A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration

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

Krithi Ramamritham
IIT Bombay and Univ. of Mass.
krithi@cse.iitb.ernet.in

Dan Fishman
BEA Systems
danf@bea.com

1 Introduction

E-business sites are increasingly utilizing dynamic web pages since they enable a much wider range of interaction than static HTML pages can provide. Dynamic page generation technologies allow a Web site to generate pages at run-time, based on various parameters. Delaying content decisions until run-time affords a Web site significant flexibility in customizing page content, thereby enriching users' Web experiences. At the same time, however, dynamic page generation technologies have resulted in serious performance problems due to the increased load placed on the server-side infrastructure. Consequently, end users experience increased response times. According to recent research [1], 40% of the total page delivery delay experienced by end users can be attributed to server-side latency. As server-side techniques such as dynamic page generation techologies become more widespread, this percentage will only increase.

There has been very little work so far to address the delays associated with dynamic page generation. One proposed approach is to cache entire pages of dynamically generated content (e.g., [3, 5]). However, caching dynamically generated pages in this manner is infeasible, since two calls to the same script with the same input parameters are not guaranteed to produce the same output. Our approach to this problem is to cache *fragments* or *components* of dynamically generated pages. Consider a dynamically generated page in an online news site. This page may consist of a *Current Headlines* component, a *Navigation Bar* component, an *Ad* component, and perhaps a *Personalized* component, (e.g., containing a personal greeting for the returning visitor). Even though the *Personalized* and *Ad* components are likely to be different for each request, the *Current Headlines* and *Navigation* components will likely be the same across all requests, and therefore, can be cached to serve future requests.

Based on this notion, we have built a *dynamic content accelerator*, a server-side caching engine that caches such dynamic page fragments in order to reduce dynamic page generation processing delays on a Web site. The novelty of our approach lies not only in the caching of dynamic page fragments, but also in the intelligent cache management strategies utilized. In particular, we have developed a set of *prediction-based* and *observation-based* techniques that are used in both cache replacement and cache invalidation. The end result is a substantial reduction in processing load on the web/application server, allowing the server to handle significantly higher user loads. In a later section, we present performance results which demonstrate that our solution significantly outperforms existing middle tier caching solutions.

2 Background and Related Work

Dynamic scripting technologies, such as Active Server Pages (ASP) (www.microsoft.com) and Java Server Pages (JSP) (www.sun.com), allow Web sites to assemble pages "on the fly" based on various run-time parameters in an attempt to tailor content to each individual user. A major disadvantage of such dynamic

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

scripting technologies, however, is that they reduce Web and application server scalability because of the additional load placed on the Web/application server. In addition to pure script execution overhead, the delays caused by dynamic scripting technologies include: delays due to fetching content from persistent storage (e.g., database systems), delays due to data transformations (e.g., XML to HTML transformations), and delays due to executing business logic (e.g., personalization software).

There has been very little work so far to address the problem of dynamic page generation delays. As mentioned previously, one approach is to cache entire pages of dynamically generated content (e.g., [3, 5]). Clearly, this approach has very limited applicability since two identical requests for the same script are not guaranteed to produce the same output. Another recent approach proposes a 3-tier customizable cache system which supports data materialization at various levels, such as database output, XML or HTML fragments [4]. A major limitation of this approach is that it requires that the site be designed using a particular declarative web site specification language.

A number of commercial dynamic content caching solutions are beginning to enter the market. Dynamai (www.dynamai.com), SpiderCache (www.spidercache.com), and XCache (www.xcache.com) are all solutions that cache dynamically generated pages, essentially commercializing the work described in [3], and suffering from the same limitations. Another class of commercial solutions that can be used to mitigate page generation delays are middle tier database caching solutions. These products include main memory database solutions from vendors such as TimesTen (www.timesten.com), and application server-specific solutions such as the Oracle 9-i Cache (www.oracle.com). These solutions can mitigate local database access latency by caching database tables. However, these solutions do not affect other types of delay associated with page generation, such as XML to HTML transformations. Furthermore, these solutions do not remove the need to establish a database connection, perhaps the scarcest resource on high-traffic, data-intensive Web sites.

3 Our Solution

Our *Dynamic Content Acceleration* (DCA) solution utilizes a fragment-level caching approach which focuses on re-using HTML fragments of dynamic pages. A dynamic script typically consists of a number of *code blocks*, where each code block performs some work that is required to generate the page and produces an HTML fragment as output. A *write to out* statement, which follows each code block, places the resulting HTML fragment in a buffer. When a dynamic script runs, each code block is executed and the resulting HTML fragment placed in the buffer. Once all code blocks have executed, the entire HTML page is sent as a stream to the user. A high-level depiction of the dynamic scripting process for our earlier news page example is shown in Figure 1A.

As indicated by the dotted lines in the figure, each code block corresponds to a component. If we know that the *Current Headlines* and *Navigation* components are reusable, we may choose to cache these components. This is accomplished by marking or *tagging* the corresponding code blocks within the script. When the script is executed, the tags instruct the application server to first check the cache before executing the code block. If the requested fragment is found in cache, then the code block logic is bypassed. If the requested fragment is not found in cache, then the code block is executed and the requested fragment is generated and subsequently placed in the cache.

A critical aspect of any caching solution is cache management. As the cache becomes full, the effectiveness of the cache replacement policy dictates the hit rates of the cache, and thus its performance. Our cache replacement algorithm, which we refer to as *Least Likely to be Used* (LLU), is based on a predictive technique. When choosing a replacement "victim", it takes into account not only how recently a cached item has been referenced, but also whether any user is likely to need the item in the near future. Also, as the underlying source data changes, some mechanism is required to keep the cached components fresh. Our solution supports several existing invalidation techniques (e.g., time-based and event-based invalidation) which have been adapted to work in the context of our component-level cache. We have also developed additional invalidation techniques, such as *observation-based invalidation*, which observes changes to data without contacting the data source.

A simplified depiction (without routers, firewalls, etc.) of an end-to-end web site architecture is shown in Figure 1B. As the figure shows, the DCA sits adjacent to the server rack, along with other resources, such as site content databases.

4 Performance Results

In this section, we present the results of a set of tests in which we demonstrate the performance of our caching solution. The metrics recorded in our tests are *average page generation time* and *average response time*. The average page generation time measures the page generation latency and is the average time required to construct a complete HTML page. Response time is an important metric from the user's perspective, and is the end-to-end delay in delivering an HTML page, including the time required to fetch rich content.

The test site used for these experiments is a catalog-based e-commerce site (a variant of Bluestone Software's Sonic site). The pages in the site are gen-

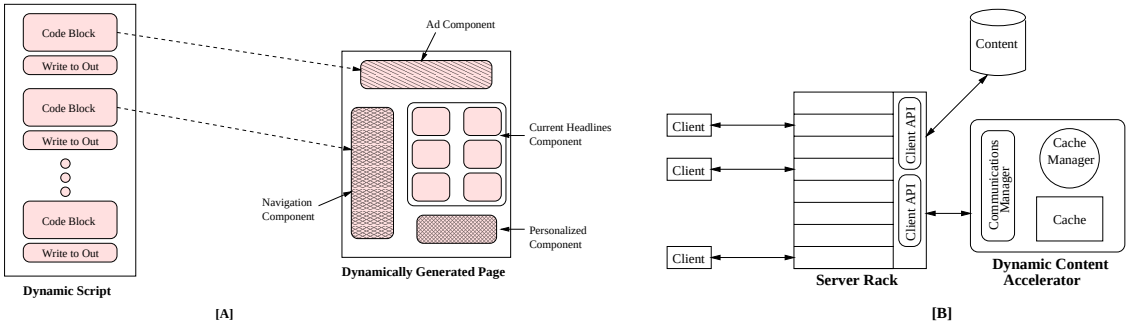


Figure 1: Dynamic Scripting Process and Architecture

erated using JSP scripts, which retrieve the product catalog data from a database. The site content database is based on the Dublin Core Metadata Open Standard (http://purl.org/dc), and contains approximately 200,000 products and 44,000 categories. Each page in the site consists of three components, where each component invokes a call to the database.

For the comparison, the two products we have selected are (1) a main memory database solution, TimesTen, and (2) an application server caching solution, BEA's WebLogic Server JSP Cache tags (WLS Cache). In these experiments, the TimesTen cache is large enough to hold the entire database so that no disk accesses will be required. For both the WLS Cache and DCA solutions, two page components are marked as cacheable.

A clustered web/application server architecture is used. The web/application server cluster consists of two server-class Intel-based machines, each having 1 GB RAM and dual Pentium III 933 Mhz processors, running WebLogic 6.0 on a Windows 2000 Advanced Server platform. The content database resides on a similar machine running Oracle 8.1.6. All other machines are Pentium III-600 machines having 512 MB RAM running Windows 2000 Advanced Server.

User load is varied by sending requests from client machines using RadView's WebLoad (www.radview.com). To model the locality that is often present in site navigation patterns, user navigation is simulated according to a Zipf 80-20 distribution (i.e., 80% of the users follow 20% of the navigation links). The DCA cache size, the percentage of fragments or components that fit in cache, is 0.75 (i.e., 75% of the total number of fragments may reside in the cache). Finally, the cache replacement policy used is LLU.

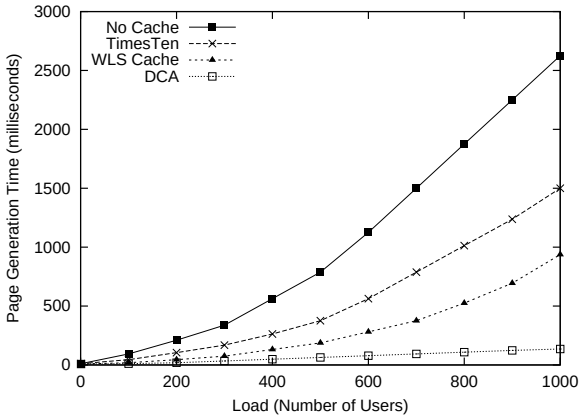
The baseline results are shown in Figure 2a. This figure shows page generation times (in milliseconds) as load is varied from 1 to 1000 users for each of the solutions under consideration, as well as the No Cache case (the uppermost curve). Each of the four curves exhibits an increasing trend as load increases. However, the rate of increase differs among the curves, which we now discuss in more detail.

As expected, the DCA case outperforms the No Cache case by a significant margin, ranging from about an 8 times improvement for low loads (e.g., 100 users) to about a 19 times improvement for higher loads (e.g., 1000 users).

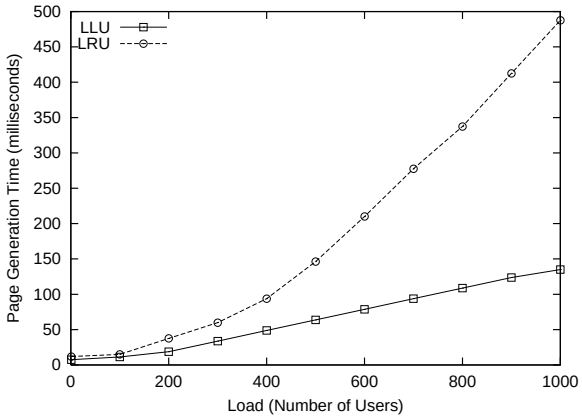
The TimesTen case does provide some improvement in page generation times over the No Cache case. For the most part, the TimesTen solution reduces page generation times by about half. This reduction is due to the fact that no disk access is required (recall that the entire database is cached in this case). However, the TimesTen solution does not perform nearly as well as the DCA solution. At low loads (e.g., 100 users), the DCA case outperforms the TimesTen case by about a factor of 4, while for higher loads (e.g., 1000 users), DCA outperforms TimesTen by about 11 times. This phenomenon can be explained by the fact that TimesTen does not address many of the delays associated with dynamic page generation.

The WLS Cache solution also provides improvement in page generation times and outperforms the TimesTen solution. The WLS Cache solution is especially interesting because, like the DCA solution, it also takes the approach of caching HTML fragments. However, the DCA case outperforms the WLS Cache case. The difference in performance occurs mostly at medium to higher loads. For instance, at a load of 500 users, the DCA solution is about 3 times faster than WLS Cache, and at a load of 1000 users, DCA is nearly 7 times faster than WLS Cache. The difference in performance is primarily due to architectural differences. For instance, one major difference is that the WLS Cache runs in the same process space as the WebLogic application server. Thus, the caching operations compete with the numerous other application server tasks for both CPU and memory resources. In addition, since this is a clustered environment, WLS Cache utilizes multiple caches. This results in redundancy of information across the caches, causing poor memory utilization. The DCA solution, on the other hand, does not suffer from these problems since it runs as a single logical cache instance, separate from the application server.

We now examine the impact of varying the cache re-



(a) DCA vs. other Caching Middleware



(b) Sensitivity to Cache Replacement Policy

Figure 2: Comparison Results: Baseline and Sensitivity to Replacement Policy

placement policy. In particular, we compare the performance of the LLU cache replacement policy with that of the commonly used LRU policy. Figure 2b shows the resulting page generation times. Interestingly, LLU outperforms LRU over the entire range. The difference ranges from about a 1.5 times improvement for a single user, to a more than 3.5 times improvement for 1000 users. The better performance of LLU can be explained by the additional information (e.g., historical site visitation patterns, current location of users) that the algorithm utilizes in making replacement decisions. Additional sensitivity results can be found in [2].

5 Conclusion

As e-business sites increasingly adopt dynamic page generation technologies, web and application server scalability is significantly reduced because of the additional load placed on these servers. In response to this problem of dynamic page generation latency, we have developed a Dynamic Content Accelerator. This fragment-level caching solution, combined with intelligent cache management strategies, can significantly reduce the processing load on the web/application server, allowing the server to handle higher user loads. Our performance results indicate that our solution significantly outperforms existing middle tier caching solutions.

References

[1] Microsoft Corp. C. Huitema. Network vs. server issues in end-to-end performance. Keynote address, Performance and Architecture of Web Servers Workshop, held in conjunction with ACM SIGMETRICS, June 2000.

[2] A. Datta, K. Dutta, D. Fishman, K. Ramamritham, H. Thomas, and D. VanderMeer. Dynamic page generation delays and the role of chutney's acceleration solution. Technical Report CIR-0012, Chutney Technologies, Inc., February 2001.

[3] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[4] Khaled Yagoub, Daniela Florescu, Valérie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 188–199. Morgan Kaufmann, 2000.

[5] H. Zhu and T. Yang. Cachuma: Class-based cache management for dynamic web content. Technical Report TRCS00-13, Dept. of Computer Science, The University of California at Santa Barbara, June 2000.