Challenges and Practices in Deploying Web Acceleration Solutions for Distributed Enterprise Systems

Wen-Syan Li Wang-Pin Hsiung Oliver Po Koji Hino K. Selçuk Candan Divyakant Agrawal

NEC Laboratories America, Inc.
10080 North Wolfe Road, Suite SW3-350, Cupertino, California 95014, USA

ABSTRACT

For most Web-based applications, contents are created dynamically based on the current state of a business, such as product prices and inventory, stored in database systems. These applications demand personalized content and track user behavior while maintaining application integrity. Many of such practices are not compatible with Web acceleration solutions. Consequently, although many web acceleration solutions have shown promising performance improvement and scalability, architecting and engineering distributed enterprise Web applications to utilize available content delivery networks remain a challenge. In this paper, we examine the challenge to accelerate J2EE-based enterprise web applications. We list obstacles and recommend some practices to transform typical database-driven J2EE applications to cache friendly Web applications where Web acceleration solutions can be applied. Furthermore, such transformation should be done without modification to the underlying application business logic and without sacrificing functions that are essential to e-commerce. We take the J2EE reference software, the Java PetStore, as a case study. By using the proposed guideline, we are able to cache more than 90% of the content in the PetStore and scale up the Web site more than 20 times.

Categories and Subject Descriptors

H.4 [**Information Systems**]: Information Systems Applications; D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Performance, Reliability, Experimentation

Keywords

J2EE, dynamic content, application server, edge server, fragment, web acceleration, reliability, scalability

1. INTRODUCTION

For many e-commerce applications, Web pages are created dynamically based on the current state of a business, such as product prices and inventory, stored in database systems. This characteristic requires e-commerce Web sites to deploy Web servers, application servers, and database management system (DBMS) to generate and serve user requested content dynamically. When the Web server receives a request for dynamic content, it forwards the request to the application server along with its request parameters (typically included in the URL string). The Web server communicates with the application server using URL strings and cookie information, which is used for customization. When the application server receives such a request from the Web server, it may query the underlying databases to extract the relevant information needed to dynamically generate the requested page.

To improve the response time, one option is to build a high performance Web site by improving network and server capacity by deploying a state of the art IT infrastructure. However, without the deployment of dynamic content caching solutions and content delivery networks (CDN), dynamic contents are generated on demand. In this case, all delivered Web pages are generated based on the current business state in the source database.

To improve scalability and performance, one solution is to deploy network-wide caches so that a large fraction of requests can be served remotely rather than all of them being served from the origin Web site. This solution has the advantage of serving users via caches closer to them and reducing the traffic to the Web sites, reducing network latency, and providing faster response times. Many CDN services [1] provide Web acceleration services. A study in [2] shows that CDN indeed has significant performance impact. However, for many e-commerce applications, content is created dynamically based on the current state of a business, such as product prices and inventory, rather than static information. Therefore, content delivery by most CDNs is limited to handling static portions of the pages and media objects, rather than the full spectrum of dynamic content that constitutes the bulk of the e-commerce Web sites.

Wide-area database replication technologies and the availability of data centers allow database copies to be distributed across the network. This requires a complete e-commerce web site suite (i.e., Web servers, application servers, and DBMS) to be distributed along with the database replicas. A major advantage of this approach is, like the caches, the possibility of serving dynamic content from a location close to the users, reducing network latency.

Many web acceleration solutions, such as [3, 4, 5, 6, 7, 8], have shown promising performance improvement and scalability. Li et al. in [9] provide evaluations of architectural designs and various implementation practices for database-driven Web sites. In [10], Li et al. further analyze the factors that have impacts on the performance and scalability of Web applications and outline a road map for Web acceleration for dynamic content. However, with the necessary requirements for enterprise Web applications, such as personalization and user behavior tracking, architecting and engineering distributed enterprise Web applications based on available Web acceleration solutions remains a challenge.

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

In this paper, we examine the challenges and practices to accelerate J2EE-based enterprise web applications. We list obstacles and recommend some practices to transform a typical database-driven J2EE applications to a cache friendly Web application where Web acceleration solutions can be applied. Furthermore, such transformation can be done without modification to the underlying application business logic and without sacrificing functions that are essential to e-commerce (e.g., personalization and user behavior tracking) and application security (e.g., URL encoding). We take the Sun's Java PetStore [11], a J2EE reference software provided by the J2EE Blueprints program, as a case study. The PetStore defines the application programming model for the J2EE platform. It provides practice guidelines and architectural recommendations for real-world application scenarios to enable developers to build portable, scalable, and robust applications using J2EE technology. To maintain the integrity of the content, the PetStore generates all user responses based on database content and session objects are used to track user behavior. We found out that most of the content in the PetStore application is not cacheable. As a result, no acceleration solution can be applied. We take the PetStore and apply cache friendly and deployment practice guidelines. By incorporating these changes, we are able to cache more than 90% of the content in the PetStore while maintaining system integrity. We further deploy NEC's CachePortal II technology [12] to accelerate the PetStore. The experiment evaluations show that the system can be scaled up to more than 20 times.

The rest of this paper is organized as follows. In Section 2 we describe a typical system architecture for database-driven Web application. In Section 3 we address challenges to deploying acceleration solutions to distributed enterprise systems and our approaches to overcome these obstacles to enable dynamic content caching. In Section 4, we use the Java PetStore as a case study to validate our proposed guideline. In Section 5, we show how the performance and scalability can be dramatically improved. In Section 6 we summarize related work. In Section 7 we give our conclusion.

2. ARCHITECTURE OF DISTRIBUTED ENTERPRISE SYSTEMS

Note 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 (shown in Figure 1) consists of the following components:

1. A database management system (DBMS) to store, maintain, and retrieve all necessary data related to the business state.

2. An application server (AS) that incorporates all the necessary rules and business logic to interpret the data and information stored in the database. AS receives user requests for HTML pages and depending upon the nature of a request may need to access the DBMS to generate the dynamic components of the HTML page.

3. A Web server (WS) which receives user requests and delivers dynamically generated Web pages.

4. A cache server in front of the Web server (i.e., front end cache) or at the edge (i.e., edge cache) to cache content. To serve dynamically assembled pages, such as JSP or ASP pages, an edge application server may be deployed to offload the application server at the origin Web site.

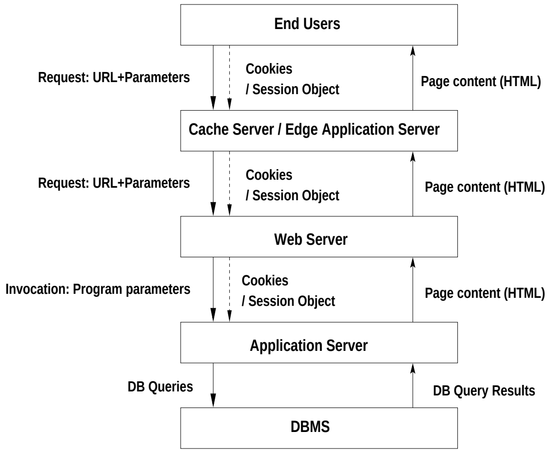


Figure 1: Architecture of a Database-Driven E-Commerce Site

When a user accesses the Web site, the request and its associated parameters, such as the product name and model number, and cookie and session object information for customization, are passed to the application server. The application server performs necessary computation to identify what kind of data it needs from the database or file system, or external data sources. Then the application server sends appropriate queries to the database or other sources. After the database returns the query results to the application server, the application uses these to prepare a Web page and passes it to the Web server, which then sends it to the user. Note that most cache servers are cookie-aware in the sense that they can serve cached pages based on matching a combination of URL string and cookie information. However, the session object information is only available to the application.

Since cache servers, Web servers, application servers, and databases are independent components, it requires a coordination mechanism to ensure that database content changes are reflected to the caches. In [4], we first developed a dynamic content caching and invalidation framework for accelerating database-driven e-commerce Web sites. The technology enables dynamic content caching by

- automatically deriving the relationships between cached pages and database contents (i.e., *URL and query mapping*); and
- intelligently monitoring database changes to ”eject” (i.e., delete) stale pages from caches.

In most existing work, the relationships between Web pages and the underlying data are specified manually. In contrast, CachePortal II[12] features a *sniffer* to automatically generate the URL and query mapping. The sniffer sits between the Web server and the application as well as between the application server and the DBMS. The sniffer intercepts the user requests and inserts a special tag to associate with the requests. The tag is then used to track the actions of each request at the application server and the DBMS. Because the cache server uniquely identified cached content based on URL and cookie information, the URL in the map includes cookie information. Also in most existing work, the invalidation checking is implemented using database trigger functions while the *invalidator* presented in [13] is implemented as an external software component based on the incremental view maintenance techniques, which does not add substantial load to the underlying DBMS.

As we can see, the framework of enabling dynamic content caching to accelerate Web applications truly relies on *automated construction of the URL and query mapping* since the accurate invalidation can not be done without the map. In the next section, we describe some obstacles to construction of the URL and query mapping.

3. OBSTACLES FOR ENABLING DYNAMIC CONTENT CACHING

We have investigated a set of J2EE e-commerce application software. In this section, we discuss the challenges and obstacles for constructing URL and query mapping.

3.1 Session Object

An application may maintain data in the session object that may influence the construction of a page. For example, a session ID based on Ethernet card ID may be used to identify user A logging in from PC A. When user A gives his password to user B to access the account of user A from another PC concurrently, the application server can use the information in the session object to reject the access by user B.

The session object is also frequently used to hide parameters since how to interpret the semantics of the session object is only available at the application server. For example, the Java PetStore uses the session object to store shopping cart information for individual user. In this example, the session object is a better option than the cookie since the session is designed to have a short TTL (i.e., less than a hour).

Since the session data spans different requests, cache servers do not know if these objects will have any impact on safe caching of a page. By default, requested pages will be marked non-cacheable if the session object contains any data objects.

To solve this problem, we develop a scheme for enabling dynamic content caching in the presentation of a session object. The procedure of our proposed scheme is as follows:

- The API of converting URL, cookie, and session object into a hash key is made available at the application server for the cache server to access or the API is replicated at the cache server.
- When the request first comes to the cache server, the cache server forwards it to the application server after checking through the API to see if there is a match of cached pages based on the hash key.
- The application server processes the page and determines if the page is cacheable or not (by CachePortal's SQL parser) and returns the page to the cache server with a special tag indicating if the page is cacheable.
- If the page is cacheable, it is cached to serve further requests.
- If the database content changes occur and the invalidator detects the cached page needs to be invalidated, the invalidation message is sent to the cache server.

Note that with this scheme, the cached pages are served to only the original requesting users rather than other users even if the page contents are identical. The impact of this solution to the cache hit rate will be discussed later.

3.2 Cache Object

Applications may maintain data obtained from data sources in data objects and provide them to the subsequent requests to the same data content. For example, when a user login to an on line banking application Web site, all information associated with the user may be retrieved to the application using a single SQL command. As a result, the subsequent user requested pages, such as account balance, history, and mailing address, do not result in any queries to the database. As a result, a sniffer implemented at the JDBC layer will see only one SQL command rather than multiple SQL commands that correspond to the subsequent requests.

To solve this problem, we allow the users to tag the application programs with additional statement to explicitly declare such intent. For example, when the first request results in the construction of the cached data object, the SQL statement and the object ID is stored in the map. Subsequent requests will be tagged with additional statement to declare usage of such cached data objects. Therefore, the URL strings of the following requests and the cached data object ID can be associated and stored in the map for invalidation process. Note that the above tagging statement can be added to the application programs systematically without affecting the application and business logic.

Construction of the URL and SQL query map in the presentation of cached data objects is not a major issue for the application server vendors since the requests to the cached data objects are monitored by the application server. The map described above can be constructed without adding additional statements to the application programs.

3.3 Unsupported Data Source

Any content that is used for the construction of a page must come from data sources that are being sniffed to detect content change. This is essential, as without this we have no way of invalidating a page when the underlying data changes. For example, a page may be generated based on data stored in the modern database as well as some files stored in content management software. If the content management software does not provide any invalidation scheme of the content it supports, we have to mark the pages as non-cacheable. The other option is to utilize fragments to cache the partial page. For example, we can mark the portion of a page that is generated using unsupported data source non-cacheable while marking other portions cacheable.

CachePortal provides a framework for developing custom sniffers and invalidators. Custom sniffers and invalidators are required to support caching of pages where all or part of the content for the page comes from data sources not supported by CachePortal.

3.4 URL Encoding

Requests may contain parameters that are encoded/encrypted with session id or some other key. For example, two users may login to a Web site and the session ID based on Ethernet card ID on the two machines where the users login are sent to the application server. Then the application may generate two pages with the identical content but all the links are encoded with the session ID from the machine. When the links are clicked, the request with encoded URL is sent to the application server. The application then checks if the session ID (i.e., Ethernet card ID) sent with the request match with the session ID encoded in the URL string. If it does not match, the application server will not generate the requested page. This practice of URL encoding allows the Web site to limit users' access from certain machines. Furthermore, when the encoded URLs generated for a particular user on a particular machine are given to another user, the content will not be returned.

The URL encoding scheme provides option for applications to add some security to the system. However, it greatly reduces the scope of dynamic content caching. With URL encoding, almost all requests will appear as a different request to cache server, even though the content returned is the same. We propose two solutions for this problem:

- Limiting usage of URL encoding: we have investigated several real world applications. We found out that many Web applications over-use the URL encoding scheme in their applications. For example, the URL encoding scheme should be applied on only transaction related operations while URLs for catalog pages should not be encoded so that caching solution can be applied.

- Currently, the encoding logic is embedded in the application server. To enable dynamic content caching of encoded URL requests (for those catalog pages), there are two options. The first option is to define an API for the cache server to access and decode the URLs. It will result in additional network latency. The second option is to replicate the decoding component at the cache server.

3.5 Use of Cookie

Most cache servers are cookie enabled; which means that the cache servers will use a combination of the URL string and cookie information to uniquely identify a page when the cookie information is presented. However, once a cookie is set at the beginning of the user session with the application server, the cookie will remain active until it is destroyed. During the entire period, the cookie information is used to identify pages even for cacheable catalog pages. As a result, all the users may access the same catalog page, but the cache server will view the same catalog pages accessed by two users different since the combination of the URL string and cookie information differentiate them although the cookie information has no impact on the generation of the catalog page. To solve this problem, CachePortal provides a tool, `Page Differentiator`, for the system administrator to configure the cache servers a list of URLs for which cookie information needs to be considered and a list of URLs for which cookie information does not need to be considered. The detailed information of `Page Differentiator` is given later.

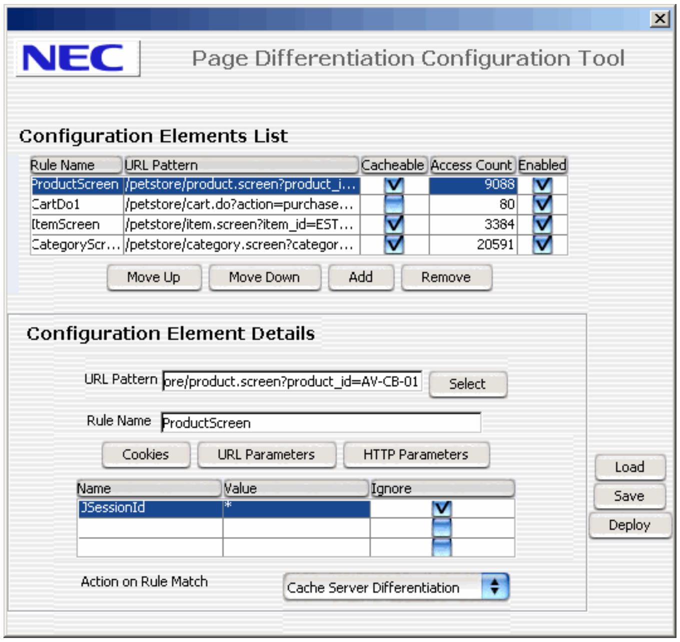
3.6 User Behavior Logging and Tracking

User behavior logging and tracking are essential to enterprise system integrity and business intelligence. For example,

- an on-line banking application needs to log all user interaction and values that the user enters through the screen so that such operation can be recovered if any error occurs; and

- an e-commerce Web site may track the items that the users browse to identify the categories of items the users are interested for future targeted marketing activities.

For the real world applications that we have investigated, we found that most applications do deploy logging and tracking functions and use a dedicated database table to store such information. All the logging and tracking information is written to the database table before or on returning the requested pages to the users. As a result, when logging or tracking is applied, even if generating a page requires only a read operation to the DBMS, it will still result in both read and write operations on the DBMS making the request non-cacheable.



Page Differentiation Configuration Tool

Configuration Elements List

Rule Name	URL Pattern	Cacheable	Access Count	Enabled
ProductScreen	/petstore/product.screen?product_i...	☑	9088	☑
CartDo1	/petstore/cart.do?action=purchase...	□	80	☑
ItemScreen	/petstore/item.screen?item_id=EST...	☑	3384	☑
CategoryScr...	/petstore/category.screen?categor...	☑	20591	☑

Configuration Element Details

URL Pattern: ore/product.screen?product_id=AV-CB-01

Rule Name: ProductScreen

Name	Value	Ignore
JSessionId	*	☑
		□
		□

Action on Rule Match: Cache Server Differentiation

Figure 2: Page Differentiator Configuration Tools

All database operations, such as logging, that do not have an impact on freshness of a cached page should be avoided in the request thread for the reasons of (1) better response time for the request pages; and (2) confusing the caching solution to make correct judgment on whether or not a requested page is cacheable.

To overcome this obstacle, we propose the following solutions:

- CachePortal provides configuration tool to allow the system administrator to specify the database tables that are used for logging and tracking. CachePortal will then ignore the SQL statement and read/write operations on such tables;

- Logging and tracking operations are done in a separate thread apart from the request thread; and

- Moving the logging and tracking function to the cache servers. For example, CachePortal Cache Manager and monitor tool are able to log all user specified parameters as well as all performance related statistics, which are more comprehensive than most of the information collected in the logging and tracking functions.

3.7 Page Differentiation

Every request from a client is uniquely identified based on the URL, cookie, HTTP header, session objects, and (if applicable) post data. If these parameters are identical to another request that led to a cached page, the page cached in the cache server is returned. However, if any of these parameters is different for any reason, the request is forwarded to application server for processing. However, in some cases, although the request parameters are different, they may actually be referring to the same page. This may happen because of two reasons:

- The request contains a parameter that has no material effect on the response sent. For example, if a different cookie is set for every user, irrespective of whether the user has actually logged in or not, the request will appear as a different request to the cache server, although the content returned is the same.

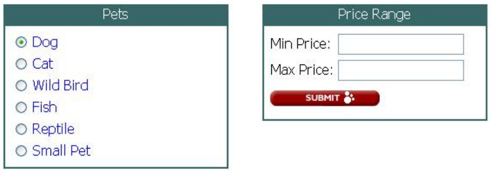


Figure 3: Cache Unfriendly Interface Design

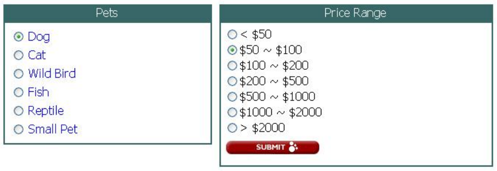


Figure 4: Cache friendly Interface Design

- The request contains parameters that are encoded/encrypted with session id or some other key. Again, the request will appear as a different request to the cache server, although the content returned is the same.

CachePortal provides two different mechanisms to deal with each of these situations. The difference between the two cases arises from the fact that the information needed to interpret the request parameters is not available at the cache server, but available at the application server. In addition to having an API at the application server for the cache server to access and provide replica of such information (or programs) at the cache server, the other solution is to use the tool `Page Differentiator` to configure the cache servers to enable caching for more dynamic pages that are really cacheable, such as the catalog.

Figure 2 shows a window dump of using `Page Differentiator` to configure a cache server to cache or not to cache pages for the Java PetStore software. On top of the window, we see that there are multiple rules for various URL string patterns. For example, in the Java PetStore, the session object is set when users login the system. When the users come to the Java PetStore site but do not login, neither the cookie nor the session object are set. However, the session object information is not used when users navigate catalog pages and item information pages. We can configure the cache server to ignore the session object, $JSessionId$, when the patterns of URL strings include `product.screen`, `item.screen`, and `category.screen`. The window also forces the cache server not to parse the pages with the URL string with the patterns like `cart.do` and `purchase`. The lower part of the window shows the details of the configuration.

3.8 Post versus Get

As a convention, HTTP POST is used when the request is idempotent (acting as if used only once, even if used multiple times) and when the request has a permanent effect on the outside world. When the intention of the request is to only obtain information, HTTP GET should be used. HTTP POST is also used when the request parameters are longer than that supported by HTTP GET.

Since the request parameters using HTTP POST are not shown as a part of URL string, POST may sometimes be used just to hide the request parameters. Unlike cookies, the HTTP POST message is not parsed by cache servers. However, the request parameters may be used by the application servers to generate page content. For example, a Web site may use cookie information to identify the origin of users (i.e., Japan or US) and use HTTP POST messages to send the product list in users' shopping carts. In this example, the users with the identical URL string and cookie information may be given different page content. Since most cache servers do not parse HTTP POST messages, the user requests using HTTP POST are marked as non-cacheable.

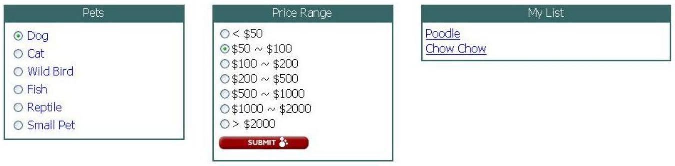


Figure 5: Interface Design with Personalization

We have enhanced the Squid cache server by adding POST message parsing capability. Thus, in our system, POST messages are parsed and used in conjunction with URL string and cookie to uniquely identify users' requested content.

3.9 Cache Friendly GUI

To utilize caching, careful and intelligent design of cache friendly Web sites is essential to scalability of a Web site using any discussed approach and configuration. In Figure 3, we show a non-cache friendly Web site design. Using this interface, users can issue unlimited number of query types to the application server and the hit ratio will be low since the probability that two users issue the same query is extremely low. On the other hand, the Web site design shown in Figure 4 is a cache friendly Web site interface since there will be only 42 different pages that can be generated (i.e., 6×7). The Web site can generate all the possible 42 pages in advance and store them at the cache server while deploying `CachePortal` technology to invalidate pages in the cache that are impacted by database content changes. The most desirable cache friendly Web access interface design would be similar to that in Figure 4. Its default query interface is cache friendly and is used by most of the users while supporting a link to go to another interface for advanced query and search.

3.10 Personalization

Personalization is essential to e-commerce applications. For example, e-commerce sites track users' navigation or purchase behavior and show the users products related to users' recent purchases or browsed categories. For example, Figure 5 shows a window similar to the window in Figure 4 except additional personalization on the right side of Figure 5. The personalization shows users with links pointing to the categories of their recent purchase is included. Once a new purchase is made, the personalization page will change accordingly. Implementing personalization using the right system architecture while reflecting the business intelligence of e-commerce has a great impact to the cacheability and performance. We see that there are two major ways to implement this example:

3.10.1 *Frame-based Implementation*

`Frames` are frequently used in most commercial Web sites for the flexibility they provide in page formatting and layout as well as to simplify users' navigation. The HTML code below shows an example of an index page consisting of three fragment pages grouped by frame.

```
<HTML>
<TITLE>
My PetShop Search Screen
</TITLE>
```

```
<FRAMESET ROWS="30%,40%,30%">
   <FRAME NAME="category" SRC="./category.html">
   <FRAME NAME="search" SRC="./product.search">
   <FRAME NAME="mylink" SRC="./mylink.html">
</FRAMESET>
</HTML>
```

After a browser receives the page *index.html*, it parses the page and then requests three additional pages. Note that although this composite page is displayed to the user as a single and personalized Web page, all fragment pages are cacheable. The category page and search page have fairly long TTL and the personalized fragment page is also cacheable for the individual user and it can be invalidated when a new purchase occurs.

For this frame-based implementation, an edge cache server deployment will be most suitable for fast delivery through caching.

3.10.2 Dynamically-assembly Implementation

If the requested page is dynamically assembled at user request time, all three fragment pages are cacheable but an edge application server is needed to assemble the three fragment pages into one page to serve the user. In this system architecture, the invalidation message for the personalization page needs to be sent to the edge application server instead to the cache server. The network latency and processing latency for fragment pages are reduced.

Note that in the case that neither the edge application server nor the CachePortal is deployed, the application server at the origin site needs to dynamically generate the personalization page for each user and then assemble the three fragment pages into one page to serve the user. The combination of network latency and processing latency may result in slow response time.

3.11 Cluster Architecture

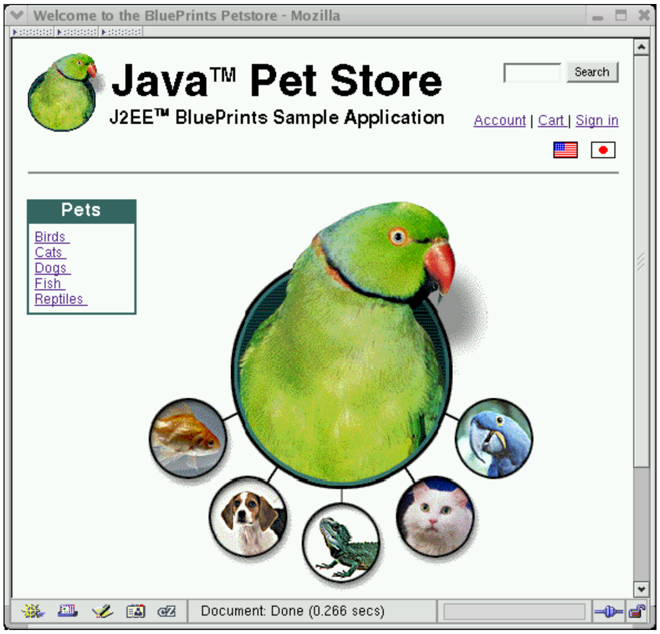
A typical approach to supporting high-availability and scalability is to use a cluster architecture for Web/Application Server and DBMS. If middleware is deployed to provide transparent access to WAS, such as [14], and DBMS, such as [15], the cluster architecture is not really an obstacle to construct the URL and query mapping. If the cluster architecture does not provide transparent access or shared memory, it requires additional effort and it is challenging to monitor all activities across multiple servers.

4. CASE STUDY: JAVA PETSTORE

The Java PetStore is a reference application provided by the Java 2 Platform, Enterprise Edition BluePrints (J2EE BluePrints) program at Java Software, Sun Microsystems. This reference application demonstrates how to use the capabilities of the J2EE platform to develop flexible, scalable, cross-platform e-business applications. It comes with full source code and documentation for users to experiment with J2EE technology and learn how to use it to build enterprise solutions. The J2EE BluePrints program defines the application programming model for the J2EE platform. It provides practice guidelines and architectural recommendations for real-world application scenarios to enable developers to build portable, scalable, and robust applications using J2EE technology. In this paper, we use it to demonstrate how we can enable dynamic content caching and accelerate an e-commerce J2EE applications.

4.1 Overview of the Java PetStore

We installed the Java PetStore and populated the database as instructed in the manual. We then deployed CachePortal on top of the



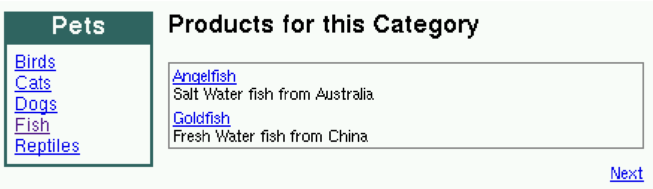
Welcome to the BluePrints Petstore - Mozilla

Java™ Pet Store

J2EE™ BluePrints Sample Application

Document: Done (0.266 secs)

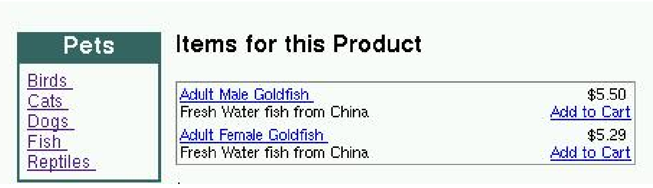
Figure 6: Java PetStore Main Page



Products for this Category

Angelfish
Salt Water fish from Australia
Goldfish
Fresh Water fish from China

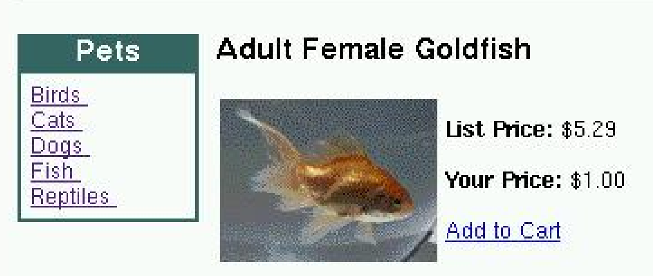
Figure 7: PetStore Category Page



Items for this Product

Adult Male Goldfish	$5.50
Fresh Water fish from China	Add to Cart
Adult Female Goldfish	$5.29
Fresh Water fish from China	Add to Cart

Figure 8: PetStore Item Page

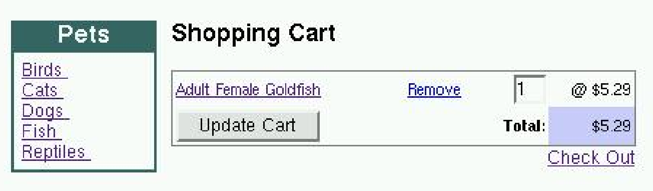


Adult Female Goldfish

List Price: $5.29

Your Price: $1.00

Figure 9: PetStore Item Information Page



Shopping Cart

Adult Female Goldfish	Remove	1	@ $5.29
Update Cart		**Total:**	$5.29

Figure 10: PetStore Shopping Cart Page

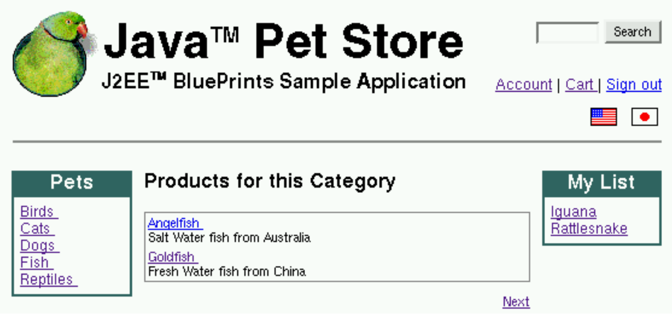

Java™ Pet Store
J2EE™ BluePrints Sample Application
Search
Account | Cart | Sign out
Pets
Birds
Cats
Dogs
Fish
Reptiles
Products for this Category
Angelfish
Salt Water fish from Australia
Goldfish
Fresh Water fish from China
Next
My List
Iguana
Rattlesnake

Figure 11: PetStore My List Page

Java PetStore. The PetStore represents a typical e-commerce Web site, where users are provided with options to navigate through the catalog, items, and place orders.

Figure 6 shows the main page of the Java PetStore, where the users can see all five categories of pets as well as performing search on the whole inventory of pets. When the users click on one of the category page links, a category page similar to Figure 7 will appear. Then, the users can click on one of the two links in the category page to go to sub-categories until an item page is reached. An example of the item page is shown in Figure 8. At the item page, the users can add the pet to the shopping cart or click on the link to go to the information page. At the information page (as shown in Figure 9), the users are presented with detailed information about the pet and have option to click on `add to cart` link. When the `add to cart` link is clicked, the users can modify the quantity or remove the items from the shopping cart completely. The users have an option to check out and log off from the PetStore. At any page, the users can click on the `categories`, `sign in`, `account`, and `change language` links.

Note that users can navigate the categories and add items to shopping carts without signing in. However, after the users login to the system, the system will provide a personalized page, `My List`, on the right side of the window as shown in Figure 10. The links in `My List` is based on what the users purchased recently and the database content is accessed every time when personalization is enabled.

4.2 Enabling Dynamic Content Caching

The purpose of our case study is to see if we are able to enable dynamic content caching without modification to its application program. The tools available to us via CachePortal include:

- CachePortal sniffer and invalidator as described in Section 2;
- `Page Differentiator` as described in Section 3.7;
- Squid cache server with enhancement of POST message handling and configurability through `Page Differentiator`;
- Light weight edge application server that is able to assemble fragment pages if necessary. The edge application server is integrated with the Squid cache server.

We do not modify the Java PetStore to build an API for the URL encoding/decoding purpose and passing cacheability and hashing keys to the edge cache/application server. There are five steps to enable dynamic content caching for PetStore.

There are total 84 possible screens in the Java PetStore. Among them, 79 screens, catalog and item information pages, are cacheable. The remaining 5 non-cacheable screens are `sign in`, `account`, `check cart`, `add to cart`, and `checkout`.

In the Java PetStore, cookies and session objects are created when the users login to the main screen of the system. However, the cookie information is not really used in the PetStore while the session objects are used to store shopping cart information.

We navigated through every page in the Java PetStore and identified that the URL strings can be classified into the following ten patterns:

```
1. petstore/category.screen?category_id=FISH
2. petstore/product.screen.screen?product_id=AV-CB-01
3. petstore/item.screen?item_id=EST-18
4. petstore/changelocale.do?locale=en_US
5. petstore/changelocale.do?locale=jp_JP
-----------------------------------------
6. petstore/signon_welcome.screen
7. petstore/cart.do
8. petstore/customer.do
9. petstore/cart.do?action=purchase&itemld=EST-19
10.petstore/enter_order_information.screen
```

On the list, the first three URLs are for browsing categories, subcategories, items, and item information. The fourth and the fifth URLs are for language switching between English and Japanese. All of these pages are cacheable using `Page Differentiator` to configure the edge cache/application server to ignore the session objects and cookies. The remaining URLs (from the sixth to the tenth) are for the sign in screen, check cart screen, account screen, add to cart screen, and checkout screen. They are not cacheable. Note that in this list, we can only show the URL strings for user requests.

Since fragment pages are used in the Java PetStore, additional included and forwarded requests for the fragment pages are sniffed at the application server filter. The URL and query map stores the mapping between

- URL of origin user requests and queries,
- URL of origin user requests and queries and URLs of included and forwarded requests for the fragment pages, and
- URL of requests for the fragment pages and queries.

For example, for the request to generate the page shown in Figure 7, a URL string, `petstore/category.screen?category_id=FISH`, is captured by the sniffer as A request results in a page with two fragment pages, `category` on the left and `product_id=FISH` on the right in Figure 7. The internal request for the fragment page `category` results in the following query, $Query1$, issued to the database:

```
select a.catid, name, descn
from category a, category_details b
where a.catid=b.catid and locale = "en_US"
order by name
```

The sniffer also captures the query, $Query2$, issued as a result of the fragment page `category_id=FISH`. $Query2$ is as follows:

```
select a.productid, name, descn
from product a, product_details b
where a.productid=b.productid
     and locale = "en_US"
     and a.catid = "FISH"
order by name
```

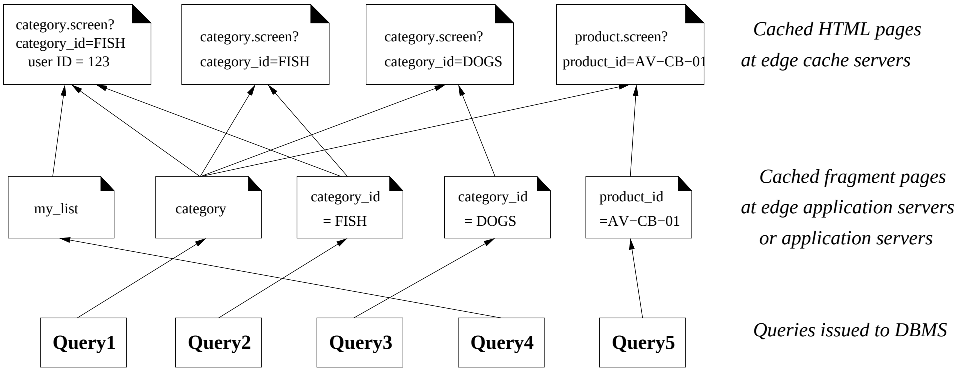


Figure 12: Mapping between HTML Pages, Fragment Pages, and Query Statements

For the user request for a category page for pets related to dogs, `petstore/category.screen?category_id=DOGS`, the sniffer captures $Query1$ and the following query, $Query3$, for the fragment page, `category_id=DOGS`

```
select a.productid, name, descn
from product a, product_details b
where a.productid=b.productid
     and locale = "en_US"
     and a.catid = "DOGS"
order by name
```

For the same URL request with a user login, a customized page shown in Figure 11 is generated. Additional query, $Query4$, is issued to retrieve the purchase history of the user and to generate the fragment page, `my_list`, on the right of Figure 11. When a request for the page shown in Figure 8 is issued, the sniffer captures the URL `/petstore/product.screen?product_id=AV-CB-01` as well as two queries to the database, $Query1$ for the category fragment page and the following query, $Query5$, to generate the fragment page, `product_id=AV-CB-01`, containing detailed information about product items:

```
select catid, name, a.itemid, b.image, b.descn,
       attr1,  attr2, attr3, attr4, attr5,
       listprice, unitcost
from item a, item_details b, product_details c,
     product d
where a.itemid=b.itemid
      and a.productid=c.productid
      and d.productid=c.productid
      and b.locale = c.locale
      and b.locale = "en_US"
      and a.productid = "AV-CB-01"
```

4.3 Monitoring and Invalidation

In Figure 12, we summarize the relationship between HTML pages cached at edge cache servers and fragment pages cached in either at the edge application servers or at the application servers (at the origin site). The relationship is captured by the application sniffer automatically. The figure also illustrates the map between fragment pages and queries, which is captured by the JDBC sniffer. Note that if a HTML page is generated without using any fragment page, a direct map between the HTML page and queries is constructed. In the implementation of PetStore, all HTML pages are constructed based on multiple fragment pages with or without user login. Also note that a request for a page may result in multiple queries.

From these captured query statements, we can identify a list of tables, including `category`, `product`, `product_details`, `item`, and `item_details`, that store information necessary to generate requested pages. Thus, these tables need to be monitored for invalidating cached HTML pages or fragments when database content changes occur. For example, if a database change is detected and the invalidator determines that `Query3` needs to be invalidated, the fragment page `category_id=DOGS` is invalidated from the edge application servers and consequently the HTML page `category.screen?category_id=DOGS` needs to be invalidated. For detailed description of the invalidation scheme, please see [10].

CachePortal is designed to accelerate very large scale data center hosted database-driven Web applications. We have evaluated the system using an e-commerce application and it is capable of tracking 50 million dynamic content pages in 10 cache servers and it assures content freshness of these 50 million pages by invalidating impacted pages within 12 seconds once database content is changed. Our approach also provides better scalability and significantly reduced response times up to 70% in the experiments. Some experimental evaluation results are described in [16, 12].

The Java PetStore is now dynamic content caching enabled and is ready for the evaluation of performance gain through the deployment of CachePortal. Note that since HTML pages in Java PetStore are personalized if users login and they have purchased some items in the past, the cache hits on edge cache servers are only for requests from the users who do not login or have no purchase history. For all other requests other than transactional requests related to orders and shopping carts, edge application servers can dynamically assemble requested pages based on cached fragments at the edge application servers. The invalidations will occur to the following fragments pages:

- `my_list` fragment (on the right Figure 11) when a user makes new purchases;
- `category` fragment (on the left of Figures 6 to 11) when a new category is added to the PetStore;
- `category_id` fragment (on the right of Figure 7) when a new item added to a category;
- `item` fragment (on the right of Figure 8) when a new item is added to a category or the standard list price for an existing item is changed; and
- `item_details` fragment (on the right of Figure 9) when the standard list price or special price for an existing item is changed.

5. EXPERIMENTS

In this section, we present the experimental results of evaluating the Java PetStore's performance gain and scalability improvement with the deployment of NEC's CachePortal Web acceleration solution and the proposed guideline. We first describe the general experiment setup that consists of Web servers, application servers, DBMS, and network infrastructure that are used in the experiments.

5.1 General Experimental Setting

The content delivery configuration is similar to that described in Section 2. We used two heterogeneous networks that are available in the NEC's facility in Cupertino, California: one is used by the C&C Research Laboratories (referred to as CCRL) and the other one is used by *cacheportal.com* (referred to as CP). Users and edge cache servers are located in the CP network while Web server, application server, and DB Caches are located in the CCRL network. The average round trip time on the CCRL-CP connections is through a network delay generator using Dummy Net provided by FreeBSD. The round trip time within the same network is negligible. In summary, connectivity within the same network is substantially better than that across the Internet and there is large network latency.

The system and software configuration for each component are as follows:

- Edge cache server: Squid 2.4.7 with enhancements to handle POST requests on a 1500 MHZ Pentium 4 machine running Linux 8.0.

- Web server and application server: Tomcat 4.1.24 Web Server and JBoss 3.2.1 Application Server are used on a 1500 MHZ Pentium 4 machine with 1 G Byte main memory running Linux 8.0.

- DBMS: Oracle 9i is used as the database system and it runs on a 2300 MHZ Xeon machine with 2 G Byte main memory running Linux 8.0.

5.2 Evaluation of Performance Gain

The first experiment we conducted is to measure the performance gain (in terms of the response time observed by the users) achieved through our proposed approach. In this experiment, the network latency is set to 200ms, 400ms, and 800ms and the number of concurrent users are set to in a range between 20 and 100. We randomly assign the users to navigate the Java PetStore applications. Thus, some users may look at the catalog while some of them may add items to shopping carts and are ready to check out. We ran the experiments based on combinations of all these parameter values for the PetStore without CachePortal and PetStore with the deployment of CachePortal with cache hit rates at 60%, 70%, 80%, and 90% (i.e., hits at the edge cache servers or the edge application servers). We then record the response time of each users. The experiments are repeated five times and the average user response time is used to plot in Figure 13.

As we can see in Figure 13, the system without dynamic content caching and CachePortal yields very large response time and the response time increases almost in proportion to the number of concurrent users and network latency. On the other hand, the systems with dynamic content caching and CachePortal consistently provide fast average response time under 4 seconds.

5.3 Evaluation of User Experience

The next experiment we conducted is to measure user experience in terms of

- percentage of the users receiving error message due to system overload or time out; and

- percentage of the users receiving response under 7 seconds.

The experiment setting is the same as the previous experiment except we vary the number of concurrent users between 20 and 200 in order to create system overload.

Note that these experiments illustrate the user's actual experience. As we can see in Figure 14, at the load of 50 concurrent users, the users of the Java PetStore start to experience unpleasant delay and at the load of more than 100 concurrent users, more than half of the users experience delay more than 7 seconds. In Figure 15, we can see at the load of more than 120 concurrent requests, the users start to receive error messages.

In the figure, we also see that the network latency has an impact to the user response time. Note that the network latency not only increases the transmission time but also cause the requests to hold on to the limited number of connections to the application servers and the DBMS.

5.4 Experiments on System Scalability

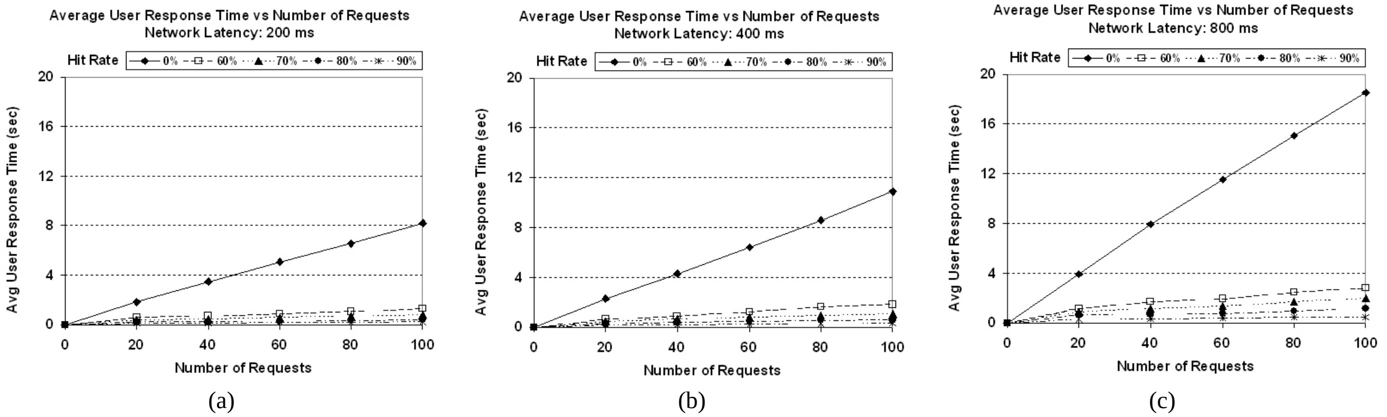
The next experiment we conducted is to measure the scalability of five different systems for Java PetStore: one system without dynamic content caching and four systems with dynamic content catching enabled and deployment of CachePortal. These four systems have hit rates of 60%, 70%, 80%, and 90%, respectively. We want to experimentally derive the limitation of each system configurations to serve user requests under 7 seconds on average.

In Table 1, we show the maximum number of concurrent users that each system can support with average user response time under 7 seconds. In Figure 16, the X-axis indicates the number of concurrent user requests and the Y-axis indicates the network latency between the users and the application server. We tested the limitation of each system by increasing the number of concurrent user requests and network latency until the average user response time is above 7 seconds. We then plot the number of concurrent user requests and network latency as the limitation for the system in terms of scalability.

As shown in Figure 16, when the system has no dynamic content caching (the left most line), its scalability is very limited. On the other hand, when we deploy CachePortal software for dynamic content caching, the system can be scaled up to handle more concurrent users as well as higher network latency. For a common network condition in which the round trip time is around 400 ms, the system with 90% hit rate is scaled up to for more than 20 times as pointed out in Figure 16 and in the second row of Table 1.

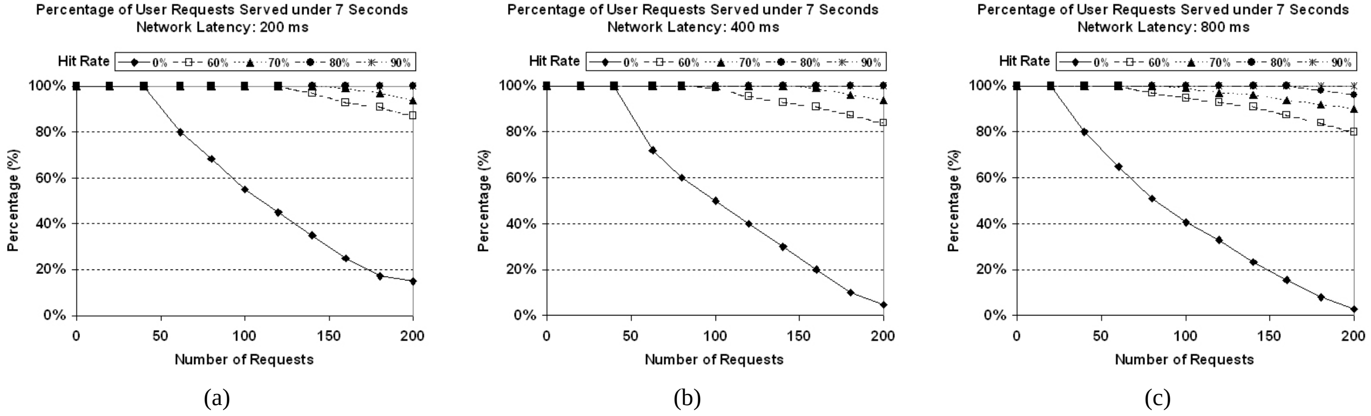
6. RELATED WORK

Applying caching solutions for Web applications and content distribution has received a lot of attention in the Web and database communities [6, 17, 18, 19, 20, 21, 22]. These provide various solutions to accelerate content delivery, such as middleware level cache/pre-fetch solutions, which lie between application servers and underlying DBMS or file systems. They do not provide automated URL/Query mapping construction and invalidation functionalities.



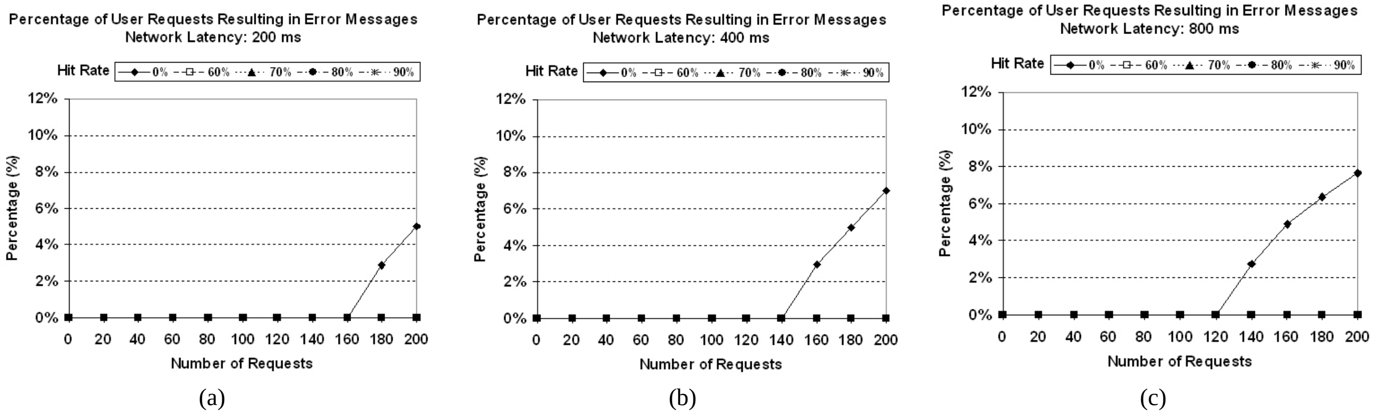
(a) (b) (c)

Figure 13: Effects of Number of Requests on Average User Response Time for Network Latency of (a) 200 ms, (b) 400 ms, and (c) 800 ms



(a) (b) (c)

Figure 14: Effects of Number of Requests on Percentage of User Requests Served under 7 Seconds for Network Latency of (a) 200 ms, (b) 400 ms, and (c) 800 ms



(a) (b) (c)

Figure 15: Effects of Number of Requests on Percentage of User Requests Resulting in Error Messages for Network Latency of (a) 200 ms, (b) 400 ms, and (c) 800 ms

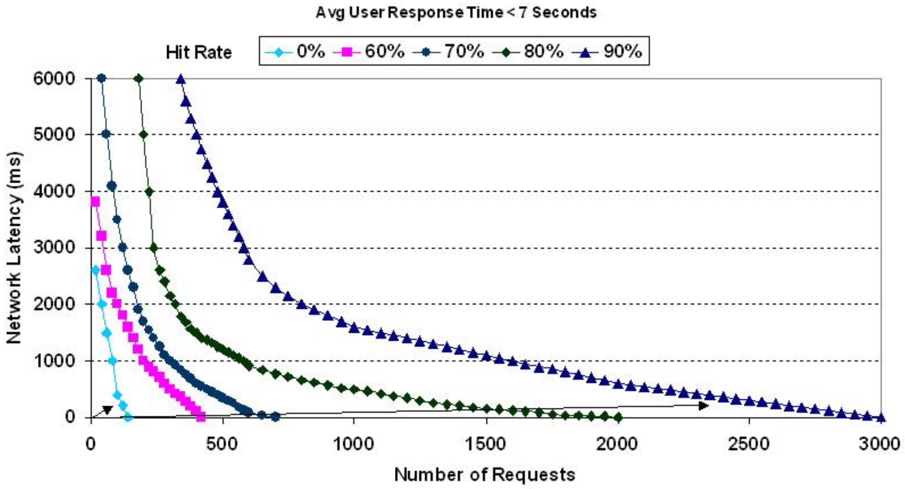


Figure 16: Evaluation of System Scalability

Latency / Cache Hit Rate	0%	60%	70%	80%	90%
200	120	385	550	1400	2600
400	100	330	480	1100	2350
600	95	280	400	860	2000
800	75	240	350	660	1800
1000	65	200	300	580	1600

Table 1: Limitation of Systems

WebCQ [23] is one of the earliest prototype systems for detecting and delivering information changes on the Web. However, the change detection is limited to ordinary Web pages. Yagoub et al. [24] have proposed caching strategies for data intensive Web sites. Their approach uses materialization to eliminate dynamic generation of pages but does not address the issue of view invalidation when the underlying data is updated. Labrindis and Roussopoulos [25] present an innovative approach to enable dynamic content caching by maintaining *static* mappings between database contents and Web pages, and therefore this approach requires a modification to underlying Web applications.

Dynamai [3] 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 into existing e-commerce framework. Jim Challenger et al. [26, 27] at IBM Research have developed a scalable and highly available system for serving dynamic data over the Web. The IBM system was used at Olympics 2000 to post sport event results on the Web in a timely manner. The system provides tools to define fragment pages and their dependency. It utilizes database triggers to generate update events as well as intimately relies on the semantics of the application to map database update events to appropriate Web pages.

SPREAD [28], 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 [28] focuses on static content and describes techniques to synchronize static content, which gets updated periodically, across Web servers.

Yuan et al. [29] evaluated the benefit of edge caching and offloading for dynamic content delivery. In this work, Pet Shop is used for benchmark. Note that Pet Shop comes from Sun's J2EE reference software PetStore but implemented using ASP.NET. The evaluations verify the benefit of dynamic content caching but at the same time they point out the enabling process is overly complex and even counter-productive. In their prototype, application programs need to be modified and user-specified TTL is used to enforce the consistency in contrast to that CachePortal provides more plug-and-play deployment and features invalidation schemes to ensure cache content freshness.

All these related research activities focus on Web acceleration solution or maintenance of strong consistency for the caches rather than the practical and engineering issues of how to apply and enable dynamic content solutions. None of their approaches, however, evaluate the impact of these caching strategies in a real e-commerce environment such as the one described in this paper.

7. CONCLUDING REMARKS

The framework of enabling dynamic content caching to accelerate Web applications and its applicability truly rely on *automated construction of the URL and query mapping*. CachePortal develops *sniffer* and *Invalidator* to automate and scale up this task. In this paper, we identify the challenges in deploying CachePortal to real world e-commerce J2EE-based Web applications. We recommend practices to transform a typical database-driven J2EE applications to a cache friendly Web application where Web acceleration solutions can be applied. Furthermore, such transformation can be done without modification to the underlying application business logic and without sacrificing functions that are essential to e-commerce. We take the J2EE reference software, the Sun's Java PetStore, as a case study. After applying the guideline, we are able to cache more than 90% of the content in the PetStore and scale up the Web site more than 20 times.

Future work includes extending our solutions to outside the J2EE setting and PHP-based applications as well as more in-depth study of deploying acceleration solutions for Web applications based on cluster architectures.

Acknowledgments

The authors would like to thank Satish Murthy for useful discussions on this work and acknowledge his contributions in engineering the CachePortal SDK used for experiments in this paper.

8. REFERENCES

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

[2] B. Krishnamurthy and C. E. Wills. Analyzing factors that influence end-to-end web performance. In *Proceedings of the 9th World-Wide Web Conference*, pages 17–32, Amsterdam, The Netherlands, June 2000.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] Qiong Luo and Jeffrey F. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites. In *Proceedings of the 2001 VLDB Conference*, Roma, Italy, September 2001.

[9] Wen-Syan Li, Wang-Pin Hsiung, Oliver Po, K. Seluk Candan, and Divyakant Agrawal. Evaluations of architectural designs and implementation for database-driven web sites. *Data and Knowledge Engineering*, 43(2):151–177, November 2002.

[10] 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 2002 VLDB Conference*, Hongkong, China, August 2002.

[11] Java Software, Sun Microsystems. *Information available at* `http://blueprints.macromedia.com/petstore/`.

[12] Wen-Syan Li, Oliver Po, Wang-Pin Hsiung, K. Selçuk Candan, Divyakant Agrawal, Yusuf Akca, and Kunihiro Taniguchi. CachePortal II: Acceleration of Very Large Scale Data Center-Hosted Database-driven Web Applications. In *Proceedings of the 2003 VLDB Conference*, Berlin, Germany, September 2003.

[13] K. Selçuk 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.

[14] BEA Systems, Inc. *Information available at* `http://edocs.bea.com/wls/docs70/cluster/`.

[15] Tangosol, Inc. *Information available at* `http://www.tangosol.com/`.

[16] Wen-Syan Li, Oliver Po, Wang-Pin Hsiung, K. Selçuk Candan, and Divyakant Agrawal. Engineering and Hosting Adaptive Freshness-sensitive Web Applications on Data Centers. In *Proceedings of the 12th WWW Conference*, Budapest, Hungary, May 2003.

[17] 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.

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

[19] Anoop Ninan, Purushottam Kulkarni, Prashant Shenoy, Krithi Ramamritham, and Renu Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *Proceedings of the 11th WWW Conference*, Honolulu, Hawaii, USA, May 2002.

[20] 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.

[21] 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.

[22] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *Proceedings of the 2003 VLDB Conference*, Berlin, Germany, September 2003.

[23] Ling Liu, Calton Pu, and Wei Tang. WebCQ: Detecting and Delivering Information Changes on the Web. In *Proceedings of International Conference on Information and Knowledge Management*, Washington, D.C., November 2000.

[24] Khaled Yagoub, Daniela Florescu, Valrie Issarny, and Patrick Valduriez. Caching Strategies for Data-Intensive Web Sites. In *Proceedings of the 26th VLDB Conference*, Cairo, Egypt, 2000.

[25] A. Labrindis and N. Roussopoulos. Self-Maintaining Web Pages - An Overview. In *Proceedings of the 12th Australasian Database Conference (ADC)*, Queensland, Australia, January/February 2001.

[26] 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.

[27] 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.

[28] 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.

[29] Chun Yuan, Yu Chen, and Zheng Zhang. Evaluation of Edge Caching/Offloading for Dynamic Content Delivery. In *Proceedings of the 12th WWW Conference*, Budapest, Hungary, May 2003.