Multi-tiered Cache Management for E-Commerce Web Sites

Wang-Pin Hsiung Wen-Syan Li K. Selçuk Candan Divyakant Agrawal

C&C Research Laboratories - Silicon Valley
NEC USA, Inc.
10080 North Wolfe Road, Suite SW3-350
Cupertino, California 95014
Email:{whsiung,wen,candan,agrawal}@ccrl.sj.nec.com
Tel:408-863-6008 Fax:408-863-6099

Abstract

Response time is a key differentiation point among electronic commerce (e-commerce) applications. For many e-commerce applications, Web pages are created dynamically based on the current state of a business stored in database systems. To improve the response time, many e-commerce Web sites deploy caching solutions for acceleration of content delivery. There are multiple tiers in the content delivery infrastructure where cache servers can be deployed, including (1) data caching (in data centers), (2) content page caching (in edge or frontend caches), (3) database query result set caching (between application servers and DBMS). The architecture of database-driven e-commerce Web sites are more complex than that of typical Web sites. It requires the integration of Web servers, application servers, and back-end database systems as well as dynamic content caching solutions. In this paper, we study issues associated with management of content page caching and database query result set caching tiers. We observe that caching management for these two tiers have their unique characteristics. It is because cached object types and information available for the caching management in the two tiers are different. We propose solutions for effective caching management for each tier and conduct extensive evaluations. The experiment results show the usefulness of our technology in improving overall system performance.

Keywords: cache replacement, multi-tiered caching, dynamic content caching, Web acceleration application server

1 Introduction

Response time and reliability are two key points of differentiation among electronic commerce (EC) Web sites. Snafus and slowdowns with major Web sites during special events or peak times demonstrate the difficulty of scaling up e-commerce sites. Such slow response times and down times can be devastating for e-commerce sites as indicated in a recent study by Zona Research[1] on the relationship between page download time and user abandonment rate. The study shows that only 2 percent of users will leave a Web site (i.e. abandonment rate) if the download time is less than 7 seconds. However, the abandonment rate jumps to 30 percent if the download time is around 8 seconds. And the abandonment rate goes up to 70 percent when the download time is around 12 seconds. This shows the importance of fast response time to a e-commerce Web site to retain its customers.

One possible solution to the issues of response time and reliability is to deploy caches so that some requests

can be served remotely rather than going to Web sites across networks. This solution has two advantages: serving users via a server closer to the users and reducing the traffic to the Web sites. Many content delivery network (CDN) vendors provide Web acceleration services for static content and streaming media. However, for many e-commerce applications, HTML pages are created dynamically based on the current state of business, such as prices and inventory. Furthermore, freshness of Web pages is essential to e-commerce practices. For example, the price and availability of an auction item must be updated periodically; however the time to live (TTL) of these dynamic pages can not be estimated in advance. As a result, dynamic content delivery by most CDNs are limited to handling static pages customized using cookies, rather than the full spectrum of dynamic content. We also observe that due to the dynamic nature of e-commerce businesses, a large number of e-commerce Web sites are database-driven. A typical database-driven Web site consists of the following three components:

- a database management system (DBMS) to store, maintain, and retrieve the business data;
- an application server (AS) that receives requests from the Web server. The application server may access databases to retrieve the most current business information to create dynamic content; and
- a Web server (WS) which receives user requests and delivers the dynamically generated Web pages.

As the significance of CDN services [2, 3] becomes widespread, many database and application server vendors are beginning to integrate Web acceleration though data caching in their software. Examples include Oracle 9i [4] which features a suite of application server, Web server, and data cache for deployment at data centers for accelerating the delivery of dynamic content. With all of these newly available software for dynamic content caching and delivery acceleration, it is more flexible to architect a "distributed" Web site, which may actually be located in multiple networks and geographical regions as studied in [5, 6, 7]. In [8], Li et al. investigate factors that impact performance of Web applications, such as system resource, caching solution selection, network latency, cache locations, etc. They also evaluate various caching solutions through experiments. The experimental results indicate caching solutions do effectively accelerate Web applications and it is beneficial to deploy available caching solutions.

There are multiple tiers in the content delivery infrastructure where cache servers can be deployed. They include

- **database caches**, which are usually placed in data centers. Database caches are considered mirror databases and they are used by applications hosted in the data centers. By generating requested contents from locations close to users, content delivery can be accelerated. The work described in [9, 10] focuses on caching issues in this tier.
- **content page caches**, which are reverse caches and can be placed close to users, functioning as edge caches, or in front of Web servers, functioning as frontend caches. The work described in [11, 12, 13, 14, 15, 16, 17] focuses on caching issues in this tier.

- **database query result set caches**, which are placed between application servers and database systems for acceleration of database access via JDBC or ODBC interfaces. The work addressed in [18] focuses on the caching issues in this tier.

In this paper, we focus issues in caching management in the tiers of content page caching and database query result set caching. We describe a typical deployment configuration as a case study to illustrate complexity of management of multiple tiers of caches. After addressing the issues, we propose our caching management solutions. We have conducted extensive evaluations and experimental results show the usefulness of our technology in improving overall system performance.

The rest of this paper is organized as follows. In Section 2 we describe a system configuration of multi-tiered cache deployment for e-commerce Web site acceleration in the scope of NEC *CachePortal* technology. In Section 3, we describe the implementation and experimental results of the cache manager in the content page tier. In Section 4, we describe the implementation and experimental results of the database access caching in the database query result set caching tier. In Section 5, we review related work and in Section 6 we give our concluding remarks.

2 Deployment of Multi-Tiered Caching Solutions for E-commerce Web Sites

In this section we describe a case study of a system architecture for accelerating content delivery of e-commerce Web sites. The architecture and underlying technology, referred to as *CachePortal*[12, 13, 14], enables caching of dynamic data over wide-area networks.

Applying caching solutions for Web applications and content distribution has received a lot of attention in the Web and database communities[19, 20, 21]. To ensure the freshness of the cached database-driven web pages, database changes must be monitored so that the cached pages that are impacted by the content changes can be invalidated or refreshed. In most commercial database systems, update and query statements are recorded in database logs. However, the knowledge about dynamic content is distributed across three different servers: the Web server, the application server, and the database management server. The database itself knows how and what data changes, but does not know how the data is used to generate dynamic Web pages. Consequently, two major obstacles for enabling dynamic content caching are (1) how to automatically extract the relationship between a page and the database content used to generate the page; and (2) how to efficiently identify the affected pages that must be invalidated.

It is not straightforward to create a mapping between the data and the corresponding Web pages *automatically*. One solution is to assume that the applications will be modified to provide explicit mappings [22, 23, 24, 25]. This however may not be desirable when applications change often and a looser coexistence between data sources and caching providers are desirable to ensure application portability. *CachePortal* [12] is a Web acceleration solution,

developed at NEC Research Laboratories in San Jose, which addresses this issue by not requiring any explicit mapping. The solution of the two problems mentioned above are handled by the two components in our proposed solution: the *Sniffer* and the *Invalidator*. CachePortal enables dynamic content caching by deriving the relationships between cached pages and database access via the *sniffer*; and by intelligently monitoring database changes to "eject" dirty pages from caches via the *invalidator*.

A generic system architecture deploying NEC's CachePortal technology is illustrated in Figure 1. The functionality of Web servers, application servers, and DBMSs are the same as a typical E-commerce Web site described in Section 1. In addition to the system described in [12], this new system architecture features a newly developed cache manager, a cache server for JDBC database access layer caching, by the name of `XJDBC`, and a modified sniffer module. A system architecture that deploys NEC *CachePortal* technology consists of cache servers, *CachePortal* software modules, and operational files. We describe their functions and data/control flows among these components as follows:

1. Cache servers: There are three types of cache servers: edge, frontend, and XJDBC cache servers.

   Edge cache servers: Edge cache servers are usually provided by CDN service providers rather than being owned by content providers and e-commerce Web sites. CDN providers, such as Akamai[2], have arrangements with hundreds of ISPs to have their cache servers located in those ISPs. Such arrangements would benefit both the ISPs and content providers in network bandwidth usage saving and providing better performance to the ISP users.

   Front end cache servers: Front cache servers are usually provided by content providers and Web sites instead of CDN providers. Frontend cache servers can be either software-based or hardware-based.

   XJDBC cache servers: The XJDBC cache server is deployed between the AS and DBMS to provide JDBC layer data caching functionality. It provides not only the necessary connectivity between the application server and DBMS, but also provides caching, invalidation, pre-fetching, refresh functions for database access by the application servers. Unlike the frontend/edge cache servers which store Web pages, the storage unit at the XJDBC cache servers are the query results.

   In our current implementation, all three kinds of cache servers are software-based. Edge cache and front-end cache servers are modified Apache Web Servers. Note that in our system configuration, user requests will always be first processed by the edge or front-end cache servers. If the requested pages are not in these caches, they will behave as proxy servers to request the pages from the Web site for the users. Tracking the IP address of requesting cache servers is required for sending out invalidation messages.

2. *CachePortal* software modules:

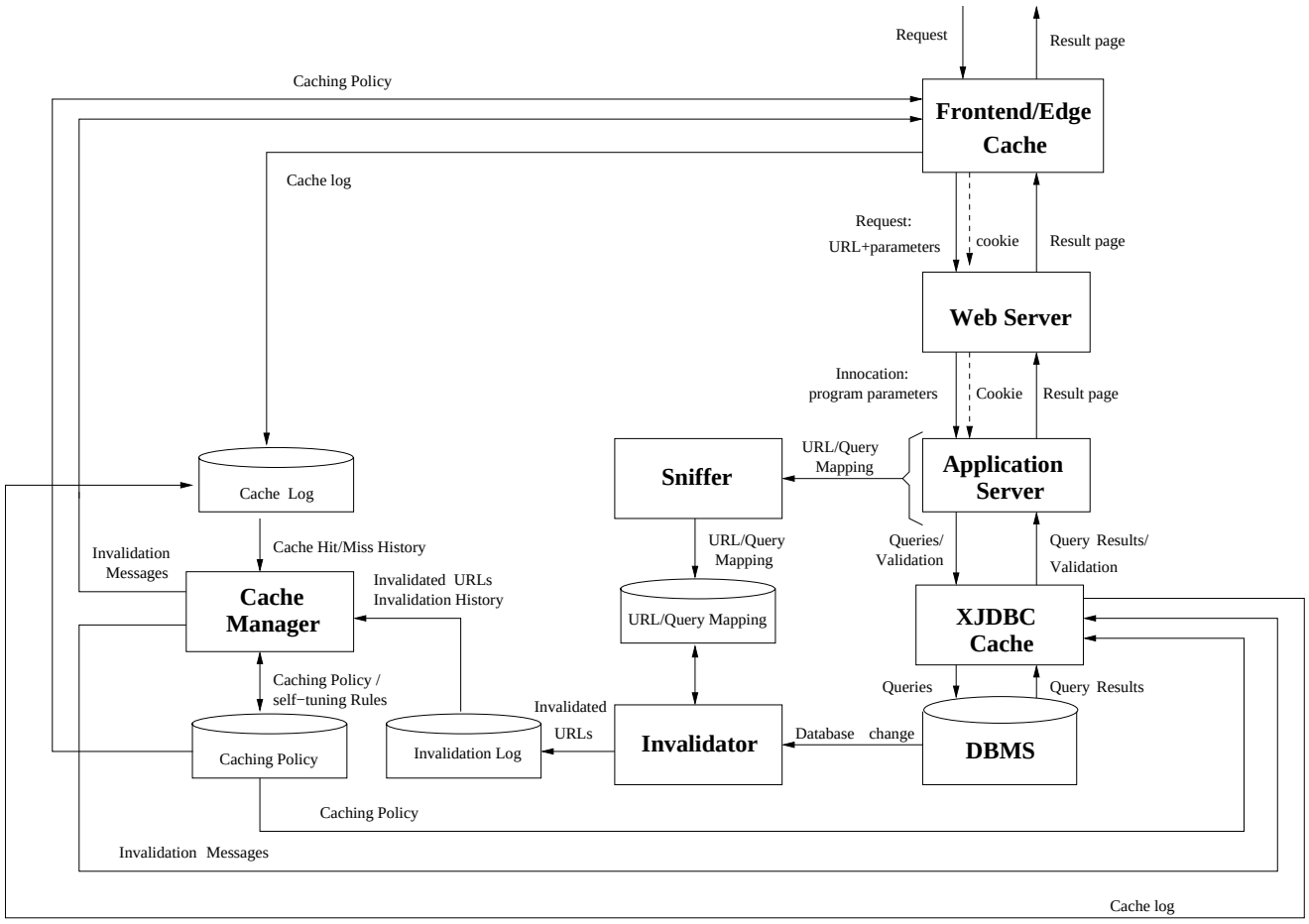

Figure 1: System architecture of a database-driven web site with *CachePortal* technology

Sniffer: *Sniffer* is a software module that behaves as a wrapper to ”sniff” the messages incoming to and outgoing from the application servers. When a request arrives, *Sniffer* extracts the URL string, cookie information, and IP address of the requesting cache server. after this information is extracted, *Sniffer* assigns (i.e. tags) each request a *unique* job identifier and forwards the request back to the appropriate application server servlet. When a job-identifier-tagged servlet issues database queries through JDBC, *Sniffer* extracts the job identifier of requesting servlet and query statements. *Sniffer* then uses the job identifier to associate each page and query statements issued to generate for the page. The association is then stored in the URL/DB Query Mapping.

Invalidator: *Invalidator* frequently polls the database log to monitor database content changes. Once the invalidator receives the log in the form of new value and old value for those tuples which have been inserted, deleted, and updated. Based on the tuple values and URL/Query DB Mapping, *Invalidator* is responsible for identifying the query results that should be invalidated. Once the query results are identified, the query statements and associated cookie information are sent to *Cache Manager* to determine the corresponding URLs and the IP addresses where the pages are cached. *Cache Manager* then sends out invalidation messages

to the appropriate cache servers. *Invalidator* is also responsible for maintaining the invalidation log, which will be utilized by *Cache Manager* to determine the caching priority of each page.

Cache Manager: *Cache Manager* is responsible for various tasks: (a) It maintains the IP addresses of the caches where the pages are cached. (b) It tunes replacement strategies and enforces the caching policies, which specifies the rules regulating which URLs must be cached and which others must not be cached. This will be described in greater detail in Section 3. (c) It pulls the cache logs from frontend, edge, and XJDBC cache servers and (d) sends out invalidation messages. In the current implementation, the cache manager maintains persistent connections with all cache servers and the XJDBC cache. Note that in this work we do not focus on how to engineer an efficient cache invalidation protocol. Many sophisticated approaches have been proposed and studied, including [26, 27].

3. Operational files:

URL/DB query map: The URL/DB query map is constructed by the sniffer. The URLs here are extended to include the IP address of the cache servers and cookie information. The URL/DB Query Mapping is maintained by multiple modules as follows:

- When a query result is invalidated (i.e. identified by the Invalidator), the Invalidator will delete the URL/DB Query entries that are associated with the query being invalidated.
- If a page is specified as non-cacheable based on the caching policy, the URL/DB Query entries that are associated with the page are deleted from the mapping.
- If a query result set in the XJDBC cache is deleted due to replacement operations initiated by the tuning tasks of the Cache Manager or the XJDBC cache itself, the URL/DB Query entries that are associated with deleted pages or query result sets are removed from the mapping.

Cache logs: The cache hit and miss history files are gathered by the cache manager from the front-end cache servers, edge cache servers, and the XJDBC cache servers. The cache hit and miss history files are used by the Cache Manager to tune the cache policy and the caching priority of objects (i.e. pages and query result sets).

Cache policy: The cache policy consists of rules on what to cache and what not to cache in regular expressions. It also stores the caching priority of certain objects that are currently in the caches or objects with high hit rates. The cache policy also includes self-tuning algorithm specifications that will be used by the Cache Manager to tune the cache replacement strategy.

Invalidation log: The invalidation log file stores the invalidation history and frequency of objects cached currently and recently. This information is used by the Cache Manager to tune the cache replacement strategy and calculate object caching priority.

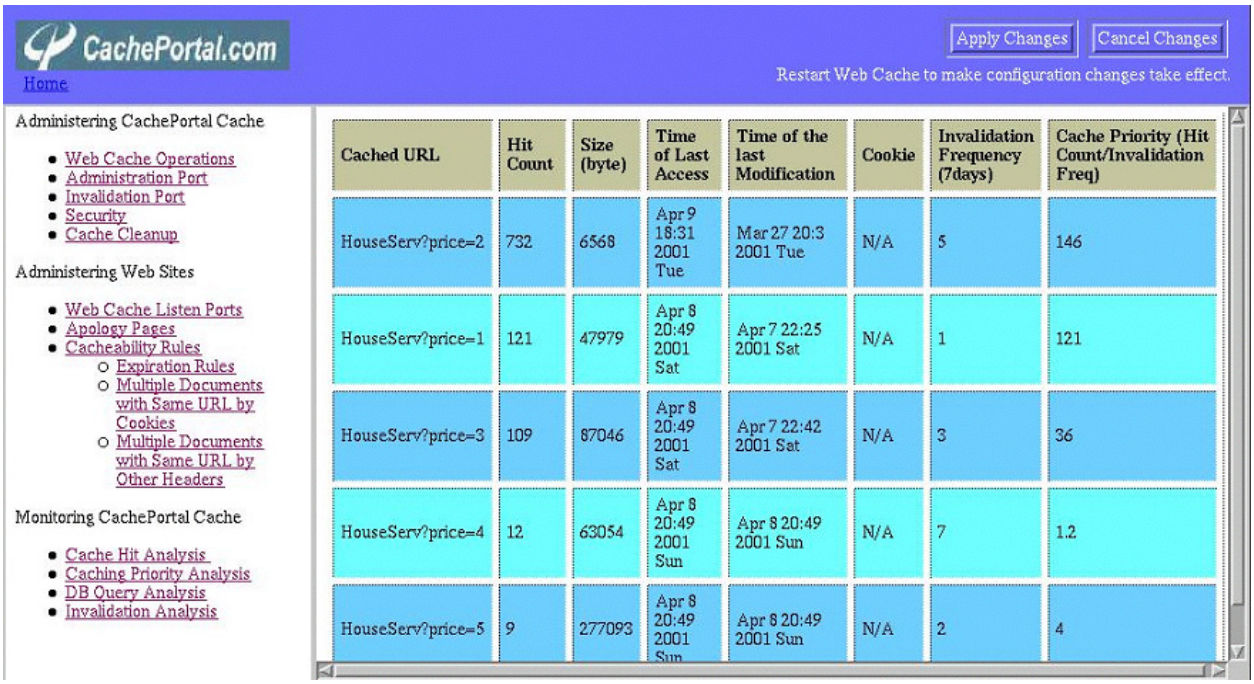

Figure 2: *CachePortal* Cache manager window

An integrated view of these operational files through a Cache Manager window is shown in Figure 2. In the window dump, we can see invalidation and cache hit frequency of each cached object and its caching priority.

We have described that the Cache Manager is responsible for (1) maintaining information about the locations of cached pages and query result sets, (2) sending out invalidation messages and (3) tuning and enforcing caching policies and replacement strategies. In the next two sections, we give details on the caching strategies and tuning algorithms that we have implemented. We also present experiment results and evaluate these algorithms. We start with the discussion of tuning caching policies for the frontend/edge cache servers followed by caching for database query result sets.

3 Content Page-Tier Caching Management

The problem of cache replacement has been extensively studied. Many algorithms have been proposed for general purpose caching, such as LRU and LFU. Some variations of these, such as [28, 29, 30], are designed specifically for caching replacement for Web pages. In [31], Cai and Irani survey nine Web page caching algorithms and provide comparisons on their performance. In [32], Barford et al. further investigate the impacts of client access pattern changes to caching performance. Most of these algorithms focus on maximizing cache hit rates based on user access patterns. However, in the scope of dynamic caching for a Web site, cache invalidation rates is an important factor since a high invalidation rate will lead to a potentially high cache miss rate in the future. As a result of the high miss rate, the WAS and DBMS have to handle a higher load to generate Web pages upon requests. Also, as a result

of high invalidation rate, the *Invalidator* component needs to handle more invalidation checking, issue more polling queries to the DBMS, and send out more invalidation messages.

3.1 Algorithms

In this section we present a self tuning cache replacement algorithm (ST) that takes into consideration (1) user access patterns, (2) page invalidation pattern, and (3) temporal locality:

- The caching priority of each page is re-calculated periodically. In the current implementation, the priority is re-calculated every minute. Note that the frequency of re-calculation does have an impact on the cache hit rate. Potentially, the more often the caching priorities are re-calculated, the higher are the cache hit rates. The frequency of re-calculation should be dynamically adjusted by considering the trade-off between the benefit of higher hit rates and the additional cost incurred due to frequent re-calculations.
- The access rate and the invalidation rate is the access count and invalidation count within a time period.
- The caching priority of a page of a time period t is calculated as

$$Caching\_priority(t) = (1-\alpha) \times \frac{access\_rate}{invalidation\_rate + 1} + \alpha \times Caching\_priority(t-1)$$

  where α is the temporal decay factor whose value is between 0 and 1. A value of 1 for α makes the system treat all access patterns equally, while a value of 0 makes the system consider only the access patterns during the current time period. Currently the value of α is set to 0.8.

The intuition behind this formula is that it estimates the average number of accesses for the page between any two successive invalidations. The higher this number the more one might want to keep this page in the cache. Note that in our current implementation, when a page is initially accessed by users, the cache manager does not try to keep it in the cache unless it is specified by the cache rule as ”must be cached”. The cache manager will monitor the caching priority of a page and will cache the page when its priority is higher than a certain threshold value.

3.2 Experiments

We have conducted experiments to evaluate our algorithm (ST) and compare its performance with LRU and LFU algorithms, two of the most frequently used algorithms. We also measured the performance of cache hits when no replacement strategy is deployed (NR). Under the NR strategy, the cache server will continue to fill in the pages until it is full. When the cache is full, a page will be cached only after some pages are removed due to invalidation.

The general experimental settings are as follows: The total number of the pages in the system is 5000, whereas the cache size is 1000 pages. All pages are of the same size. In the experiments, the cache starts empty and each

experiment is run for 200 minutes, where 5 requests are generated per second. The page access pattern shows an uneven distribution ($\frac{W}{X}$, where W percentage of the user requests access to X percentage out of all 5000 pages). The invalidation rate is also 5 requests among all 5000 pages per second. The invalidation pattern also shows an uneven distribution ($\frac{Y}{Z}$, where Y percentage of invalidations affect Z percent of the pages). When the access patterns change, these parameters stay the same, whereas the set of popular pages change. Note that the invalidation pattern is assumed to be independent of the user access patterns. The cache hit rate is calculated every 200 seconds.

In Figures 3 to 6, we show four sets of experimental results for various user access and invalidation patterns. In these figures, the X-axis represents the elapsed time and the Y-axis represents hit rates.

In Figure 3, we see that ST, LRU, and LFU all perform fairly well with the hit rates close to 80% for the pattern (80/10/90/10), which denotes 80% access to 10% pages and 90% invalidation to 10% pages. After the pattern changes at 6000 second, it takes longer time for LFU to make appropriate adjustments. This is because LFU does consider longer user access history to adjust page caching priorities. Note that the upper bound for the hit rate for the access pattern (80/10) can be calculated as

$$80\% + 20\% \times \frac{1}{9} = 82.2\%.$$

The hit rate of 82.2% can be reached when 80% of users who access these 10% of pages, that are always in the cache. And the cache still has room to cache additional 10% of pages which only serve 2.2% of additional users.

In Figures 4 and 5, we see that ST performs 10% better than LRU and LFU and the average hit rate is close to 70% for the patterns (80/20/90/10) and (80/20/90/30). Note that the upper bound for the hit rate for the access pattern (80/20) is 80%, which can be reached when 80% of users who access these 20% of pages, that are always in the cache. However, compared with the configuration in Figure 3, the configurations in Figures 4 and 5 have less flexibility to manage the cache since the cache size (i.e. 1000 pages) is the same as the size of all frequently accessed pages (i.e. $5000 pages \times 20\%$). In addition, invalidations can cause the hit rate to drop. As we can see, the average hit rate for ST in Figure 5 is lower than that in Figure 4 due to the 20% higher invalidation rate.

In Figure 6, we see that ST performs almost 15% better than LRU and LFU for the pattern (60/20/90/10). Note that the upper bound for the hit rate for the access pattern (60/20) is 60%, which can be reached when 60% of users who access these 20% of pages, that are always in the cache.

The experimental results in Figures 3 to 6 indicate that our algorithm (ST) performs well in most of cases and out-performs other algorithms substantially when it is more difficult to tune the cache priority, such as small cache size, high invalidation rate, and more diverse user access patterns.

In Figure 7, we summarize the average hit rates (long bars) and the average cache invalidation rates (short bars) for various experimental settings. The hit rates and cache invalidation rates are calculated after both rates become

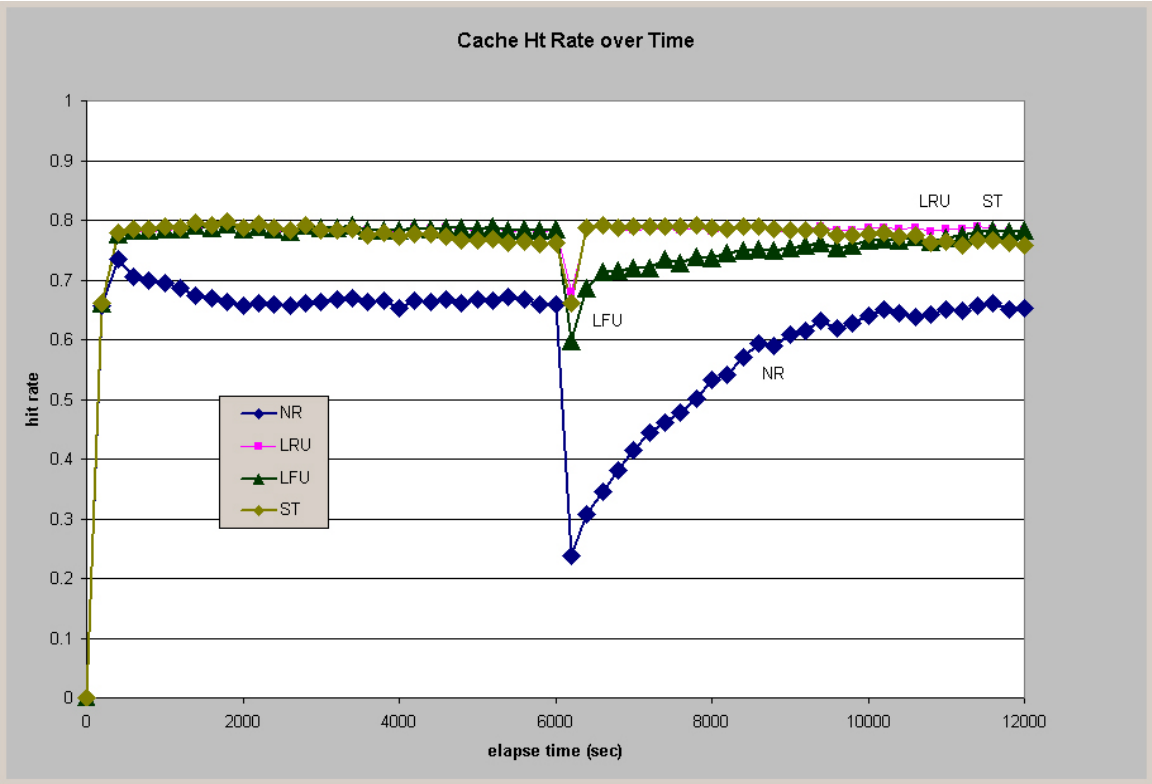

Figure 3: Hit rate comparison with user access and update pattern changes (80/10/90/10)

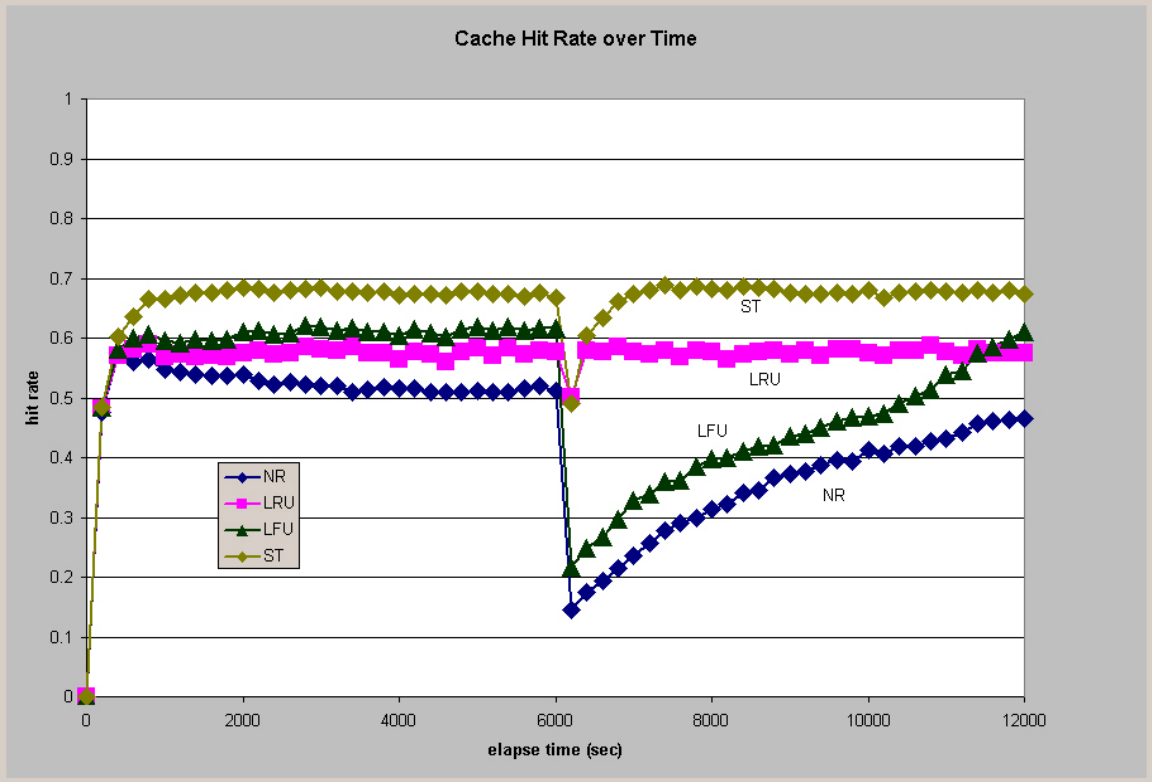

Figure 4: Hit rate comparison with user access and update pattern changes (80/20/90/10)

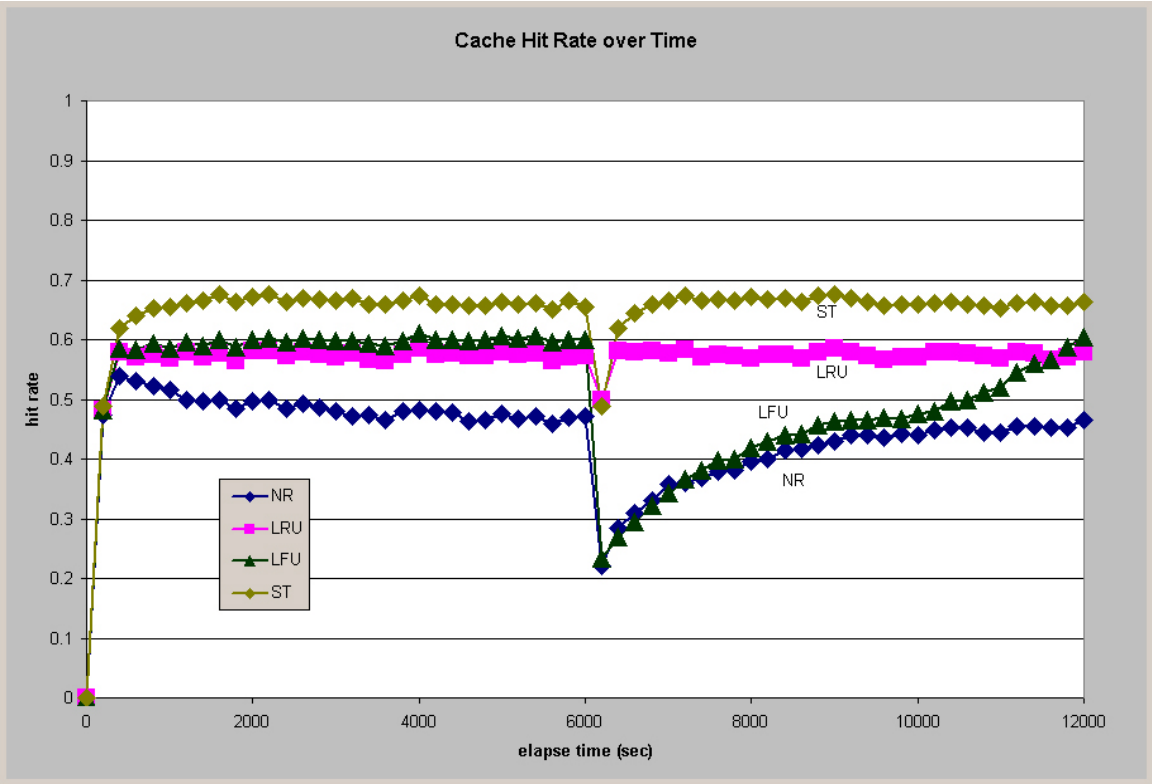

Figure 5: Hit rate comparison with user access and update pattern changes (80/20/90/30)

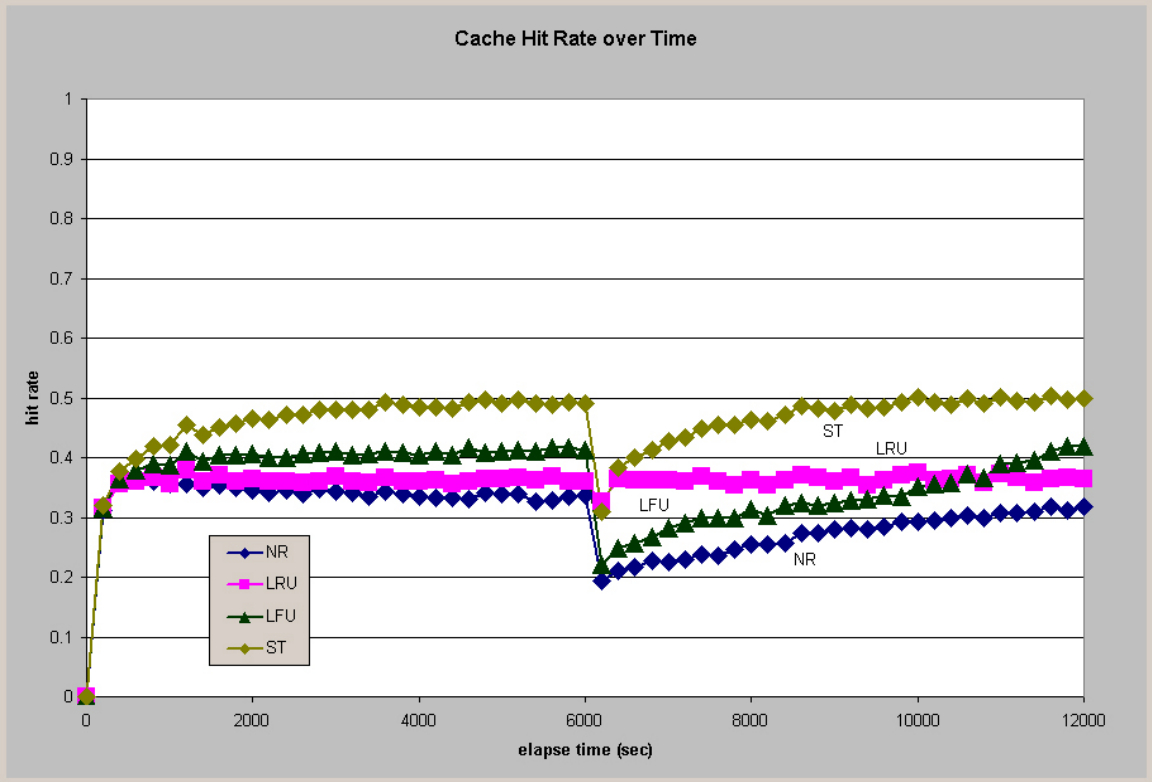

Figure 6: Hit rate comparison with user access and update pattern changes (60/20/90/10)

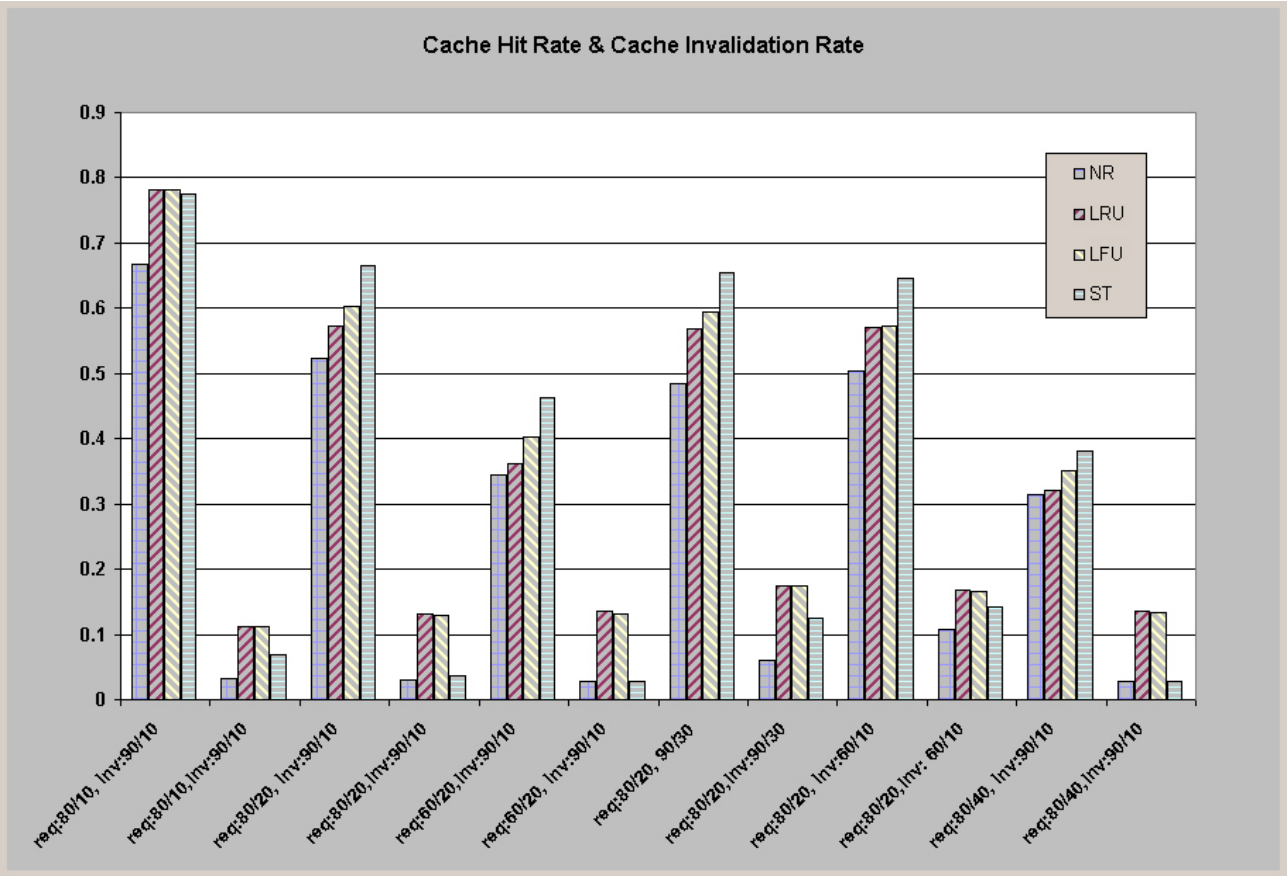

Figure 7: Comparisons of hit rates and invalidation rates

stable. As we can see in this figure, the hit rates achieved by the ST replacement strategy are better than other algorithms in all settings. In general it performs 5 to 10 percent better than the most frequently used LRU and LFU algorithms. This improvement is good, but it should not be viewed as significant by itself. However, the cache invalidation rates tuned by the ST replacement strategy are 75% lower than the LRU and LFU algorithms. This is because that LRU and LFU only track access patterns and do not account for update activities and invalidation patterns. On the other hand, ST tracks both. The combination of higher hit rates and lower invalidation rates give us the advantages of faster delivery from the caches and faster generation of the missed pages due to lower system load.

4 Database Access Tier Caching Management

Most database access in application servers is through either ODBC or JDBC. *XJDBC* is a software module developed in the scope of *CachePortal*. It interposes itself between the servlet (above) and the JDBC layer (below) to provide database access caching management for application servers. It caches query results that come from JDBC to the upper layer (servlet). It can also pre-fetch additional rows of result sets from DBMS to reduce the process

latency in the application servers. If the servlet might modify the received query result set, the cloning of query result set is required to serve multiple applications running in the application server that are using the same query result set. The purpose is to maintain the consistency between the cached result sets and the corresponding database content because the servlet might modify the received result set.

The tasks performed by XJDBC include caching and pre-fetching. The task of invalidation is carried out by the *Invalidator* and once query results are deemed as being invalid, invalidation messages are sent to XJDBC. The cache replacement and policy is carried out by the *Cache Manager*. In the current implementation, the caching priority of each query result set is calculated by the *Cache Manager* based on *response time gain-based replacement*, which is quite different from the caching management in the content page tier in the following ways:

- The size of each query result set varies greatly since it depends on query statements, table size, and selectivity. On the other hand, the size of generated dynamic content page is usually small since it is text data.
- Response time for generating dynamic content pages and for accessing database vary greatly from one request to another request. However, frontend cache servers and edge cache servers do not have access to most of the information that is needed to calculate caching priority, such as network latency, process latency by application servers and DBMSs, is not available to the frontend and edge cache managers; while the cache manager for the database access caching has access to all of this information.
- The target for cache management is to minimize response time. However, if it is not feasible due to incomplete information, the target could be set as to maximize cache hit rates.

4.1 Response Time Gain-based Cache Replacement

Web server receives user requests and hands those requests to application programs to retrieve data from DBMS. If query result sets are cached in the memory, the request response time can be improved substantially. Since there is limited cache space and there is invalidation cost associated with every query result set in the cache, we have developed *response time gain-based replacement* for dynamically adjusting the priority of objects in the database access tier. *Response time gain-based replacement* uses a metric for object "profitability" that measures what the actual benefit of replacing a certain object with another, given previously collected statistics. The caching priority of a query result set of a time period t is calculated as

$$Caching\_priority_t = \alpha \times \frac{query\_result\_set\_access\_rate \times query\_response\_time}{(invalidation\_rate + 1) \times query\_result\_set\_size} + (1-\alpha) \times Caching\_priority_{t-1}$$

where α is the temporal decay factor whose value is between 0 and 1. A value of 1 for α makes the system treat all access patterns equally, while a value of 0 makes the system consider only the access patterns during the current time period. Currently the value of α is set to 0.8.

The intuitions behind this formula are as follows:

- It estimates the average number of accesses for the query result set between any two successive invalidations. The higher is this number the more one might want to keep this query result set in the cache.
- For two query results which are accessed the same number of times between any two successive invalidations, the smaller query result set has a higher priority than the larger query result set since the smaller query result set consumes less space in the cache server.
- For two query results of the same size which are accessed the same number of times between any two successive invalidations, the query result set that requires more time to generate has a higher priority than the query result set that requires less time to generate since caching the query result set that requires more time to generate provides more response time gain.

The caching priority of each query result set is re-calculated periodically. In the current implementation, the priority is re-calculated every minute. Note that the frequency of re-calculation does have an impact on the cache hit rate. However, if data access rate is not high, re-calculating caching priority frequently does not make much difference.

When XJDBC receives query result sets from the DBMS, it simply caches all result sets if there is enough cache space. If there is not enough space for the incoming query result set and the caching priority of the incoming query result set is higher than the priority of some cached query result sets, the query result set or sets in the cache with the lowest priory will be replaced to make room for the incoming query result set. In the scenario that there is still not enough space for the incoming query even if we replace all the query result sets with lower cache priority than the incoming query result set, we calculate the weighted average of the caching priority for all the query result sets that need to be replaced:

$$\frac{Caching\_Priority(Obj1) \times Size(Obj1) + Caching\_Priority(Obj2) \times Size(Obj2).... + Caching\_Priority_{Obj_n} \times Size(Obj_n)}{Size(Obj1) + Size(Obj2) + ... + Size(Obj_n)}$$

If there exists a group of objects whose group caching priority is lower than the caching priority of the incoming query result set, we replace the group of objects as a whole with the incoming query result set.

4.2 Experiments

We have conducted experiments to evaluate our algorithm (CP) and compare its performance with LRU's and LFU's. We also measure the query response time of the same experiment setup when no any replacement strategy is used (NC). The general experimental settings are as follows:

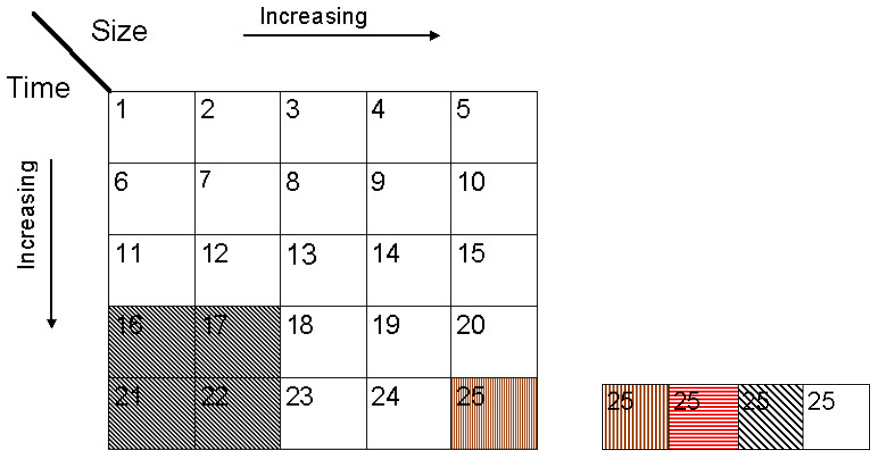

Figure 8: Database Query Result Set Access and Invalidation Patterns Used in the Experiments

- The total number of queries in the system is 5000 and the size of cache is 3000 units.
- Caching priority is maintained and re-calculated every 200 seconds.
- Database query result sets can be categorized into 25 groups as shown in Figure 8(a) by their response time (i.e. 1, 2, 3, 4, and 5 seconds from top to bottom) and query result set size (i.e. 1, 2, 4, 8, 16 units from left to right).
- The database query result sets can be further categorized into four groups by their access and invalidation patterns: (1) high request rate and high invalidation rate (marked with vertical lines); (2) high request rate and low invalidation rate (marked with diagonal line); (3) low request rate and high invalidation rate (marked with horizontal lines); and (4) low request rate and low invalidation rate (not marked).
- In Figure 8, we show two experiment setup. In the first setup (Figure 8(a)), groups 16, 17, 21, and 22 have 100 percent of queries with high request rates and low invalidation rates. Group 25 has 100 percent of queries with high request rates and high invalidation rates. The other 20 groups have 100 percent of query result sets with low request rates and low invalidation rates. In the second setup (Figure 8(b)), the query result sets with different request rates and invalidation rates are uniformly distributed.
- In each experiment, the cache is empty to start with and each experiment lasts for 6,000 seconds for the first request and invalidation pattern as shown in Figure 8(a) and another 6,000 seconds for the second request and invalidation pattern as shown in Figure 8(b).
- The rate of queries to the database is 40 requests per second and the invalidation rate of the query result sets is 10 per second for all experiments.

Figures 9 to 12 illustrate the experimental results. We summarize our observations as follows:

- Figures 9 shows the average response time for both experiment setup. We tested four algorithms; namely, NC (i.e. no caching applied), LRU, LFU, and CP (i.e. response time gain-based replacement). The average response time for the database access without any caching applied is around 4.3 seconds and CP can reduce that average response time to around 1 second, which is better than LFU's 2 seconds and LRU's 2.7 seconds.

- Figures 10 and Figure 11 show the cache hit rates and invalidation rates for the four algorithms. It is clear that CP has both the highest cache hit rate and the lowest cache invalidation rate. These characteristics yield two advantages: (1) fast response time by higher cache hit rates; and (2) lower load for the application server and DBMS since less content pages need to be re-generated due to lower cache invalidation rates.

- When we look at the right side of Figures 9, 10, and 11, the CP algorithm still yields the best performance in response time, cache hit rates, and invalidation rates. However, the benefits are not so obvious. This is because there is no clear database query result access and invalidation pattern.

- Figure 12 shows the average response time for the same experiment setup except that we change the query response time distributions from 1, 2, 3, 4, and 5 seconds to 1, 2, 4, 8, and 16 seconds. The average response time for the database access without any caching applied increases to 11.5 seconds and CP can reduce it to 2 second, which is much better than LFU's 4.5 seconds and LRU's 6.7 seconds. This experiment shows that when the patterns are strong and the difference in response times for queries increases, the benefits of CP can further differentiate it from other algorithms.

5 Related Work

Applying caching solutions for Web applications and content distribution has received a lot of attentions in the Web and database communities, including many recent tutorials taught at most major database conferences. To ensure the freshness of the cached database-driven web pages, database changes must be monitored so that the cached pages that are impacted by the content changes can be invalidated or refreshed. It is not straightforward to create a mapping between the data and the corresponding Web pages *automatically*. One solution is to assume that the applications will be modified to provide explicit mappings [22, 23, 24, 25]. This however may not be desirable when applications change often and a looser coexistence between data sources and caching providers are desirable to ensure application portability. *CachePortal* [12, 13, 14] is a Web acceleration solution, developed at NEC Research Laboratories in San Jose, which addresses this issue by not requiring any explicit mappings. *CachePortal* enables dynamic content caching by deriving the relationships between cached pages and database access via the *sniffer*; and by intelligently monitoring database changes to "eject" dirty pages from caches via the *invalidator*.

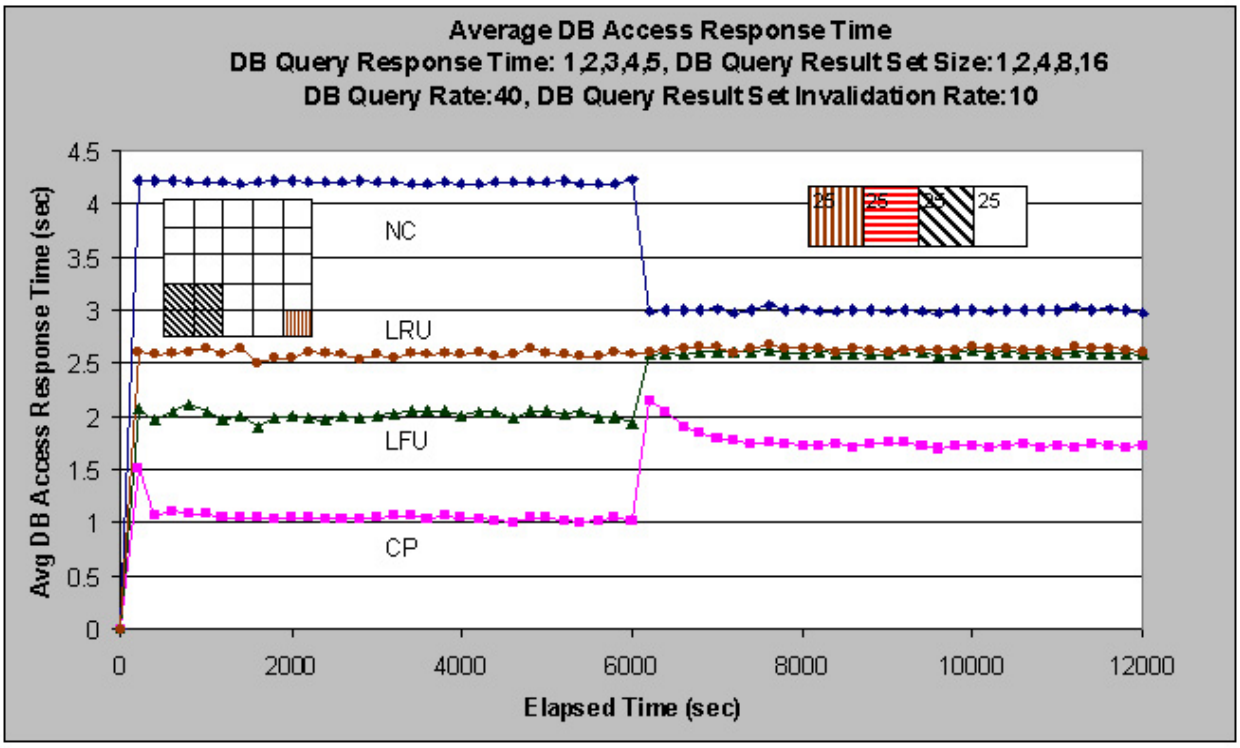

Figure 9: Measurement of Average Response Time

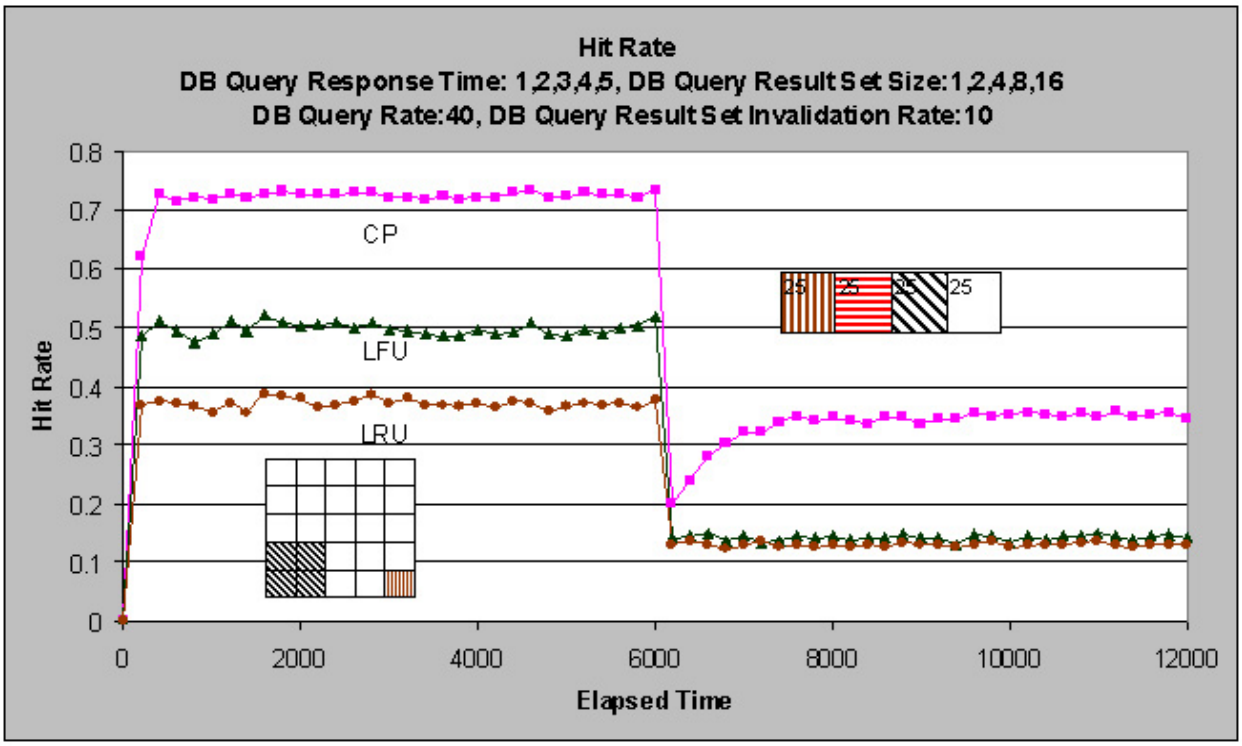

Figure 10: Measurement of Cache Hit Rates

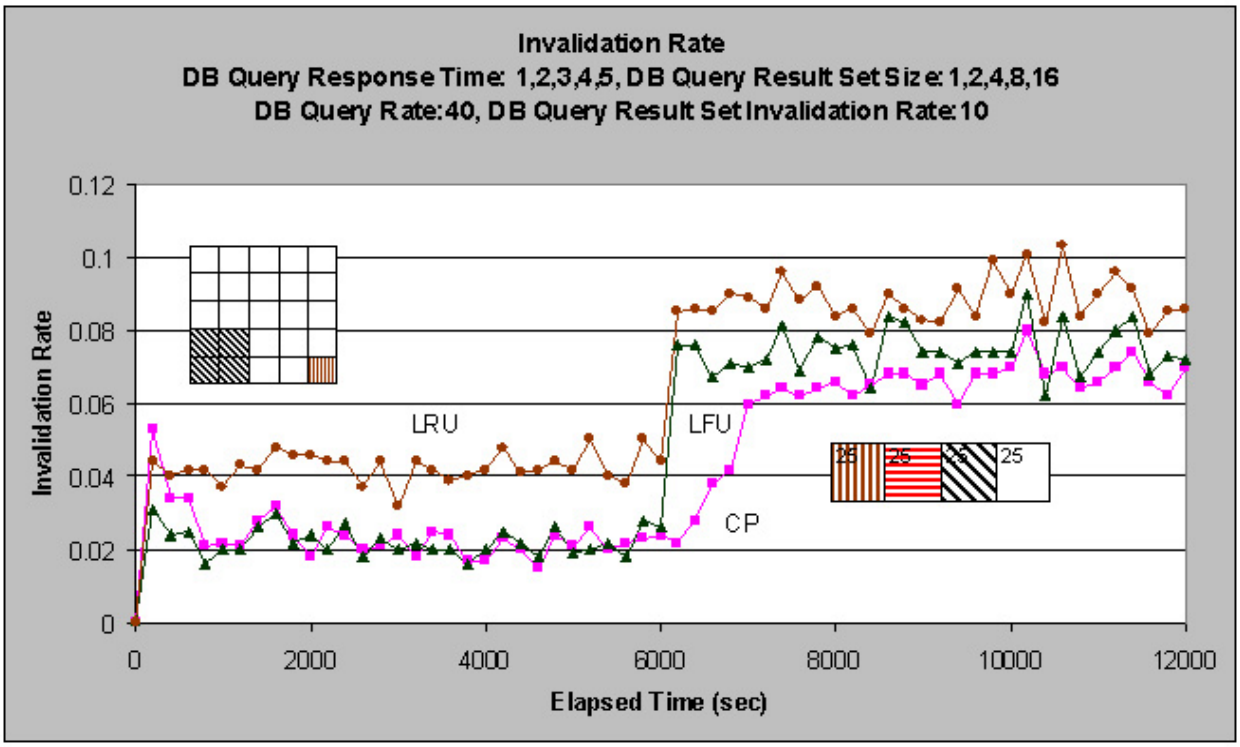

Figure 11: Measurement of Cache Invalidation Rates

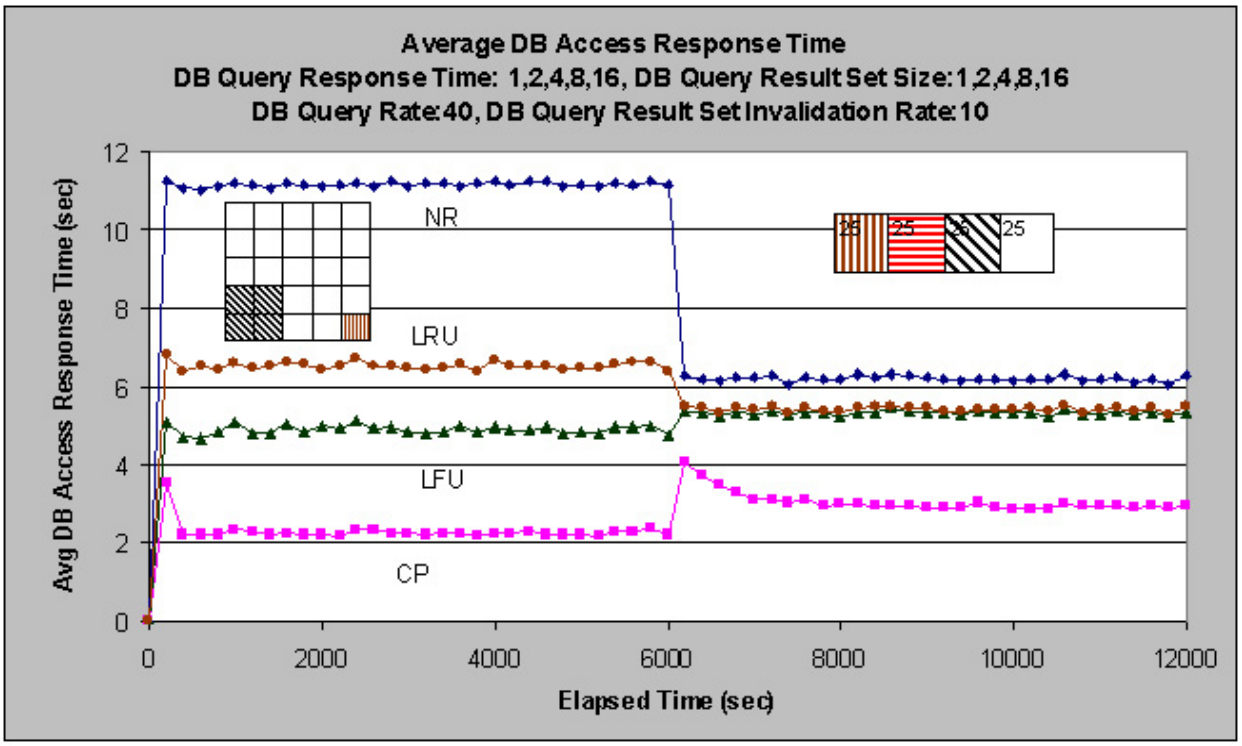

Figure 12: Measurement of Average Response Time

Issues related to caching of dynamic data have also received significant attention recently [33, 34]. Dynamai [22] from Persistence Software is one of the first dynamic caching solution that is available as a product. However, Dynamai relies on proprietary software for both database and application server components. Thus it cannot be easily incorporated in existing e-commerce framework. Challenger et al. [23, 24, 25] at IBM Research have developed a scalable and highly available system for serving dynamic data over the Web. In fact, the IBM system was used at Olympics 2000 to post sport event results on the Web in timely manner. This system utilizes database triggers for generating update events as well as intimately relies on the semantics of the application to map database update events to appropriate Web pages.

Other related works include [5, 6], where authors propose a diffusion-based caching protocol that achieves load-balancing, [35] which uses meta-information in the cache-hierarchy to improve the hit ratio of the caches, [36] which evaluates the performance of traditional cache hierarchies and provides design principles for scalable cache systems, and [7] which highlights the fact that static client-to-server assignment may not perform well compared to dynamic server assignment or selection.

SPREAD[37], a system for automated content distribution is an architecture which uses a hybrid of *client validation*, *server invalidation*, and *replication* to maintain consistency across servers. Note that the work in [37] focuses on static content and describes techniques to synchronize static content, which gets updated periodically, across Web servers. Therefore, in a sense, the invalidation messages travel horizontally across Web servers. Other works which study the effects of *invalidation* on caching performance are [38, 39]. Consequently, there has been various cache consistency protocol proposals which rely heavily on *invalidation* [37, 40, 26]. In our work, however, we concentrate on the updates of data in databases, which are by design not visible to the Web servers. Therefore, we introduce a *vertical* invalidation concept, where invalidation messages travel from database servers and Web servers to the front-end and edge cache servers as well as JDBC database access layer caches.

6 Concluding Remarks

To improve system response time, many e-commerce Web sites deploy caching solutions for acceleration of content delivery. There are multiple tiers in the content delivery infrastructure where cache servers can be deployed, including (1) data caching (in data centers), (2) content page caching (in edge or frontend caches), (3) database query result set caching (between application servers and DBMS). In this paper, we focus on issues in cache management in multiple tiers of content distribution infrastructure. In the scope of NEC's CachePortal project, we observe that the caching management for content pages at frontend- and edge cache tier and for database access acceleration tier are quite different; mainly due to information available for the content page tier is less complete. We have conducted extensive experiments to evaluate our proposed solutions to cache management. The experimental results strongly

show the usefulness of our technology in improving overall system performance. The combined effect of caching at multiple tiers ramains an area of future investigation.

References

[1] Zona Research. http://www.zonaresearch.com/.

[2] Akamai Technology. *Information available at* `http://www.akamai.com/html/sv/code.html`.

[3] Digital Island, Ltd. *Information available at* `http://www.digitalisland.com/`.

[4] Oracle Corp. http://www.oracle.com/.

[5] A. Heddaya and S. Mirdad. WebWave: Globally Load Balanced Fully Distributed Caching of Hot Published Documents. In *Proceedings of the 1997 IEEE International Conference on Distributed Computing and Sytems*, 1997.

[6] A. Heddaya, S. Mirdad, and D. Yates. Diffusion-based Caching: WebWave. In *Proceedings of the 1997 NLANR Web Caching Workshop*, 1997.

[7] R.L. Carter and M.E. Crovella. On the network impact of dynamic server selection. *Computer Networks*, 31(23-24):2529–2558, 1999.

[8] Wen-Syan Li, Wang-Pin Hsiung, Dmitri V. Kalashnikov, Radu Sion, Oliver Po, Divyakant Agrawal, and K. Selçuk Candan. Issues and Evaluations of Caching Solutions for Web Application Acceleration. In *Proceedings of the 28th Very Large Data Bases Conference*, Hongkong, China, August 2002.

[9] Oracle9i data cache. http://www.oracle.com/ip/deploy/ias/caching/index.html?database_caching.html.

[10] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier Database Caching for e-Business. In *Proceedings of 2002 ACM SIGMOD Conference*, Madison, Wisconsin, USA, June 2002.

[11] Oracle9i web cache. http://www.oracle.com/ip/deploy/ias/caching/index.html?web_caching.html.

[12] K. Seluk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling Dynamic Content Caching for Database-Driven Web Sites. In *Proceedings of the 2001 ACM SIGMOD Conference*, Santa Barbara, CA, USA, May 2001. ACM.

[13] Wen-Syan Li, K. Seluk Candan, Wang-Pin Hsiung, Oliver Po, Divyakant Agrawal, Qiong Luo Wei-Kuang Wayne Huang, Yusuf Akca, and Cemal Yilmaz. Cache Portal: Technology for Accelerating Database-driven E-commerce Web Sites. In *Proceedings of the 2001 VLDB Conference*, Roma, Italy, September 2001.

[14] K. Selcuk Candan, Divyakant Agrawal, Wen-Syan Li, Oliver Po, and Wang-Pin Hsiung. View Invalidation for Dynamic Content Caching in Multitiered Architectures . In *Proceedings of the 28th Very Large Data Bases Conference*, Hongkong, China, August 2002.

[15] P. Deolasee and A. Katkar and A. Panchbudhe and K. Ramamritham and P. Shenoy. Adaptive Push-Pull: Dissemination of Dynamic Web Data. In *the Proceedings of the 10th WWW Confernece*, Hong Kong, China, May 2001.

[16] Anoop Ninan and Purushottam Kulkarni and Prashant Shenoy and Krithi Ramamritham and Renu Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. Honolulu, Hawaii, USA, May 2002.

[17] Anindya Datta, Kaushik Dutta, Helen M. Thomas, Debra E. VanderMeer, Suresha, and Krithi Ramamritham. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In *Proceedings of 2002 ACM SIGMOD Conference*, Madison, Wisconsin, USA, June 2002.

[18] Anindya Datta, Kaushik Dutta, Krithi Ramamritham, Helen Thomas, and Debra VanderMeer. Dynamic Content Acceleration: A Caching Solution to Enable Scalable Dynamic Web Page Generation . In *Proceedings of the 2001 ACM SIGMOD Conference*, Santa Barbara, CA, USA, May 2001. ACM.

[19] C. Mohan. Application Servers: Born-Again TP Monitors for the Web? (Panel Abstract). In *Proceedings of the 2001 ACM SIGMOD Conference*, Santa Barbara, CA, USA, August 2001.

[20] C. Mohan. Caching Technologies for Web Applications. In *Proceedings of the 2001 VLDB Conference*, Roma, Italy, September 2001.

[21] Mitch Cherniack, Michael J. Franklin, and Stanley B. Zdonik. Data Management for Pervasive Computing. In *Proceedings of the 2001 VLDB Conference*, Roma, Italy, September 2001.

[22] Persistent Software Systems Inc. http://www.dynamai.com/.

[23] Jim Challenger, Paul Dantzig, and Arun Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of ACM/IEEE Supercomputing'98*, Orlando, Florida, November 1998.

[24] Jim Challenger, Arun Iyengar, and Paul Dantzig. Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of the IEEE INFOCOM'99*, New York, New York, March 1999. IEEE.

[25] Eric Levy, Arun Iyengar, Junehwa Song, and Daniel Dias. Design and Performance of a Web Server Accelerator. In *Proceedings of the IEEE INFOCOM'99*, New York, New York, March 1999. IEEE.

[26] D. Li, P. Cao, and M. Dahlin. WCIP: Web Cache Invalidation Protocol, 2000. http://www.ietf.org/internet-drafts/draft-danli-wrec-wcip-00.txt.

[27] Jian Yin and Lorenzo Alvisi and Mike Dahlin and Arun Lyenger. Engineering Server-Driven Consistency for Large Scale Dynamic Web Services. In *the Proceedings of the 10th WWW Confernece*, HongKong, China, May 2001.

[28] M. Abrams and C.R. Standbridge and G.Abdulla and S. Williams and E.A. Fox. Caching Proxies: Limitations and Potentials. In *the Proceedings of the 4th WWW Confernece*, Boston, MA, USA, December 1995.

[29] S. Williams and M. Abrams and C.R. Standbridge and G.Abdulla and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the 1996 ACM SIGCOMM Conference*, Stanford, CA, USA, August 1996.

[30] Junho Shim and Peter Scheuermann and Radek Vingralek. Proxy Cache Design: Algorithms, Implementation and Performance. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):549–562, 1999.

[31] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the Symposium on Internet Technologies and Systems*, Monterey, California, USA, December 1997.

[32] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *World Wide Web*, 2(1-2), 1999.

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

[34] Ben Smith, Anurag Acharya, Tao Yang, and Huican Zhu. Exploiting Result Equivalence in Caching Dynamic Web Content. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1999.

[35] M.R. Korupolu anf M. Dahlin. Coordinated Placement and Replacement for Large-Scale Distributed Caches. In *Proceedings of the 1999 IEEE Workshop on Internet Applications*, 1999.

[36] Renu Tewari and Michael Dahlin and Harrick M. Vin and Jonathan S. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, 1999.

[37] P.Rodriguez and S.Sibal. Spread: Scaleable platform for reliable and efficient automated distribution. In *Proceedings of the 9th World-Wide Web Conference*, pages 33–49, Amsterdam, The Netherlands, June 2000.

[38] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World Wide Web. *IEEE Transactions on Computers*, 47(4), 1998.

[39] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of 1996 USENIX Technical Conference*, pages 141–151, San Diego, CA, USA, January 1996.

[40] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of the ACM SIGCOMM'99 Conference*, Boston, MA, USA, September 1999.