A Proxy-Based Approach for Dynamic Content Acceleration on the WWW

Anindya Datta, Kaushik Dutta, Helen Thomas, Debra VanderMeer
Chutney Technologies
Atlanta, GA
{anindya,kaushik,helen,deb}@chutneytech.com

Krithi Ramamritham
Indian Institute of Technology-Bombay
Powai, Mumbai, India
krithi@iitb.ac.in

Suresha
Indian Institute of Science
Bangalore, India
suresha@csa.iisc.ernet.in

Abstract

Various dynamic content caching approaches have been proposed to address the performance and scalability problems faced by many Web sites that utilize dynamic content generation applications. Proxy-based caching approaches store content at various locations outside the site infrastructure and can improve Web site performance by reducing content generation delays, firewall processing delays, and bandwidth requirements. However, existing proxy-based caching approaches either (a) cache at the page level, which does not guarantee that correct pages are served and provides very limited reusability, or (b) cache at the fragment level, which requires the use of pre-defined page layouts. To address these issues, several back end caching approaches have been proposed, including query result caching and fragment level caching. While back end approaches guarantee the correctness of results and offer the advantages of fine-grained caching, they neither address firewall delays nor reduce bandwidth requirements.

In this paper, we present an approach and an implementation of a dynamic proxy caching technique which combines the benefits of both proxy-based and back end caching approaches, yet does not suffer from their above-mentioned limitations. Our analysis of the performance of our approach indicates that it is capable of providing significant reductions in bandwidth. Experimental results from an implementation of this approach indicate that our technique is capable of providing order-of-magnitude reductions in bandwidth.

1. Introduction

To provide visitors with dynamic, interactive, and personalized experiences, web sites are increasingly relying on dynamic content generation applications, which build Web pages on the fly based on the run-time state of the Web site and the user session on the site. However, these benefits come at a cost – each request for a dynamic page requires computation as well as communication across multiple components inside the server-side infrastructure.

Caching is a widely-used approach to mitigate the performance degradations due to WWW content distribution and delivery. Here, content generated for one user is saved, and used to serve subsequent requests for the same content.

In general, there are two basic approaches: *back-end caching* and *proxy-based caching*. Back-end caches typically reside within a site, and cache at the granularity of a *fragment*, i.e., a portion of a Web page. Back-end caching solutions do not rely on URLs to identify cached content (as is the case with proxy-based solutions), and thus guarantee correctness of the contents in a generated page. However, this type of solution does not reduce the bandwidth needed to connect to the server to obtain content. In contrast, proxy-based caches typically store content at the granularity of full web pages, and reside outside the site's infrastructure. This type of caching can provide significant bandwidth savings, both in the site's infrastructure as well as on the WWW infrastructure; however, it suffers from two major drawbacks: (1) full-page dynamically generated HMTL files generally have little reusability, leading to low hit ratios; and (2) cache hits are determined based on a request's URL, which does not necessarily uniquely identify the page content, leading to the possibility of serving incorrect pages from cache.

In this paper, *we propose an approach for caching gran-*

Proceedings of the 4th IEEE Int'l Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2002)
1530-1354/02 $17.00 © 2002 **IEEE**

ular proxy-based dynamic content that combines the benefits of both approaches, while suffering the drawbacks of neither. The remainder of this paper is organized as follows. In Section 2 we provide a brief overview of dynamic content generation, and in Section 3 we discuss existing approaches to dynamic content caching. In Section 4 we describe our approach for caching granular proxy-based dynamic content. In Sections 5 we present an analysis of our approach, and in Section 6 we present experimental results which validate our analytical findings. We conclude in Section 7.

2 Dynamic Content Generation: Background and Preliminaries

Web sites are increasingly using dynamic content generation applications to serve content. At a high level, dynamic content generation works as follows. A user request maps to an invocation of a script. This script executes the necessary logic to generate the requested page, which involves contacting various resources (e.g., database systems) to retrieve, process, and format the requested content into a user deliverable HTML page.

Consider a Web site that caters to both *registered* users (i.e., users who have set up an account with the site) and *non-registered users* (i.e., occasional visitors). Suppose the site allows registered users to create a user profile, which specifies the user's content preferences and allows him to control the layout of the page. Here, pages contain a number of elements or *fragments*. For each request, the Web site lays out the fragments on a page in a specific default configuration for non-registered, and based on a user profile for registered users.

In general, an HTML page consists of two distinct components: *content* and *layout*. Content refers to the actual information displayed and layout refers to a set of markup tags that define the presentation (e.g., where the content appears on the page). Loosely speaking, the different fragments on a page represent content, whereas the layout determines how the fragments are presented on the user viewable page. Here, the final presentation of the page is partially determined by the order in which the markup tags appear in the page, as well as the actual markup tags themselves.

The foregoing discussion highlights two important characteristics of dynamically generated content. First, *not only is the content of many sites dynamic, but also the page layout*. Second, and most important, *the same request URL can produce different content and/or different layouts*.

There are several potential bottlenecks involved in serving dynamic content. These bottlenecks can be classified into two broad areas: (a) network latency, i.e., delays on the network between the user and the Web site, and (b) server latency, i.e., delays at the Web site itself. Delays at the Web server can be further classified into two categories: (1) *session processing delays*, and (2) *dynamic content generation delays*. Web server session processing delays occur as a result of the numerous devices (e.g., routers, firewalls, switches) through which requests must pass before reaching the Web server.

Content generation delays occur as a result of the work required to generate a Web page. Due to the complexity of modern Web site application layers, sites are increasingly employing a layered or *n-tier* application architecture, which partitions the application into multiple layers. For instance, the *presentation layer* is responsible for the display of information to users and includes formatting and transformation tasks. Presentation layer tasks are typically handled by dynamic scripts (e.g., ASP, JSP). The *business logic layer* is responsible for executing the business logic, and is typically implemented using component technologies such as Enterprise Java Beans (EJB). The *data access layer* is responsible for enabling connectivity to back-end system resources (e.g., DBMSs), and is typically provided by standard interfaces such as JDBC or ODBC.

This process can incur several types of delays, including computational delays (e.g., query processing). interaction bottlenecks (e.g., waiting for a DBMS connection), cross-tier communication (e.g., TCP/IP stack traversal), object creation and destruction, and content conversion (e.g., XML-to-HTML transformations). Each of these content generation delays contributes to the end-to-end latency in delivering a Web page. As user load on a site increases, the site infrastructure is often unable to serve requests fast enough. The end result is increased response times for end users.

3 Existing Approaches and Their Limitations

A widely used existing approach to address WWW performance problems is based on the notion of content caching. These caching approaches can be classified as *proxy-based* and *back end* caching solutions.

Proxy-based caching approaches are based on caching content outside the site's infrastructure. When used to cache dynamically generated content, proxy-based approaches typically employ *page-level caching*, where the proxy caches full page outputs of dynamic sites. This approach has been considered in the literature, e.g., [11, 10], and serves as the basis for a number of commercial solutions as well. Some of these solutions operate in *reverse proxy* mode (e.g., Inktomi's Traffic Server [4]). Other solutions are deployed in *forward proxy* mode (e.g., *Content Delivery Networking* (CDN) solutions offered by vendors such as Akamai [1]), and are based largely on the fundamental body of work that addresses distributed proxy caching, e.g., [15].

In general, page-level caches can improve web site performance by reducing (a) delays associated with generating the content, (b) delays associated with packet filtering and other firewall-related delays, and (c) the bandwidth required to transmit the content from the back end application to the proxy-based cache. However, this approach suffers from two major drawbacks: (1) full-page dynamically generated HMTL files generally have little reusability, leading to low hit ratios; and (2) cache hits are determined based on a request's URL, which does not necessarily uniquely identify the page content, leading to the possibility of serving incorrect pages from cache.

An approach that attempts to address the issue of page reusability is *dynamic page assembly*, an approach popularized by Akamai [1] as part of the Edge Side Includes (ESI) initiative [12]. This approach entails decomposing each page into a number of fragments (specifically, separate dynamic scripts) that are used to assemble the page at a network cache when the page is requested. While dynamic page assembly allows content to be cached at finer granularities, it has two major limitations. First, this approach requires that the page layouts be specified in advance. Secondly, dynamic page assembly cannot be used in the context of pages with semantically interdependent fragments.

Back-end caches typically reside within the site infrastructure, and cache at the granularity of a *fragment*, i.e., a portion of a Web page. This type of cache attempts to reduce the computational and communication resources required to build the page on the site, thus reducing server-side delays. Back-end caching approaches include various types of database caching (e.g., [8, 6]), as well as presentation layer caching (e.g., [2]). Another more general back-end caching approach is *component level caching* [13], which caches arbitrary objects, addressing delays due to computation as well as delays due to communication between different modules (available commercially from Chutney Technologies [3]).

In addition to reducing server-side delays, back-end caching solutions address the above-mentioned limitations of proxy-based approaches: (1) they allow caching at finer granularities and (2) they do not rely on URLs to identify cached content, and thus guarantee correctness of the contents in a generated page. However, this type of solution does not reduce the bandwidth needed to connect to the server to obtain content.

4 Dynamic Proxy-Based Caching Approach

In this section, we describe our proposed approach for granular proxy-based caching of dynamic content, which attempts to combine the above-mentioned benefits of proxy-based and back-end caching approaches, without the limitations.

Our objective is to deliver dynamic pages from proxy caches. Any dynamic content caching system must account for dynamic content and layout - in fact, the primary weakness of existing proxy caching schemes arises from their inability to map a URL to the appropriate content and layout. To mitigate this weakness, our essential intuition may be summarized as follows: We will route a request R_i through a dynamic proxy, D_i, to the site infrastructure. Upon reaching the site infrastructure, R_i will cause the appropriate dynamic script to run. A back end module will observe the running of this script and determine the layout of the page to be generated. This layout, which will be much smaller than the actual page output, will be routed to the proxy D_i. The proxy will fill in the content from its cache and route it to the requestor.

There are two main components in our dynamic proxy caching system, the Dynamic Proxy Cache (DPC) and the Back End Monitor (BEM). The *Dynamic Proxy Cache* (DPC) stores dynamic fragments outside the site infrastructure and assembles these fragments in response to user requests. The BEM resides at the back end and generates the layout for each request. This layout is passed back to the DPC, which assembles the page that is returned to the requesting user.

The DPC can reside either (a) at the origin site (in a reverse proxy configuration), or (b) at the network edge (in a forward proxy configuration). In the former case, the primary benefit is the reduction in the number of bytes transferred through the site infrastructure for each request. In the latter case, the forward proxy configuration (similar to that of present-day CDNs), the benefits are even greater – the reduction in bytes transferred for each request is realized not only within the site infrastructure, but also across the Internet. This paper will focus on the reverse proxy case.

A prerequisite of our dynamic proxy caching system is that the cacheable fragments be identified and marked. Once the cacheable fragments are identified, each of the corresponding code blocks in the script is *tagged*. Tagging essentially means marking a code block as cacheable, an initialization activity. This is done by inserting APIs around the code block, enabling the output of the code block to be cached at run-time. The tagging process assigns a unique identifier to each cacheable fragment, along with the appropriate metadata (e.g., time-to-live).

At run-time, a user submits a request to the site. The application logic in the script runs as usual, until a tagged code block is encountered. When such a code block is encountered, a check is made to see whether the fragment produced by that code block exists in the DPC. This is done by looking up the fragmentID in the BEM. There are two general cases possible:

1. **The fragmentID is not in cache or is in cache but invalid.** In this case, an entry is inserted into the BEM

for this fragment, the content is generated, and a `SET` instruction is written to the page template. This instruction will insert the fragment into the DPC.

2. **The fragmentID is in cache and is valid.** In this case, a `GET` instruction is written to the page template. This instruction will retrieve the fragment from the DPC.

For the first request for a given page, none of the fragments will be in cache, so the layout will consist of `SET` instructions, along with the generated content. For subsequent requests, the cacheable fragments will likely be cached, assuming that they have not been invalidated. In this case, the layout will consist mostly of `GET` instructions and hence will be much smaller.

We now describe the BEM in more detail. The BEM has two primary functions: (1) managing the cache for the DPC, and (2) caching intermediate objects. We discuss the former function, since the latter has been described elsewhere [13].

Managing the DPC cache is a critical function of the BEM. This function is enabled by the *cache directory*, a critical data structure contained in the BEM. The cache directory keeps track of the fragments in the DPC and their respective metadata. The cache directory includes the following elements: `fragmentID`, a unique fragment identifier (assigned during initialization), `dpcKey`, a unique fragment identifer (assigned by the system), `isValid`, a flag to indicate validity of fragment, and `ttl`, a time-to-live value for the fragment.

The `dpcKey` is a unique integer identifier associated with each fragment that serves as a common key for both the BEM and the DPC. The use of such an integer key has two benefits: (a) it reduces the tag size, and (b) it eliminates the need for explicit communication between the BEM and the DPC.

There are two basic ways in which fragments can become invalid: (a) an invalidation policy determines that a fragment is invalid, or (b) a replacement policy determines that a fragment should be evicted from cache. In any case, the fragment's `isValid` flag will be set to `FALSE` to indicate that it is no longer valid. When this occurs, the `dpcKey` for the fragment is inserted at the end of a list of available `dpcKeys`. This technique ensures that a subsequent request for the fragment will be generated and served fresh.

5 Analytical Results

There are two types of benefits that accrue in our model: (a) performance and scalability of the server side, and (b) bandwidth savings. In this section, we briefly present the results of our bandwidth savings analysis. Further details can be found in [14]. Table 1 contains the notation to be used throughout this section.

Symbol	Description
$\mathcal{E} = \{e_1, e_2, \ldots, e_m\}$	set of fragments
$\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$	set of pages
$E_i = \{e_j : e_j \in c_i\}$	set of fragments corresponding to page c_i
s_{e_j}	average size of fragment e_j (bytes)
g	average size of tag (bytes)
f	average size of header (bytes)
h	hit ratio, i.e., fraction of fragments found in cache
R	total number of requests during observation period

Table 1. Notation

In our analysis, we wish to compare the bandwidth savings for two cases: (a) with the dynamic proxy cache and (b) without. For the purposes of this analysis, we model a given Web application as a set of such pages $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$. Each page is created by running a script (as described in Section 2), and the resulting page consists of a set of fragments, drawn from the set of all possible fragments, $\mathcal{E} = \{e_1, e_2, \ldots, e_m\}$. We let E_i, $E_i \subseteq \mathcal{E}$, be the set of fragments corresponding to page c_i. There exists a many-to-many mapping between $\mathcal{C}$ and $\mathcal{E}$, i.e., a page can have many fragments and a fragment can be associated with many pages. The average size of a fragment e_j is denoted by s_{e_j}. Each page also has f bytes of header information (e.g., HTTP headers) associated with it. We define *expected bytes served*, $\overline{B}$, as the average number of bytes served by the Web site that is hosting the application during some time interval.

We compute $\overline{B}$ based on the size of each response and the number of times the page is accessed during the time interval. The general form of $\overline{B}$ over a given time interval is given by: $\Sigma_{i=1}^{n} S_{c_i} \times n_i(t)$, where $n_i(t)$ is the number of times the page c_i is accessed during the specified time interval, and S_{c_i} is the size of the response corresponding to page c_i as delivered by the hosting site. $n_i(t)$ is given by $\mathcal{P}(i) \int_{t_1}^{t_2} f(t)dt$, where $\mathcal{P}(i)$ is the probability that page c_i is accessed for a given request and $f(t)$ is the probability density function (pdf) that describes the arrival rate of requests. We assume that $\mathcal{P}(i)$ is governed by the Zipfian distribution, which has been shown to describe Web page requests with reasonable accuracy [9].

For the no cache case, the size of the response for page c_i, denoted as $S_{c_i}^{NC}$, is given by $\Sigma_{\forall e_j \in c_i} s_{e_j} + f$, which is the sum of the sizes of all the fragments on the page and the header information. For the dynamic proxy cache case, the size of the response for page c_i, denoted as $S_{c_i}^{C}$, is given by $\Sigma_{\forall e_j \in c_i}[X_j[(h \times g) + (1-h)(s_{e_j} + 2g)] + (1 - X_j)(s_{e_j}) + f]$, where X_j is an indicator variable denoting whether a frag-

ment is cacheable. We have compared the expected bytes served for the two cases using the baseline parameter values shown in Table 2.

Parameter	Value
hit ratio (h)	0.8
fragment size (s_e)	1K bytes
number of fragments per page	4
number of pages	10
average size of header information (f)	500 bytes
tag size (g)	10 bytes
cacheability factor	0.6
number of requests during interval (R)	1 million

Table 2. Baseline Parameter Settings for Analysis

Figure 1(a) (the curve labeled 'Analytical') shows the results of the comparison of the ratio $\frac{\overline{B}^C}{\overline{B}^{NC}}$ as fragment size (s_e) is varied. As this figure shows, this ratio decreases as fragment size increases. For small fragments sizes (e.g., less than 1 KB), the ratio exhibits a steep drop. This drop can be explained as follows: For small fragment sizes, the size of the tags is large with respect to the fragment size, decreasing the savings in bytes served for the dynamic proxy cache. This is why the ratio is greater than 1 as the fragment size approaches 0. As these results indicate, our dynamic proxy caching technique has a greater impact for larger fragment sizes (e.g., greater than 1 KB).

We now examine the sensitivity of expected bytes served with respect to the hit ratio (h), while holding all other parameter values constant. Figure 1(b) (the curve labeled 'Analytical') shows the percentage savings in expected bytes served as the hit ratio is varied from 0 to 1. In the case where no fragments are served from cache (i.e., $h = 0$), we see that the savings is negative. In other words, there is a cost to use the dynamic proxy cache in this case because it adds tags to the responses, thereby increasing the response sizes. This effect holds up to the point where $h = 0.01$. Thus, as long as 1% or more fragments are served from cache, using the dynamic proxy cache will reduce the expected bytes served.

Figure 2(a) (the curve labeled 'Analytical') shows a comparison of the percentage savings in expected bytes served as the *cacheability factor* is varied. The cacheability factor is the percentage of all fragments that are cacheable for a given application. As expected, this savings increases as the cacheability ratio increases.

6 Experimental Results

We have implemented our dynamic proxy caching system. Both the DPC and the BEM are written in C++. The

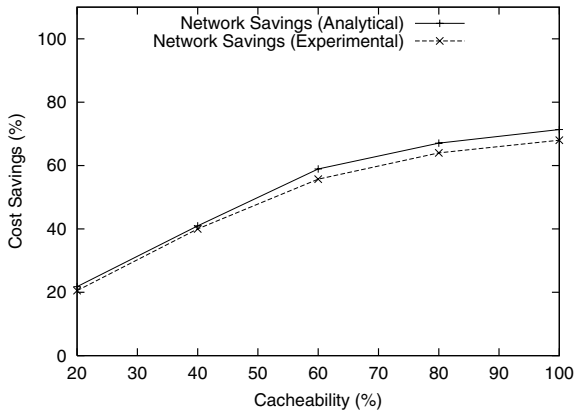


Figure 2. Comparison of Cost Savings

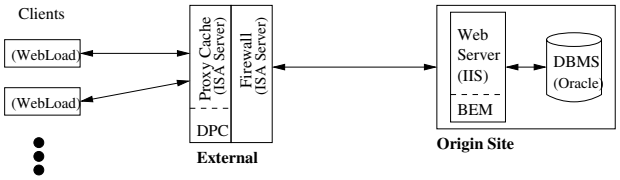


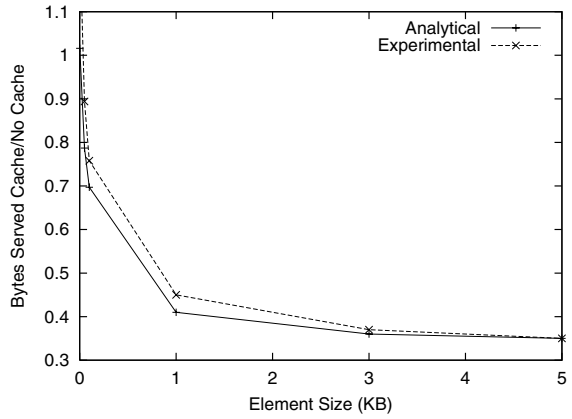
Figure 3. Test Configuration

DPC is built on top of Microsoft's ISA Server [5] so that we can take advantage of ISA Server's proxy caching features. The page assembly code is implemented as an ISAPI filter that runs within ISA Server.

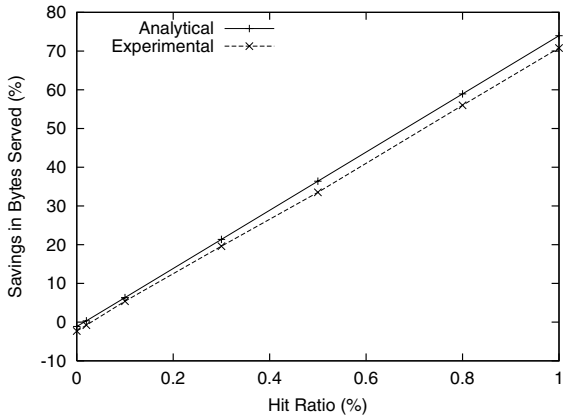
We ran a set of experiments in an attempt to validate our analytical results. Our experiments were run in a test environment that attempts to simulate the conditions described in Section 5. Thus, we have incorporated the parameter settings in Table 2. The test site is an ASP-based site which retrieves content from a site content repository.

The basic test configuration consists of a Web server (Microsoft IIS), a site content repository (Oracle 8.1.6), a firewall/proxy cache (ISA Server), and a cluster of clients. The client machines run WebLoad, which sends requests to the Web server. For the dynamic proxy cache case, the DPC runs on the ISA Server machine, and the BEM runs on the IIS machine. Communication between all software modules is via sockets over a local area network. Figure 3 shows the test configuration. The number of bytes served is obtained by measuring bandwidth using the Sniffer network monitoring tool [7]. More precisely, the bandwidth measurement is taken between the Origin Site machine and the External machine in Figure 3.

Figure 1(a) (the curve labeled 'Experimental') shows the ratio $\frac{\overline{B}^C}{\overline{B}^{NC}}$ as fragment size is varied. As this figure shows, our experimental results follow our analytical results closely. Interestingly, the analytical curve falls below



(a) $\overline{B}^{C}/\overline{B}^{NC}$ vs. Fragment Size



(b) Expected Bytes Served (%) vs. Hit Ratio

Figure 1. Analytical and Experimental Results

the experimental curve. This difference can be explained by the network protocol headers (e.g., TCP/IP headers) that are included in the responses. However, we do not account for these headers in our analytical expressions. Thus, for every response, there is some network protocol messaging overhead. The smaller the response, the greater this overhead is. This is why the difference between the analytical and experimental curves is higher for smaller fragment sizes than it is for larger fragment sizes.

As in Section 5, we now examine the sensitivity of expected bytes served to changes in hit ratio. Figure 1(b) shows the results of this analyis. Here again, our experimental results closely follow our experimental results.

Figure 2 shows a comparison of the sensitivity of the percentage savings in expected bytes served to changes in cacheability. Once again, the experimental results follow our analytical results closely.

7 Conclusion

In this paper, we have proposed an approach for granular, proxy-based caching of dynamic content. The novelty in our approach is that it allows both the content and layout of Web pages to be dynamic, a critical requirement for modern Web applications. Our approach combines the benefits of existing proxy-based and back end caching techniques, without their respective limitations. We have presented the results of an analytical evaluation of our proposed system, which indicates that it is capable of providing significant reductions in bandwidth on the site infrastructure. Furthermore, we have described an implementation of our system and presented experimental results, which demonstrate that our system is capable of providing order-of-magnitude reductions in bandwidth requirements.

References

[1] Akamai technologies. http://www.akamai.com.

[2] Bea systems. http://www.bea.com/products/weblogic/index.html.

[3] Chutney technologies. http://www.chutneytech.com.

[4] Inktomi. http://www.inktomi.com/products/network/.

[5] Microsoft isa server. http://www.microsoft.com/isaserver.

[6] Oracle corp.: Oracle 9ias database cache. http://www.oracle.com/ip/deploy/ias/db_cache_fov.html.

[7] Sniffer technologies. http://www.sniffer.com/products/sniffer-basic/.

[8] Timesten software. http://www.timesten.com.

[9] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, 1996.

[10] K. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the ACM SIGMOD 2001 Conference*, pages 532 – 543, 2001.

[11] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[12] E. Consortium. Edge side includes. http://www.esi.org, 2001.

[13] A. Datta, K. Dutta, D. Fishman, K. Ramamritham, H. Thomas, and D. VanderMeer. A comparative study of alternative middle tier caching solutions. In *Proceedings of the 2001 VLDB Conference*, Rome, Italy, September 2001.

[14] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: An approach and implementation. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, June 2002.

[15] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large internet caches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 1997.