have replicated servers [32], the above scheme can be extended to list multiple machines for each URL class and the system selects the best one for content regeneration.

There are still two questions to be answered: How many precomputing requests are issued when system load is low? Which dynamic pages should be precomputed? We answer these two questions as follows.

1. The answer for this question depends on how aggressive an application wants to prefetch dynamic pages. The criterion is that the system resource should be fully utilized while not flooding the server with precomputing requests. In our experiments, we issue precomputing requests to the server sequentially to avoid overloading the server.

2. Invalidated page entries are maintained in a stale page list sorted on their last access times. Entries with large last-access-time values (recently accessed pages) are selected for precomputing. The philosophy used here is the same as for the LRU algorithm: pages accessed most recently are expected to be accessed soon and should be precomputed first. Note that since we only precompute invalidated page entries, precomputing does not introduce new entries to the cache.

4 Caching System Design and Implementation

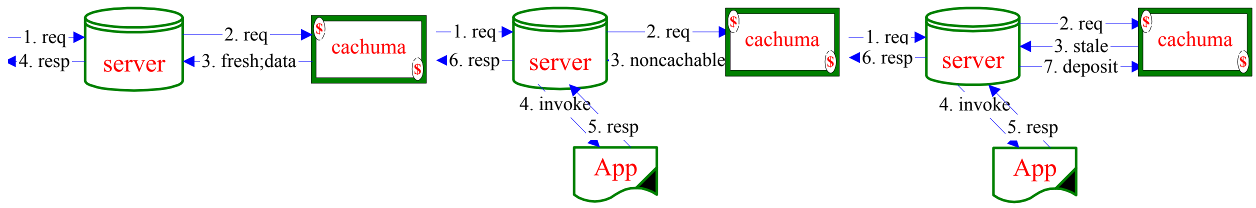
4.1 System Architecture

Our design goal is to add caching capability to existing commodity Web servers without modifying server source code. Thus our dynamic page cache is designed and implemented as a component independent of Web servers. We have successfully integrated our cache manager prototype (Cachuma) with Apache 1.3 server, Netscape Enterprise Server 3.6, and the Swala server [17], based on APIs of those servers.

The functionalities of Cachuma include: storing cached dynamic pages, invalidating cached contents when it receives invalidation messages from applications, issuing requests to the server for precomputing when system resources are available, and executing an LRU cache replacement policy when cache is full. Its data components include the I-set, the CP-set, cachability and dependence specification, and a table of cache entries storing page meta-information, including MD5 digests of page URLs and some other bookkeeping fields. Instead of sitting in front of a Web server as a normal proxy or accelerator, Cachuma sits behind the Web server. We made such a decision based on two reasons: 1). We do not want to duplicate URL and header parsing functionalities in the cache. 2). Many requests require authorization and authentication, which should be done by the server.

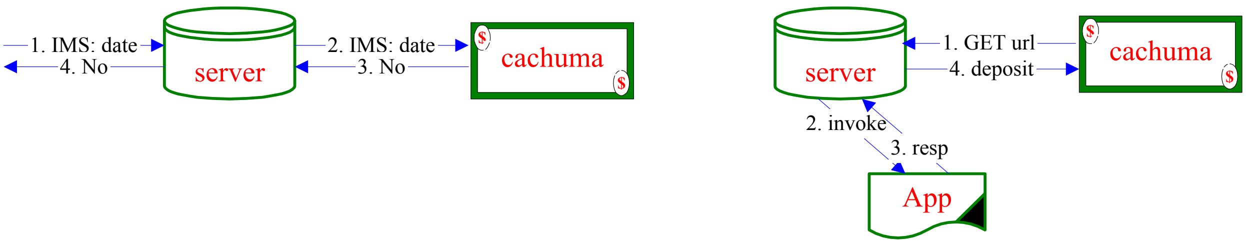
Cachuma and a Web server interact in the following three scenarios:

- **Content caching and retrieving**: There are three cases: cached page is fresh, page is not cached or is stale, and page is non-cachable. Figure 4 illustrates the steps involved for the three cases.



(a). Fresh cache hit (b). Non-cachable page request (c). Stale cache hit

Figure 4: Interaction between the server and the cache when handling client requests.



(a) IMS request processing (b) Page precomputing

Figure 5: Content validation and precomputing

- **Content validation:** Cachuma supports the weak page validation model specified in HTTP/1.1. Upon receiving an "If-Modified-Since" request from a client, the server passes it to Cachuma. If the corresponding page is not cached or is stale, the server invokes the application to generate a response. If the cached copy was created later than the IMS time-stamp, the cached page is retrieved to answer this request. Otherwise, the server sends "304 Not Modified" response to the requesting client. Figure 5(a) demonstrates steps involved in the last case.

- **Precomputing:** There are two mechanisms to invoke an application for page precomputing. 1) Let Cachuma invoke the application. This solution requires Cachuma to replicate all facilities relevant to executing programs in a Web server (e.g. setting up environment variables). 2) Let Cachuma issue requests to the Web server and then the Web server invokes applications to generate responses. Currently we use the second mechanism since it is much simpler. To inform the Web server a request comes from the cache other than a real client, we add header `From-Cache:true` to the HTTP header part of the request. The server can then look for this header to determine the origin of a request. This mechanism greatly simplifies the implementation with added small overhead due to server participation. Since precomputing is invoked only when the server load is low, the overhead for this indirect application invocation is negligible. Figure 5(b) demonstrates the steps for precomputing.

4.2 Data Structure for Efficient Search

A critical issue in implementing Cachuma is to search the CP-set and I-set for page invalidation. There are two fundamental searching problems: 1) given a page URL, find its minimal URL classes in the CP-set to set up the class-page links. 2) given a page URL, find all URL classes in the I-set that may invalidate this page.

We use a data structure composed of a *trie* and a few *parameter classifiers* for both I-set and CP-set. A trie [19] is a string tree in which common prefixes are represented by tree branches, and the unique suffixes are stored at the leaves. We use a trie to classify net paths and attach a parameter classifier, as defined in next paragraph, to some leaf nodes of the trie to capture parameter conditions of URL classes with the same netpaths. Trie is a well studied data structure for longest prefix match and we simply use a trie searching algorithm from our previous work which enhances processor caching performance by dynamically transferring trie nodes among small arrays, B-trees, and hash tables [1]. The real difficulty lies in the parameter classifier which has to determine what parameters from a page should be used for locating matching URL classes.

Given a set $\mathcal{G}$ of URL classes with the same net path, a parameter classifier for this set has three components: a parameter name index table, a sorted list of parameter groups, and a hash table for each parameter group. (1) The parameter name index table maps all parameter names used in $\mathcal{G}$ into consecutive integers starting from 0. (2)Each entry in the parameter group list is represented by a bit vector and covers the URL classes which use the same set of parameter names. The i-th bit in the bit vector is set if the corresponding parameter (derived from the parameter name index table) appears in this group of URL classes. (3) The hash table for each parameter group further differentiates URL classes based on parameter values. When searching for URL classes that match a page p, the parameter name index table decides what parameters are useful in p and their corresponding locations in bit vectors representing parameter groups. Each parameter group then decides a subset of the useful parameter names and their values to construct a string for hashing within the group. Obviously p can match at most one URL class in each parameter group.

Figure 6 illustrates our data structure for a CP-set with six URL classes: `/cgi-bin/news`, `/cgi-bin/sports/tennis`, `/cgi-bin/sports/soccer?events=world-cup`, `/cgi-bin/sports/soccer?country=Italy`, `/cgi-bin/sports/soccer?country=Germany`, `/cgi-bin/sports/soccer?country=Italy&events=League-match`. There is a parameter classifier attached to node "soccer" for the four URL classes with the same net path "`/cgi-bin/sports/soccer`". This classifier uses two parameters "country" and "events" which are mapped to bit location 0 and 1 respectively. There are three parameter groups in the classifier. The first group uses both "country" and "event' parameters. The second group uses parameter "event" only, and the third group uses parameter "country" only.

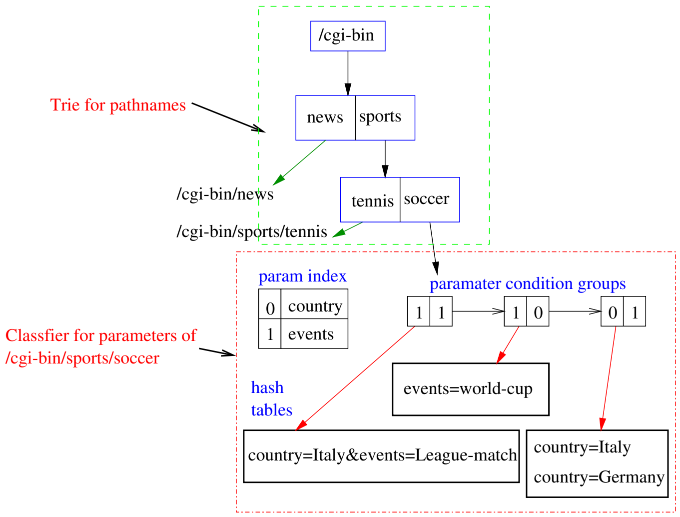


Figure 6: Data structure for a CP-set with six URL classes.

Given a page p, we describe how to find the minimal classes of this page using the above data structure. Notice that once a minimal class is found, searching its superclasses should be avoided. For this purpose, the group bit vectors in a classifier are sorted decreasingly based on their numerical values. If a URL class x is a subclass of class y, then the bit vector v_x for x's group contains the bit vector v_y for y' group[1]. The numerical value of v_x is therefore larger than that of v_y and appears first in the list. When searching for p's minimal classes, if x is found to match p, it will not be necessary to search within parameter groups whose bit vectors are contained in v_x because URL classes matching p from those groups can not be p's minimal classes. For the example in Figure 6, we illustrate how to find the minimal classes of URL `/cgi-bin/sports/soccer?country=Italy&events=League-match&year=1999`. We first find the classifier for net path `/cgi-bin/sports/soccer`. From its parameter index table, we know that only two parameters "country" and "event' are useful. We first compare this page with the first group with bit vector $(1,1)$ and we find a match in this group. The next two group bit vectors to check are $(1,0)$ and $(0,1)$. We skip them because these two groups cannot contain p minimal classes.

Note that the bit vector ordering trick only works when looking for minimal classes. In the worst case, we may still need to do a hash lookup for every parameter group in a classifier. However, we do not expect that searching all groups creates performance concerns because although the number of possible values for parameters could be unbounded, in practice the number of parameter names and their possible combinations used for defining URL classes is expected to be small. Therefore, the size of a parameter group list is not large for searching. Our test result in Section 5.4 verifies efficiency of our data structure.

4.3 Cost Analysis for Searching URL Classes

The space cost for the CP-set is proportional to the number of predictable URL classes and the number of cached pages belonging to these classes. The space cost for the I-set is proportional to the number of URL classes invalidated. Section 3.1 and Section 3.2 have discussed the space and size control issues for these two sets. As a result, their space costs are small.

We further merge multiple hash tables in parameter classifiers for space optimization. Maintaining many hash tables for different classifiers complicates implementation and reduces space efficiency. One simplification is to merge multiple hash tables in a parameter classifier into one table. It is clear that no entries in those original tables are the same. As a further optimization, we also merge the hash tables from all the classifiers into one. To make sure that all entries are unique, we append the virtual memory address of each classifier to the hash entries from that classifier.

Next we discuss time complexity. Given a trie augmented with parameter classifiers for a set of URL classes, let N_p be the maximum number of path segments of a URL class (i.e. the depth of this trie) and N_g be the maximum number of parameter groups in a parameter classifier. We can show that the overall time complexity of searching is $O(N_p + N_g)$. The reason is as follows. The total searching time includes the time for locating a longest prefix from the trie and time for locating matching classes within a classifier. The time for searching a trie is $O(N_p)$ because time spent on each trie node in matching a path segment is constant in practice. Once a classifier is located, we need to scan all parameter groups within this classifier. We have discussed in the previous subsection that a bit vector containment test and a possible hash table lookup are needed for each parameter group. To determine if a bit vector v_1 is contained in bit vector v_2, we can use the C expression "`!(`v_1 `&` $\sim$ v_2`)`" whose cost is constant. Since hash table lookup time is also constant in practice, we conclude the time spent in a classifier is $O(N_g)$ and the total time for searching is $O(N_p + N_g)$.

[1]A bit vector v_x contains another bit vector v_y if any bit positions set in v_y are also set in v_x.

5 Experimental Results

In this section, we illustrate the use of Cachuma in three applications: on-line forum, customized news service, and on-line auction, and demonstrate performance of the proposed techniques. We also study effectiveness of our data structure by examining searching overhead.

In the experiments, we run a Swala Web server, the Cachuma caching system, and the involved applications on the same machine. All the tests were done on a 128MB memory 248MHz Sun Ultra-30 machine running Solaris 2.6. Client requests are generated from machines within a local area network following some real traces or client behavior models. For the three applications we have studied, we compare the response time and cache hit ratios for the following four strategies: (1) no dynamic page caching; (2) caching with lazy invalidation; (3) Caching with instant precomputing; and (4) Caching with selective precomputing. Page precomputing is invoked when the host CPU utilization and I/O bandwidth consumption over a certain period (5 seconds in our tests) are both below a threshold (40%).

It should be noted that collecting traces and conducting evaluations for dynamic pages are harder than for static pages. This is because dynamic contents usually involve sensitive or personal data which web sites are reluctant to share. Even if we have traces for dynamic page accesses, it is difficult to replay such a trace because replaying requires setting up application execution environments such as database backends. Our strategy is either to develop an application prototype to simulate behavior of a Web site (in the case of auction) or make assumptions on server execution time in processing dynamic requests (in the case of on-line forums and customized news service). For client access patterns, we either use a real trace (in the case of on-line forums) or develop a state transition graph to model client behaviors (in the case of auction and news service). The state transition graph approach was used in a recent study on e-commerce user modeling [22].

5.1 Online Forums

Web site `www.melissavirus.com` provides an on-line discussion forum on the notorious Melissa Virus after its tremendous destruction to computers worldwide. By observing how this Web site responds to client requests, we have developed an application to mimic the design and behavior of this Web site. We have also obtained an access log of this Web site and use it to evaluate performance of Cachuma.

There are mainly three types of dynamic pages generated at this Web site: page `/default.asp` is the front page and shows a summary of major news from CNN and others on this virus; page `/forumchat.asp` displays a summary of all messages posted to the online forum; and each individual page `/chat/default.asp?M=msg-id` shows the content of a particular message together with a summary of all posted messages. We have collected a real trace from www.melissavirus.com on April 3, 1999. This one-day trace has a total of 53508 requests, among which 4612 are for dynamic contents. The rest of the requests are mainly for fetching embedded images. From this trace we observed that the number of front page changes is 29 while the number of front page accesses is 3063; the number of new messages posted is 106 while the number of message accesses is 1413.

To simulate such a Web site with Cachuma, we specify the following three cachable classes in a configuration file: /default.asp, /forumchat.asp, and /chat/default.asp. The cached front page needs to be invalidated only when new information on this virus is added to it. We use the invalidation message "Invalidate-Page:

/default.asp" to invalidate the front page. The forum summary page and individual message pages are changed frequently because once a new message is posted to the discussion forum, the content of these pages changes. We use URL class `/chat/default.asp` to invalidate all the message pages and use "Invalidate-Page: /forumchat.asp" to invalidate the forum summary page.

We use a client simulator which forks a new thread for each request and issues requests to the server by following the time intervals recorded in the one-day trace but linearly scaled to a two hour period. The dynamic pages are generated by a script with an average processing time of 0.5 second for each request[2]. Table 1 reports the number of fresh cache hits, fresh cache hit ratios and the average response times for the four different strategies based on the average of three runs. As this table shows, dynamic content caching leads to a cache hit ratio of 68%. Selective precomputing further increases the cache hit ratio by 16% and instant precomputing increases it by 20%. In terms of average response times, selective precomputing achieves a speedup of 3.5 compared to the no-caching case. Notice that though instant precomputing leads to a higher cache hit ratio than selective precomputing, selective precomputing outperforms it by 22% in terms of response time. This is because instant precomputing consumes too much server resource during load peaks and slows down user request processing significantly.

Strategy	Number of fresh cache hits	Fresh cache hit ratio	Avg CGI resp time	Avg HTML resp time
No caching	N/A	N/A	1375 msec	67 msec
Caching, lazy invalidation	3148	68%	557 msec	58 msec
Instant precomp.	4047	88%	505 msec	53 msec
Selective precomp.	3874	84%	391 msec	45 msec
Selective precomp. (75%)	3781	82%	404 msec	43 msec
Selective precomp. (50%)	3695	80%	418 msec	43 msec

Table 1: Test results for the Melissa Virus trace.

The last two rows of Table 1 show what will happen if we impose space restriction on the CP-set and only maintain links from URL class "/chat/default.asp" to 75% and 50% of the cached message pages (only most recently accessed pages are kept). Memory restriction limits the number of pages that can be precomputed. However, the test shows that the response time does not change much (3.5% degradation for keeping 75% of pages while 6.9% degradation for keeping 50% of pages). This is because precomputing pages which are not recently accessed does not lead to much performance gain for this case.

5.2 Customized News Service

At many Web sites such as www.MyCNN.com, online pages can be customized by clients to suit their own interests. We wrote a simple customized news application which classifies headline news from each country based on topics such as weather, politics, and sports, and generates dynamic pages for users based on their profiles. Dynamic pages are generated using a script: `/cgi-bin/news` with two arguments: `topic=some_topic` and `country=some_country`. Initially, we specify that the URL class `/cgi-bin/news` is cachable but the URL class `/cgi-bin/news?topics=stock_quote` is non-cachable. When the content

[2]The data is based on a measurement of the message retrieval time from www.melissavirus.com. The average response time (not counting roundtrip time) is about 0.54 sec when requests are issued one by one.

related to a topic (such as sports) changes, all pages which contain related headlines (e.g. URL class: `/cgi-bin/news?topic=sports`) are invalidated.

To model client access patterns at a customized news site, we have followed the state transition graph approach used in [22]. Figure 7 shows our client behavior model. A client moves between two states: SLEEP and STAY. It moves from SLEEP state to STAY state by issuing a new request. During the STAY state, the client may automatically reload the same page every T_{RE} seconds, or with a probability p, the client may manually refresh this page before auto-reloading occurs. When the client completes the action in the STAY state, he/she moves to SLEEP state. The above client behaviors mimic the access patterns at Web sites such as www.MyCNN.com, which generates customized news pages with a meta field `<META HTTP-EQUIV=''refresh'', CONTENT=''num_seconds''>` to enable automatic page reloading in every "num_seconds".

Let T_{ST} be the average time in STAY state and T_{SL} be the average time in SLEEP state by a client. Given N clients and the above client behavior model, the following formula estimates λ, the rate at which page loading requests are sent by the N clients.

$$\lambda = \frac{N}{(p \times T_{RE}/2 + (1-p) \times T_{RE})} \times \frac{T_{ST}}{T_{ST} + T_{SL}}.$$

Notice that "$p \times T_{RE}/2 + (1-p) \times T_{RE}$" is the average waiting time for a user to launch a new refresh request when he/she is in the STAY state.

In order to select reasonable parameters for our tests, we first examine a possible scenario at MyCNN. MyCNN sets page refresh time T_{RE} to 90 minutes. Assume clients have a stay and sleep time ratio of $2:1$, i.e. $T_{ST}/(T_{ST} + T_{SL}) = 0.5$. And assume MyCNN has $N = 1$ million active users on average and that $p = 0.3$. Then it will see a client request rate of $\lambda = 145$/second. Further assume that each news request takes 0.5 seconds processing time on average [3] and there is a cluster of 100 nodes for processing user requests. This gives a processing ratio of $\mu = 200/sec$. As a result, MyCNN will have a server utilization of $\rho = \lambda/\mu = 72.5\%$. In our experiment with one host machine, the script simulating customized news generates pages at a rate of $\mu = 2/sec$. To design a test with a load situation similar to the above MyCNN scenario, we have scaled down the request activities by setting $T_{RE} = 30$ seconds, $T_{ST} = 120$ seconds, $T_{SL} = 60$ seconds, and $p = 30\%$. The numbers of clients are $40, 60, 80$, and 160 respectively. If $N = 60$, then $\lambda = 1.57$ and the server utilization is $\rho = \lambda/\mu = 78.5\%$. We can see that our load design reaches the same utilization level as the MyCNN case.

In our experiment, we assume that there are 10 different news topics and each topic has 100 different countries to choose from. We also assume that at every 30 second interval, contents related to an average of three topics may change, which means that about 30% of cached dynamic pages need to be invalidated every 30 seconds. We assume the probability that a topic changes contents follows Zipf distribution with $\alpha = 1.0$, a distribution which has been observed in many human behaviors and Internet related scenarios [6, 7]. Each test run lasts 1 hour. Table 2 shows the average response times of dynamic requests seen by clients and the cache hit ratios seen by the server under different caching and precomputing strategies. The result shows that selective precomputing is very effective and can have a response time speedup of 5.5, 2.1, and 1.5 compared to the no caching, caching with lazy invalidation, and instant precomputing cases respectively

[3] This cost setting is reasonable based on the information we collected from MyCNN.com and melissavirus.com. The Web searching time is also close to this number. For example, a trace from Google.com indicates that the average response time is 0.70 second for 8.9 million searching requests.

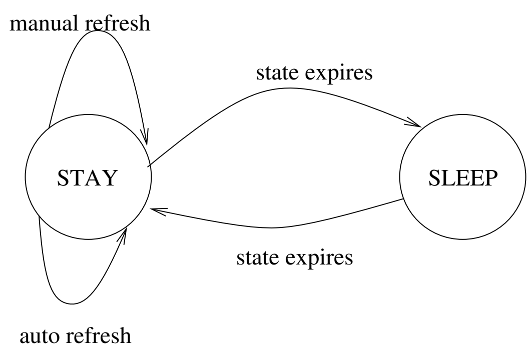


Figure 7: State transition diagram of clients retrieving customized news pages.

when there are about 160 concurrent client accesses. We also observe that more performance improvement is obtained when we increase the number of concurrent client accesses.

Number of clients	40		60		80		160	
	resp time	hit ratio	resp time	hit ratio	resp time	hit ratio	resp time	hit ratio
No caching	840	N/A	1130	N/A	1570	N/A	8340	N/A
Caching, lazy invalidation	570	31%	850	35%	1100	42%	3250	58%
Instant precomp.	350	83%	720	80%	980	78%	2270	84%
Selective precomp.	320	75%	590	77%	740	77%	1520	80%

Table 2: Average response time(milliseconds) and fresh cache hit ratios for a news service.

To study impact of precomputing on server CPU load, we also recorded server CPU utilization for every 12 seconds during the one hour test period for each strategy. Figure 8(a) shows server CPU utilization when serving 60 clients with caching only scheme and Figure 8(b) for selective computing case. We consider that server load is low when the CPU utilization is below 30% while server load is high when utilization is above 90%. It can be observed that server CPU utilization exceeds high load water-mark more frequently in Figure 8(a) than in Figure 8(b). This shows that selective precomputing can not only increase cache hit ratio but also smoothen burstiness of server resource demands by averaging out server loads. To further validate this statement, in Figure 8(c), we sort the probing points in an increasing order of the corresponding CPU utilization and mark the percentage of points below the low load water-mark and above the high load water-mark. The results show that the percentage of points that exceed high server load water-marks reduces from 16% to 10% by using selective precomputing. Without precomputing, 40% of time the server is under utilization while selective precomputing reduces this number to 22%.

5.3 Online Auction

Typically, pages at an auction site such as eBay can be categorized as: 1) **main page**: It lists all the item categories and the number of bidding items in each category. 2) **category pages**: For each category, there is a page(or possibly broken into multiple pages) listing all the bidding items in that category, and their current bidding status(end time, current price, etc). 3) **bid form**: This page is for users to view detailed information on an item and submit new prices for the item. When a user bids on an item, price change for that item should be reflected in its category page. I.e. the category page containing this item should be

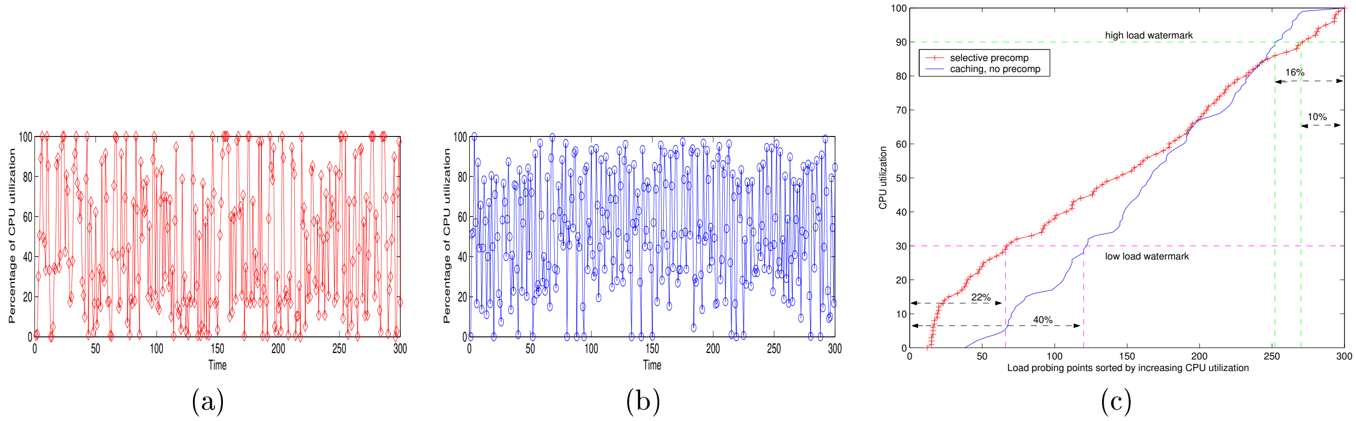


Figure 8: Comparison of server CPU utilization when accessed by 60 clients. (a)CPU utilization for the caching-only case. (b) CPU utilization for selective precomputing. (c)Sorted CPU utilization for both cases.

invalidated if cached. 4) **item-add form**: This page is for users to add new items. When a new item is added, the main page as well as the category page containing this item may also change.

According to data published in the eBay web site (www.ebay.com), eBay currently receives 50 million hits per day, among which about 800,000 are bidding requests and 250,000 are new item posting requests. If we do not count the requests for embedded images, then among all the requests for eBay pages, about 10% requests are for bidding, and 3% requests are for adding new items. To model user bidding patterns, we have collected the numbers of daily bids on featured items at eBay from May 28, 1999 to June 8, 1999 and have observed that for items whose bidding period will end in a day, the numbers of bids on them ordered from "hot" to "cold" (i.e. receiving least bids) also follow Zipf distribution [33]. Figure 9(a) shows a typical distribution on June 1. 1999, which follows Zipf distribution with $\alpha = 0.3$. The X axis shows item ranks in logarithmic scale based on number of bids received. The Y axis shows the number of bids received in logarithmic scale.

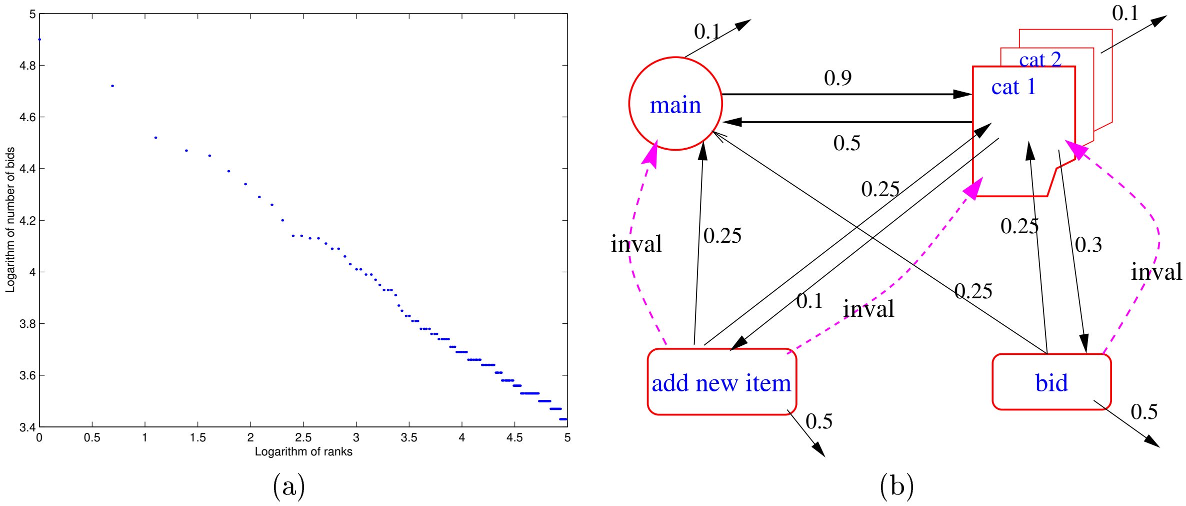


Figure 9: (a) Log-Log distribution of numbers of user bids on featured items at eBay. (b) User transition patterns during our on-line auction test and content dependence of dynamic pages.

We have developed a prototype application which models activities at on-line auction sites. In our setting, main page is generated by script /main.cgi, category pages are generated by script /items.cgi with category name as an argument, and bid forms are generated by script /bid.cgi with item id as the argument. These pages are pre-registered as cachable with three URL classes: "URL-class: /main.cgi", "URL-class: /items.cgi" and "URL-class: /bid.cgi". When the bidding price of an item changes, the dynamic page for the corresponding category is invalidated. When a new item is added, both the main front page and the corresponding category page are invalidated. Figure 9(b) illustrates possible user browsing movement among these pages and invalidation scenarios. The values attached to transition edges are the assumed transition probabilities in our experiments. We view the state transitions as a Markov Process and our calculation of stable state probability shows that such a transition graph gives us 10.7% requests for bidding and 3.5% requests for adding new items among all the page access requests. This setting is consistent to the statistics of eBay mentioned above. We use mSQL database system (available at www.hughes.com.au) as the bidding management backend. The database is initialized with 1000 bidding items belonging to 100 categories before each test. As is observed at eBay, Zipf distribution is used in our tests to model user bidding patterns on listed items.

Table 3 shows the test results we obtained for dynamic pages when 40, 80, and 160 concurrent clients were used to drive the tests for each 2 hour test period. Each client issues about 200 dynamic content requests during each test. Table 3 shows that selective precomputing can result in 84% fresh cache hit ratio, and yields a 7.6-fold response time speedup compared to the no-caching case and 1.9-fold compared to lazy invalidation. Improvement increases substantially as more concurrent clients access the server.

# clients	40		80		160	
	avg. resp	hit ratio	avg. resp	hit ratio	avg. resp	hit ratio
No caching	810	N/A	1230	N/A	9860	N/A
Caching, lazy inval.	220	67%	310	65%	2450	70%
Selective precomputing	170	82%	260	82%	1280	84%

Table 3: Request response times in milliseconds and hit ratios in on-line auction.

5.4 Overhead for Searching URL Classes

Because there are few URL classes involved in the tested applications discussed above, we have conducted more experiments to quantify the performance of our data structure in the presence of large numbers of URL classes. Trie searching has already been shown very fast [1], therefore, we only examine the time spent in a parameter classifier for locating matching URL classes. Table 4 shows the measured results for different test settings. The first row lists the number of parameter groups in a classifier. The second row lists the average number of URL classes covered by each group. The third row lists measured classifier searching cost per request. We also measured the cost when linearly comparing a given page to all URL classes and the results are show in the fourth row. We have only tested up to 100 parameter groups per classifier because in practice the number of parameter names used in an application for invalidation is usually limited. This table shows that the proposed data structure results in very fast URL class searching. And the search time is fairly independent of the number of URL classes in each parameter condition group, which is an advantage of our data structure. The 0.126 millisecond overhead for the 1 parameter group and 10 classes per group case actually represents the overhead for our customized news service test case where 10 URL classes and 1 parameter group were used in invalidation.

There are two points we want to emphasize. 1) The overhead we measured in the tests is the overhead for locating matching classes for a page, which requires linearly scanning all parameter groups and performing a hash lookup for each matching group. The overhead for locating minimal URL classes for a page would be less since searching of some groups can be avoided. 2) Our data structure is necessary to achieve low-overhead searching. If linear comparison is used instead, The cost can exceed 624 milliseconds for a large number of URL classes.

# param. groups/classifier	1			10			100		
# URL classes per group	10	100	1000	10	100	1000	10	100	1000
Search overhead per request	0.126	0.127	0.126	0.179	0.177	0.174	0.235	0.224	0.225
Cost of linear comparison per request	0.17	0.562	4.564	0.645	5.4	59.0	5.4	58.7	624.2

Table 4: Overhead for searching URL classes in a parameter classifier (milliseconds).

6 Related Work

Extensive research has been conducted for caching static pages on proxies [8, 10, 12, 20, 21, 29, 30]. For some applications, dynamic pages are stable for some period and can also be cached by proxies using application-specific rules [9, 26]. These studies do not address invalidation of dynamic pages in details. It is more appropriate to cache dynamic pages at sever sites because of their strong consistency requirement and the difficulty to invalidate cached pages on proxies which are beyond control of a Web site [10, 17, 18, 27]. To invalidate dynamic pages, Holmedehl et al. [17] uses the standard time-to-live (TTL) technique. Vahdat et al. [27] proposes techniques that intercept file system calls and monitor change of source files. These techniques are only effective for limited applications.

Iyengar et al. [18, 10] propose a fine-grain dependence based approach that allows individual applications to explicitly cache application-specific contents and invalidate them when the data they depend on change. Our class based approach is closely related to this work and a detailed comparison between their approach and ours is given below.

- **Expressiveness and manageability**.

  Dependence. The class-based approach allows dependence information for individual pages that share same patterns to be defined collectively. This makes dependence specification more modular and easy to manage. The fine-grain dependence approach requires dependence among individual pages and data objects (e.g. database tables) to be specified explicitly. Since the caching system needs to collect dependence information from an application, communication between an application and a cache is simpler and more efficient for the class-based approach.

  Invalidation. In the class-based approach, cached pages which share the same patterns can be invalidated using a short message. This is especially useful for invalidating pages matching unpredictable URL classes. In the fine-grain dependence approach, an application notifies the cache manager of object change, which triggers invalidation of pages that depend on this object. But If dependence on this object is unknown in advance, the application has to invalidate affected pages one by one.

  Identification. The class-based approach also allows information other that URLs to be used in page identifications. This feature is necessary for many applications but is not available in IBM's fine grain dependence-based approach.

- **Size of control information**.

  We have discussed how sizes of CP-set and I-set are controlled in our class-based approach. The fine-grain approach requires maintenance of the entire dependency graph for all individual pages. Its size could grow very large if a Web site hosts a large number of dynamic pages (e.g. personalized pages based on user profiles) and part of the entire dependence graph may need to be swapped out to the second storage [18, 10]. One advantage of building the entire dependence graph is that stale pages can be identified immediately after an invalidation message in received, which is needed for page precomputing. We achieve the same feature in the class-based approach by maintaining a mapping from predictable URL classes to cached pages.

Another difference between our work and the work in [10] is in precomputing. the work in [10] uses instant precomputing while we use selective precomputing. Selective precomputing can be viewed as a technique orthogonal to server clustering with load balancing [2, 11, 23, 32] because server clustering distributes and balances load spatially while selective precomputing tries to distribute and balance load temporally.

A disadvantage of server site dynamic content caching is that every request has to go to the origin server, even though a request can use "If-Modified-Since" header to query a Web server. Recently, new algorithms and infrastructures for proxy caching have been proposed for efficiently enforcing consistency over the Internet. These include invalidation message multicasting [21, 29, 30] and adaptive lease [13]. These works do not explicitly address dynamic content invalidation. Our proposed work can be extended and integrated with these techniques for distributed dynamic content caching.

URL patterns and client information have been used in proxy caching for defining page equivalences [26, 4]. But either the patterns do not consider parameters [4] or the proposed data structure does not allow URL patterns with same netpaths to have different parameter sets [26].

7 Concluding Remarks

This paper proposes a URL class based page invalidation scheme. The scheme enables flexible and efficient content caching for Web sites hosting an arbitrarily large number of dynamic pages and avoids fine-grain level cache manipulation. Lazy invalidation works under space constraint and avoids slow disk accesses while selective precomputing averages out server load peaks and increases cache hit ratios. The proposed data structure using tries and parameter classifiers allows for fast searching of URL classes. Our caching system integrates the proposed techniques and is implemented on top of standard Web servers without modifying server source code.

Instant precomputing has better caching hit ratios than selective precomputing, however, it aggravates resource contention during load peaks and leads to higher response times. Thus load smoothening benefit of dynamic page prefetching outweighs caching benefit and selective precomputing strikes a necessary balance for both smoothened server load and higher cache hit ratios.

The experimental results in [18, 10, 17] show that dynamic content caching at a server site can lead to about 30% improvement in cache hit ratios. The average response time reduction reported in [17] is also about

30%. Our results indicate performance impact of caching on response times can be much more substantial, especially during load peaks. For example, response times decrease from 8.34 to 1.52 seconds in a customized news test case and from 9.86 to 1.28 seconds (i.e. a 7.6-fold speedup) in an auction case, which is substantial even considering Internet delay. We expect more improvements can be observed if page generation requires intensive resources such as digital libraries [32, 17] or if our techniques are properly integrated with proxy caching. Since performance enhancement is significant normally during load peaks, dynamic content caching is most useful for busy Web sites.

References

[1] A. Acharya, H. Zhu, and K. Shen. Adaptive Algorithms for Cache-efficient Trie Search. In *ACM and SIAM Workshop on Algorithm Engineering and Experimentation*, Jan. 1999.

[2] D. Andresen, T. Yang, V. Holmedahl, and O. H. Ibarra. SWEB: Towards a Scalable WWW Server on Multi-Computers. In *Proceedings of the IEEE International Symposium on Parallel Processing*, pages 850–856, Apr. 1996.

[3] D. Andresen *et al.* The WWW Prototype of the Alexandria Digital Library. In *Proceedings of ISDL'95: International Symposium on Digital Libraries*, Aug. 1995. Revised version appeared in IEEE Computer, 1996, No.5.

[4] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the USENIX Technical Conference*, Jan. 1997.

[5] G. Banga and J. Mogul. Scalable Kernel performance for Internet servers under realistic load. In *Proceedings of USENIX Annual Technical Conference*, 1998.

[6] P. Barford and M. E. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proc. of ACM SIGMETRICS'98*, 1998.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE Infocom'99*, Mar. 1999.

[8] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[9] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, Mar. 1998.

[10] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable system for Consistently Caching Dynamic Web Data. In *Proceedings of the IEEE INFOCOM'99*, Mar. 1999.

[11] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of Task Assignment Policies in Scalable Distributed Web-Server Systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 585–598, June 1998.

[12] F. Douglis, A. Haro, and M. Rabinovich. HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[13] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of IEEE Infocom 2000*, Mar. 2000.

[14] L. Fan, P. Cao, and Q. Jacobson. Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance. In *Proc. of ACM SIGMETRICS'99*, May 1999.

[15] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Oct. 1997.

[16] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of 1996 USENIX Annual Technical Conference*, Jan. 1996.

[17] V. Holmedahl, B. Smith, and T. Yang. Cooperative Caching of Dynamic Content on a Distributed Web Server. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[18] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[19] D. Knuth. *The Art of Computer Programming. volume 3: Sorting and Searching.* Addison-Wesley, 1973.

[20] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[21] D. Li and D. R. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, pages 92–103, Oct. 1999.

[22] D. Menasce, V. Almeida, R. Fonseca, and M. A. Mendes. Resource Management Policies for E-Commerce Servers. In *Second Workshop on Internet Server Performance*, May 1999.

[23] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Service. In *Proceedings of ASPLOS-VIII*, Oct. 1998.

[24] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99),*, Feb. 1999.

[25] R. Rivest. *The MD5 Message-Digest Algorithm.* http://www.ietf.org/rfc/rfc1321.txt, Apr. 1992.

[26] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploiting Result Equivalence in Caching Dynamic Web Content. In *Proceedings of Second USENIX Symposium on Internet Technologies and Systems(USITS99)*, Oct. 1999.

[27] A. Vahdat and T. Anderson. Transparent Result Caching. In *Proc. of 1998 USENIX Technical Conference*, 1998.

[28] D. Wessels. *Squid Internet Object Cache.* http://squid.nlanr.net, 1995.

[29] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in WAN. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, Oct. 1999.

[30] H. Yu and L. Breslau. A Scalable Web Cache Consistency Architecture. In *Proc. of ACM SIGCOMM'99*, Sept. 1999.

[31] H. Zhu, B. Smith, and T. Yang. Scheduling Optimization for Resource-Intensive Web Requests on Server Clusters. In *the Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, June 1999.

[32] H. Zhu, T. Yang, Q. Zheng, D. Watson, O. H. Ibarra, and T. Smith. Adaptive load sharing for clustered digital library servers. *Proceedings of the Seventh High Performance Distributed Computing*, July 1998.

[33] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Cambridge, MA, 1949.