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. This paper proposes a complementary solution for applications that require coarse-grain cache management. The key idea is to partition dynamic pages into classes based on URL patterns so that an application can specify page identification 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 IDs of dynamic pages are stored in memory with a digest format. Selective precomputing is further proposed to refresh 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 system 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 are effective in supporting coarse-grain cache management and reducing server response times for tested applications.**

I. Introduction

Dynamic page generation based on user profiles and request parameters has become popular at many Web sites. Previous work on Web caching and invalidation [1], [2] 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 since it is easy for a Web site to control its cache; 3) it also helps proxies to cache dynamic pages [3], [4] 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 on the IBM Olympics Web server [5], [6] and on the Alexandria Digital Library system [7] demonstrated the feasibility and significance of dynamic content caching.

A major problem with dynamic content caching is to ensure that strong consistency be maintained between cached pages and underlying data objects. Vahdat et al. [8] and Holmedehl et al. [7] have proposed invalidation techniques using file operation interception or TTL for a class of applications. Iyengar et al. [5], [6] proposes a general approach which lets applications explicitly issue invalidation messages to a cache. While this approach is powerful for many Web applications (for example, the Olympics Web site), it relies on maintaining a fine-grain graph that specifies 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.

We propose an approach that complements the fine-grain method and allows users to specify coarse-grain dependence among underlying datasets and groups of dynamic pages called URL classes which share common URL patterns or client information. As a consequence, application programmers do not need to enumerate data dependences 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 IDs 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 this pattern becomes difficult without in-memory knowledge of page IDs. We use a lazy invalidation strategy that accumulates invalidation messages and only performs necessary page invalidation when a client accesses a page. Second, to support dynamic page prefetching for increasing cache hit ratios, we further study selective page precomputing which 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 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 II describes class-based caching specification and invalidation. Section III presents lazy page invalidation and selective precomputing. Section IV discusses interactions among the server, cache and applications and presents an efficient data structure for searching URL classes. Section V describes our simulation and experimental results. Section VI is related work. Section VII concludes the paper.

II. URL Classes and Cache Control Specification

A. URL Classes

We group Web pages into classes based on page URLs and client information. Page grouping allows applications to cache and invalidate pages on 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 applications that generate pages, application names, and parameters. 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. In the specification, the keyword "**_pathinfo**" identifies the possible extra path information following an application name. And the keyword "**_Cookie:**$< ID >$" identifies cookie "$< ID >$" appearing in a request header. Some examples of URL classes and their meanings are as follows. "`/cgi-bin`" covers all dynamic pages which are generated by applications with /cgi-bin as their net path prefix. Class "`/cgi-bin/news?topic=sports&country=USA`" groups pages which are generated by application `/cgi-bin/news` with parameters `topics=sports` and `country=USA`. This application may have other parameters, but they are not used by this class.

Before explaining how an application specifies page cachability and invalidation through URL classes, 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. In case of specification conflicts from multiple URL classes on page cachability and dependence, we will use a concept called *minimal* class which is defined as follows to resolve such conflicts. 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 shows the relationship of four URL classes. 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.

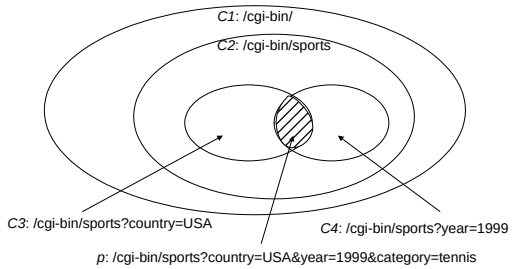


Fig. 2. Four URL classes and one page URL are listed. There are two minimal classes for this page URL.

B. Class-based Cache Control Specification

We use URL classes to manage the three aspects of page caching: (i) page cachability and dependence specification, (ii) 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 a set of cachable URL classes. Subclasses of cachable classes may be registered as non-cachable, and vice versa. Page identification specifies additional information (besides URL) needed to differentiate cached pages. The additional information includes cookies and client IP addresses.

Figure 3 illustrates cachability and object dependence specification for two URL classes. The first example specifies that pages matching URL class `/cgi-bin/news?topic=stock` are non-cachable. The second example says that pages matching `/cgi-bin/news` are cachable and they should be invalidated when data object "news-table" changes. Note that here "news-table" is just an identifier which 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
```

Fig. 3. Cache specification for two classes.

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 at 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. For example: "`Object-Change: news-table`". When such a message arrives, all pages *matching* URL classes depending on object `news-table` are invalidated.
- **Invalidate-Class:** direct invalidation of a URL class. For example: "`Invalidate-Class: /cgi-bin/news?topics=sports`". All 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 [5] uses object changes to trigger invalidation of individual pages. Our scheme expands that work by allowing class-based page invalidation. One thing covered in [5] is that pages may be invalidated due to dependence between cached pages. Since the authors in [5] 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.

III. Page Invalidation and Precomputing Techniques

When a URL class is to be invalidated, we need to locate all the cached fresh pages matching the URL class. If the cache system is "aware" of this URL class beforehand, it can maintain links between this class and its matching pages to speed up page searching. Otherwise the process of locating all pages that

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

Fig. 1. The BNF specification of a URL class.

match the given class can be expensive if the class condition is complicated. For this reason, we distinguish 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 *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 the page links can become expensive. Moreover, there could be an efficiency concern if, when a page is created, we have to find all of its matching URL classes.

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 selective precomputing.

A. Lazy Page Invalidation for Unpredictable URL Classes

When an invalidation message targeting an unpredictable URL class arrives, we need to compare the IDs of cached pages with this URL class. For efficiency, page IDs (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 [9]) 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 [10] computed from page URLs as page IDs and the space saving by using MD5 could be tremendous because the length of a URL can be several hundred bytes. We also use the same scheme to save memory space while storing the actual page IDs on disks. Consequently, actual page IDs are not readily available in memory for comparison with an unpredictable URL class because the inverse of an MD5 digest is not tractable.

To avoid retrieving page IDs from disk when an invalidation message is received, we use a lazy invalidation approach: *invalidation is not performed until a cached page is accessed*. Lazy invalidation works as follows:

1. We maintain a set called *I-set* which stores invalidated URL classes annotated with their latest invalidation time. When an invalidation message arrives, if it is of type "Invalidate-Page", the targeted page is located and invalidated immediately. If it is of type "Invalidate-Class", the targeted URL class is recorded in the I-set and marked with the new invalidation time. If it is of type "Object-Change", all the URL classes depending on this object are recorded in the I-set. Note that only the latest invalidation time needs to be maintained for each invalidated URL class.
2. When a fresh cache entry is found for a dynamic page request, the I-set is searched 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 minimizing such overhead in Section IV-B. The size of the I-set will not grow too big in practice because for invalidation messages targeting the same URL class, we only need to keep the latest copy. 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.

B. Page precomputing for predictable URL classes

As discussed above, pages matching predictable URL classes can be invalidated immediately by maintaining class-page links between predictable classes and their matching pages. We call this page invalidation process *eager invalidation* and use the term *CP-set* to refer to the set of class-page links. Eager invalidation reduces lazy invalidation overhead, but its major advantage is to allow us to perform page prefetching. Prefetching static pages has been studied in previous research [11], [12]. 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 substantial cache hit ratio improvement was observed [5]. As we show in Section III-C, by prefetching stale but popular 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*. Notice that lazy invalidation cannot be used for recomputing stale pages because the cache is not aware if a page is stale until this page is accessed by some client.

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. URL classes used in cachability and dependence specification are also considered predictable and are maintained in the CP-set.
2. To reduce space usage, the links to a page are 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 allows us to locate those subclasses efficiently. The system also enforces a specified memory limit for the 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 IV-B, our data structure allows fast searching of URL classes and this checkup overhead is negligible.

C. Instant precomputing vs. selective precomputing

In [5], dynamic page precomputing is conducted whenever a page becomes stale. We call this *instant precomputing*. Instant precomputing is effective for a Web site in which each dynamic page is frequently accessed. If a large number of dynamic pages are not accessed frequently and server computing resource is limited, delaying precomputation for these pages can reduce resource contention. 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.* If selective precomputing does not precompute a page and a client needs it, the system is forced to generate this page and deliver it on demand.

We define our composite load index as follows. 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. Let H be load threshold (we use $H = 40\%$ in our tests). 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. When a URL-class is registered, its resource demands can also be specified by "`Resource:<machine-name>, <weight>` " where `<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 CPU.

We have verified the effectiveness of the above scheme in the Alexandria digital library server cluster and a few other content-generation applications [13]. It should be noted that this scheme only monitors the load of one machine which generates a dynamic page. An extension is needed to consider a complicated backend system where each request transaction involves multiple distributed machines.

In our system implementation, we use the *LRU principle* to select popular pages for precomputing: stale pages with latest-access-times are selected for precomputing. To avoid flooding the server with excessive precomputing requests, we issue precomputing requests to the server sequentially during our experiments. A more aggressive scheme can be adopted if it does not create resource contention.

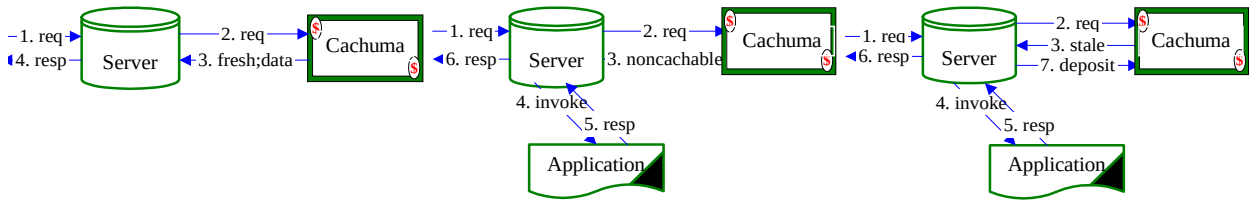
IV. Caching System Design and Implementation

A. System Architecture

Our goal is to add caching capability to existing commodity Web servers without modifying server source code. Thus our dynamic page caching system, which is named *Cachuma*, is designed and implemented as a component stand-alone from Web servers. We have successfully integrated Cachuma with Apache server v1.3, Netscape Enterprise Server v3.6, and the Swala server [7], based on APIs of those servers. The functionalities of Cachuma include: storing cached pages, invalidating cached contents when it receives invalidation messages, 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, and a table of cache entries storing page meta-information, including MD5 digests of page IDs 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.

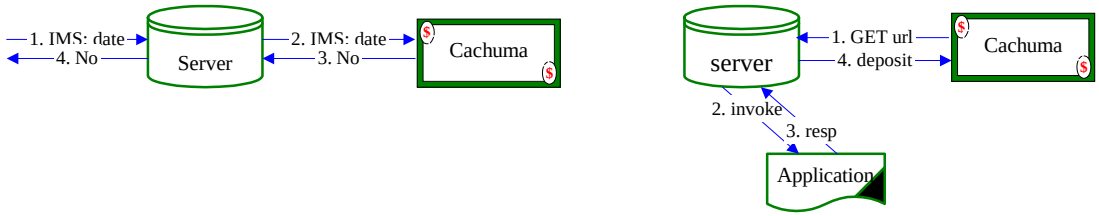
Cachuma and a Web server interact in the following three scenarios:

- **Content caching and retrieving**. There are three cases: cached page is fresh, page in non-cachable, or page is not cached or is stale. Figure 4 illustrates the steps involved for the three cases.
- **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, it 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.
- **Selective 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 invocation is negligible. Figure 5(b) demonstrates the steps for precomputing.



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

Fig. 4. Interaction between the server and the Cachuma cache when handling client requests.



(a) IMS request processing (b) Page precomputing

Fig. 5. Content validation and precomputing

B. *Data Structure for Efficient Search*

There are two fundamental URL-class searching problems in Cachuma: 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 consisting of a *trie* and a few *parameter classifiers* for both I-set and CP-set. A trie 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 parameter classifiers, as defined in next paragraph, to some leaf nodes of the trie to capture parameter conditions of URL classes with the same netpath. Trie is a well studied data structure for longest prefix match and we simply use a trie searching algorithm from our previous work which explores memory hierarchy by dynamically transferring trie nodes among small arrays, B-trees, and hash tables [14]. 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 the useful parameters 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. 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.

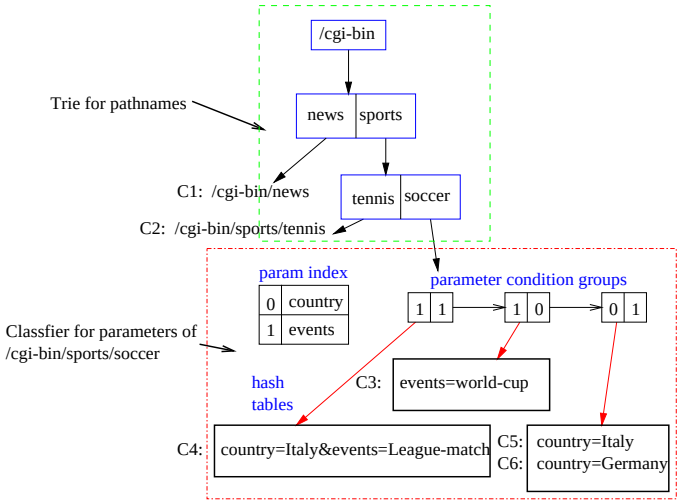


Fig. 6. Data structure for a CP-set with six URL classes.

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 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 is unnecessary 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 page `/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 bit vectors to check are $(1, 0)$ and $(0, 1)$. We skip them because these two groups cannot contain p's minimal classes.

C. 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 III-A and Section III-B have discussed the space and size control issues for these two sets. As a result, their space costs are small. To further reduce space usage, we merge multiple hash tables in parameter classifiers in the implementation.

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 that the time spent in a classifier is $O(N_g)$ and the total time for searching is $O(N_p + N_g)$.

V. 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 examine if the proposed techniques can efficiently support coarse-grain invalidation. We also study effectiveness of our data structure by examining searching overhead.

[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.

For all the tests, we run a Swala Web server, Cachuma, and the involved applications on the same machine which is a 248MHz Sun Ultra-30 running Solaris 2.6. Client requests are issued following some real traces or client behavior models from machines in a LAN. 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 which 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%).

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 them 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 [15].

A. Online Forums

We have developed an application to mimic the design and behavior of Web site `www.melissavirus.com` for online discussion of Melissa virus. 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 on this virus; page `/forumchat.asp` displays a summary of all messages posted to the 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 this site on April 3, 1999 to evaluate Cachuma. This one-day trace has a total of 53508 requests, among which 4612 are for dynamic contents.

To simulate such a Web site with Cachuma, we identify and pre-register the three cachable classes: /default.asp, /forumchat.asp, and /chat/default.asp. The cached forum summary page and individual message pages are invalidated once a new message is posted to the discussion forum. We use URL class /chat/default.asp to invalidate all the message pages.

Table I reports our measured results for the four different strategies based on the average of three runs. The table shows that 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. 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

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 precomputing	4047	88%	505 msec	53 msec
Selective precomputing	3874	84%	391 msec	45 msec
Selective precomputing (75%)	3781	82%	404 msec	43 msec
Selective precomputing (50%)	3695	80%	418 msec	43 msec

TABLE I
TEST RESULTS FOR THE MELISSA VIRUS TRACE.

down processing normal user requests significantly.

The last two rows of Table I 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.

B. 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 to simulate such sites. Dynamic pages are generated by script: `/cgi-bin/news` with two arguments: `topic=some_topic` and `country=some_country`. The URL class `/cgi-bin/news` is registered cachable but the URL class `/cgi-bin/news?topic=stock` is registered non-cachable. When the content related to a topic (such as sports) changes, all cached pages which contain related headlines (i.e. 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 [15]. 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 probability p, the client may manually refresh this page before auto-reloading occurs. The above client behaviors mimic the access patterns at Web sites such as www.MyCNN.com, which generate 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 second processing time on average [2] 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.

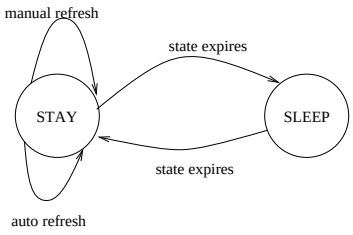


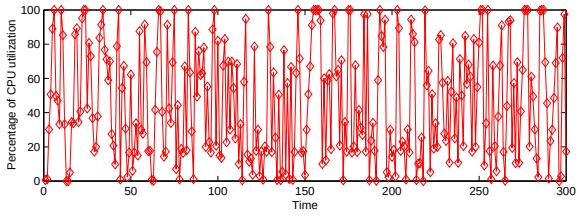
Fig. 7. State transition diagram of clients retrieving customized news pages.

In our experiment, we assume 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 change, meaning 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 [16], [17]. Each test run lasts 1 hour. Table II shows

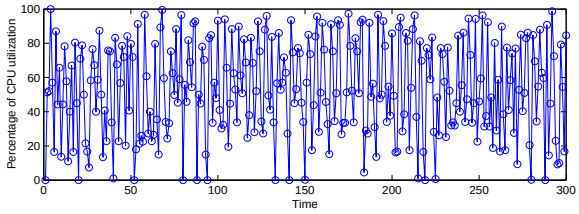
[2]This cost setting is reasonable based on the information we collected from MyCNN.com.

the measured results under different caching and precomputing strategies. The results show that selective precomputing is very effective and has 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. In this case, there are about 160 concurrent client accesses. We also observed more performance improvement when we increased the number of concurrent client accesses.

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) (16% vs. 10% of the measured periods). 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.



(a) CPU utilization for instant computing.



(b) CPU utilization for selective precomputing.

Fig. 8. Comparison of CPU utilization.

C. 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 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 receives 50 million hits per day in 1999, 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 [18]. Figure 9 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.

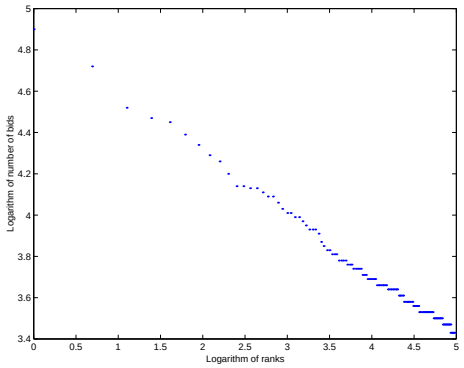


Fig. 9. Log-Log distribution of numbers of user bids on featured items at eBay.

We have developed a prototype application which models activities at on-line auction sites. In our setting, the 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 10 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 calculation of stable state probability shows that such a transition graph gives 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 to model user bidding patterns on listed items.

Table III 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 III shows that selective precomputing can result in 84% fresh cache hit ratio,

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 precomputing	350	83%	720	80%	980	78%	2270	84%
Selective precomputing	320	75%	590	77%	740	77%	1520	80%

TABLE II
AVERAGE RESPONSE TIME(MILLISECONDS) AND FRESH CACHE HIT RATIOS FOR A NEWS SERVICE.

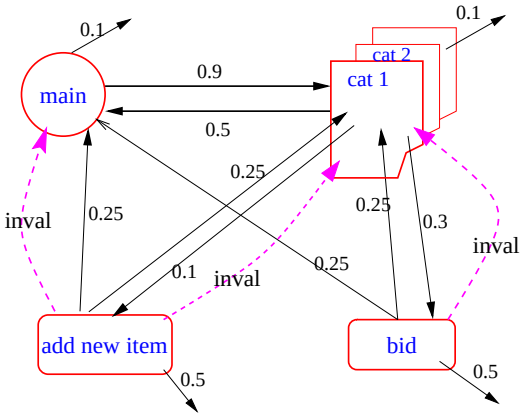


Fig. 10. User transition patterns during on-line auction and content dependence of dynamic pages.

and yield 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.

D. 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 [14], therefore, we only examine the time spent in a parameter classifier for locating matching URL classes. The third row of Table IV lists measured per-request classifier searching cost with our data structure. We have also measured the cost of linearly comparing a given page to all URL classes and the results are shown 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 our scheme outperforms linear comparison substantially. The search time is fairly independent of the number of URL classes in each parameter group, which is an advantage of our data structure. The 0.126 millisecond overhead for the 1 parameter group and 10 classes/group case actually represents the overhead for our customized news service test where 10 URL classes and 1 parameter group were used in invalidation.

VI. RELATED WORK

Extensive research has been conducted for caching static pages on proxies [1], [19], [12], [20], [21]. For some applications, dynamic pages are stable for some period and can also be cached by proxies using application-specific rules [3], [4] which can exploit URL patterns. 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 [5], [7], [6], [8]. To invalidate dynamic pages, Holmedehl et al. [7] uses the standard time-to-live (TTL) technique. Vahdat et al. [8] proposes to intercept file system calls for monitoring source file changes. These techniques are only effective for limited applications.

Iyengar et al. [6], [5] propose a fine-grain dependence based approach that allows individual applications to explicitly cache dynamic contents and invalidate them. Our class-based approach complements this fine-grain approach by enhancing expressiveness and manageability to support group-oriented dependence specification and page invalidation. Another difference is that the work in [5] uses instant precomputing while we use selective precomputing. Selective precomputing can be viewed as a technique orthogonal to server clustering with load balancing [22], [23], [13] because server clustering distributes and balances load spatially while selective precomputing tries to distribute and balance load temporally.

VII. CONCLUDING REMARKS

This paper proposes a URL class based page invalidation scheme which enables flexible content caching for Web sites hosting an arbitrarily large number of dynamic pages. 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. Cachuma integrates the proposed techniques and interoperates with standard Web servers without modifying server source code. Our future work is to extend cache management for cluster-based network services [7], [24].

Instant precomputing has better cache hit ratios than selective precomputing, however, it aggravates resource contention during load peaks and leads to higher response times, unless a Web site has sufficient computing resources. Selective precomputing strikes a necessary balance between smoothened server load and higher cache hit ratios.

It should be noted that if the content of a dynamic page

# 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 invalidation	220	67%	310	65%	2450	70%
Selective precomputing	170	82%	260	82%	1280	84%

TABLE III

REQUEST RESPONSE TIMES (MILLISECONDS) AND HIT RATIOS IN ON-LINE AUCTION.

# parameter groups per classifier	1			10			100		
# URL classes per group	10	100	1000	10	100	1000	10	100	1000
Per-request search overhead	0.126	0.127	0.126	0.179	0.177	0.174	0.235	0.224	0.225
Per-request cost of linear comparison	0.17	0.562	4.564	0.645	5.4	59.0	5.4	58.7	624.2

TABLE IV

OVERHEAD FOR SEARCHING URL CLASSES IN A PARAMETER CLASSIFIER (MILLISECONDS).

changes frequently such as stock quote, it does not make sense to use caching because data become stale too frequently and there is too much overhead to maintain consistency. Our caching solution in general fits into applications with relatively slower changing data. For such an application, a good cache hit ratio can be achieved in our coarse-grain scheme with low performance overhead. It is worthy to point out that our experimental results further confirm the importance of dynamic content caching studied in the earlier work [6], [5], [7], 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 improvement can be observed if page generation requires intensive resources such as digital libraries [13], [7] or if server-site caching is properly integrated with proxy caching. Since performance enhancement is significant during load peaks, dynamic content caching is most useful for busy Web sites.

Acknowledgment. This work was supported in part by NSF CCR-9702640, ACIR-0082666 and 0086061. We would like to thank Arun Iyengar at IBM, Hong Tang, Kai Shen, and the anonymous referees for their valuable comments.

REFERENCES

[1] Pei Cao and Sandy Irani, "Cost-Aware WWW Proxy Caching Algorithms," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[2] James Gwertzman and Margo Seltzer, "World-Wide Web Cache Consistency," in *Proceedings of 1996 USENIX Annual Technical Conference*, Jan. 1996.

[3] Pei Cao, Jin Zhang, and Kevin 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)*, Mar. 1998.

[4] Ben Smith, Anurag Acharya, Tao Yang, and Huican Zhu, "Exploiting Result Equivalence in Caching Dynamic Web Content," in *Proceedings of Second USENIX Symposium on Internet Technologies and Systems(USITS99)*, Oct. 1999.

[5] Jim Challenger, Arun Iyengar, and Paul Dantzig, "A Scalable system for Consistently Caching Dynamic Web Data," in *Proceedings of the IEEE INFOCOM'99*, Mar. 1999.

[6] Aran Iyengar and Jim Challenger, "Improving Web Server Performance by Caching Dynamic Data," in *Proc. of USENIX Symp. on Internet Technologies and Systems*, Dec. 1997.

[7] Vegard Holmedahl, Ben Smith, and Tao Yang, "Cooperative Caching of Dynamic Content on a Distributed Web Server," in *Proc. of the Seventh IEEE International Symp. on High Performance Distributed Computing*, July 1998.

[8] Amin Vahdat and Thomas Anderson, "Transparent Result Caching," in *Proc. of 1998 USENIX Technical Conference*, 1998.

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

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

[11] Li Fan, Pei Cao, and Quinn Jacobson, "Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance," in *Proc. of ACM SIGMETRICS'99*, May 1999.

[12] Thomas M. Kroeger, Darrell D. Long, and Jeffrey 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.

[13] Huican Zhu, Tao Yang, Qi Zheng, David Watson, Oscar H. Ibarra, and Terry Smith, "Adaptive load sharing for clustered digital library servers," *Proceedings of the Seventh High Performance Distributed Computing*, July 1998.

[14] Anurag Acharya, Huican Zhu, and Kai Shen, "Adaptive Algorithms for Cache-efficient Trie Search," in *ACM and SIAM Workshop on Algorithm Engineering and Experimentation*, Jan. 1999.

[15] Daniel Menasce and Vigilio Almeida, *Scaling for E-business: technologies, models, performance, and capacity planning* , Prentice Hall, 2000.

[16] Paul Barford and Mark E. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in *Proc. of ACM SIGMETRICS'98*, 1998.

[17] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," in *Proceedings of IEEE Infocom'99*, Mar. 1999.

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

[19] Fred Douglis, Antonio Haro, and Michael Rabinovich, "HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[20] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin, "Hierarchical Cache Consistency in WAN," in *Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems*, Oct. 1999.

[21] Haobo Yu and Lee Breslau, "A Scalable Web Cache Consistency Architecture," in *Proc. of ACM SIGCOMM'99*, Sept. 1999.

[22] Michele Colajanni, Philip S. Yu, and Daniel M. Dias, "Analysis of Task Assignment Policies in Scalable Distributed Web-Server Systems," *IEEE Transactions on Parallel and Distributed Systems*, pp. 585–598, June 1998.

[23] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum, "Locality-Aware Request Distribution in Cluster-based Network Service," in *Proceedings of ASPLOS-VIII*, Oct. 1998.

[24] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu, "Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services," To appear in the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01), Mar. 2001.