Caching personalised and database-related dynamic web pages

Yeim-Kuan Chang*, I-Wei Ting and Yu-Ren Lin

Department of Computer Science and Information Engineering,
National Cheng Kung University, No. 1,
University Road, Tainan City 701, Taiwan
E-mail: ykchang@mail.ncku.edu.tw
E-mail: p7893113@mail.ncku.edu.tw
E-mail: p7895107@mail.ncku.edu.tw
*Corresponding author

Abstract: In recent years, web development is the most important application in internet. Caching related technique improves the web server performance significantly. However, existing caching schemes cannot deal with the dynamic web pages efficiently. Thus, in this paper, we propose a caching scheme and then use web session objects and database-related dynamic web cache to implement the dynamic web cache system in Tomcat web server. We show how to build the dependency between dynamic web pages and the underlying database fields and session objects. Our experimental results show that Tomcat with proposed dynamic web cache can increase the stability of web server and improve web server throughput by up to 290%.

Keywords: dynamic web pages; cache; consistency; Tomcat; session objects.

Reference to this paper should be made as follows: Chang, Y-K., Ting, I-W. and Lin, Y-R. (2010) 'Caching personalised and database-related dynamic web pages', *Int. J. High Performance Computing and Networking*, Vol. 6, Nos. 3/4, pp.240–247.

Biographical notes: Y-K. Chang received his MS in Computer Science from University of Houston at Clear Lake in 1990 and his PhD in Computer Science from Texas A&M University, College Station, Texas, in 1995. He is currently a Professor in the Department of Computer Science and Information Engineering at National Cheng Kung University, Tainan, Taiwan, Republic of China. His research interests include computer architecture, multiprocessor systems, internet router design, computer networking.

I-W. Ting received his MS in Computer Science and Information Engineering from Chaoyang University of Technology, Taiwan, R.O.C., in 2004. He is currently working towards his PhD in Computer Science and Information Engineering at National Cheng Kung University, Taiwan, R.O.C. His current research interests include cache design system, cooperative caching and broadcasting strategy.

Y-R. Lin received his MS in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, R.O.C., in 2005. His research interests include cache design system in web server.

1 Introduction

As the internet traffic grows continuously, more and more users browse their desired personalised or dynamic web pages which are created dynamically based on the data stored in the database system. This characteristic requires web servers to generate and serve users the requested page content dynamically. When the web server receives a request for the dynamic content, it queries the database to extract the relevant information needed to generate the requested content dynamically. Even when the underlying data retains the same value in the database, web servers query the database and regenerate the same content of dynamic web pages every time a request is received. As a result, to improve the clients' response time, one option is to build a high performance website for improving network and server capacity by deploying a state of the art IT infrastructure. Another option is to use 'load balancing' structure among multiple web servers. It can be achieved by migrating a document from a server to another and/or replicating a document across more than a web server. DC-Apache (Li and Moon, 2001) can dynamically manipulate the hyperlinks embedded in web documents in order to distribute access requests among multiple cooperating web servers. To improve the web server performance, one solution is to cache the result of requested content. The cached copy may exist in client/browser, proxy, content delivery networks (CDN), and the web server itself described as follows. The Browser Cache caches the static

request content in the hard-disk of clients. Like Internet Explorer (IE) caches all requested content except it recognises the request content is a dynamic web page. IE does this simply by examining the extension of the request content. If the extension is .jsp, .php, .asp, IE would not cache it. If the request URL contains the cgi string, it is also a dynamic web page request and IE would also not cache it. Browser Cache improves the performance a lot, by dispalying image files and, static content without reloading them. However, the main problem of Browser Cache is that it cannot make promise the current static content is up-to-date and it cannot cache dynamic content.

Proxy Cache is used by ISPs, large corporations, schools and workgroups as well as multi-pc home networks. It is also designed to cache static content. When one user accesses some static web content, proxy servers cache them and update the lifetime of cached objects. Later, when another user access to the same static web content, proxy servers return the request content without re-requesting the web servers. The main problem of Proxy Cache is similar to Browser Cache but one advantage is proxy server has more disk space to cache static content and the cached copy can be shared by many clients.

Network-Wide Caches, implemented by CDN is to deploy many CDN servers in the world-wide so that a large fraction of requests can be served remotely rather than all of them being served from the original web server. This solution has the advantage of serving users via CDN servers closer to them and reducing the traffic to the websites, reducing network latency and providing faster response time. Many companies provide CDN services as web acceleration services. However, for many e-commerce and personalised content, web pages are generated dynamically based on the underlying database, rather than static information. Therefore, content delivery by most CDNs is limited to handle static portions of web pages rather than dynamic content.

Server Cache does cache mechanism in the server side. When web servers generate a dynamic content, they cache this content at the same time. Later, when a request to the dynamic content is received, web servers just return the cached copy to the clients. This does not reduce the network latency, but it improves the web server performance a lot. The web servers have no need to access all underlying data like database queries when serving a dynamic content request.

In addition to determining where to cache, push-pull (Bhide et al., 2002) discussed how to maintain data consistency between servers and proxies. In the case of servers adhering to the HTTP protocol, clients need to frequently pull the data based on the dynamics of the data and a user's coherency requirements. In contrast, servers that possess push capability maintain state information pertaining to clients and push only those changes that are of interest to a user. Their studies show that a push-based approach is suitable when a client has stringent coherency requirements or when communication overheads are the bottleneck. A pull-based approach is better suited under less stringent coherency requirements, when server computational loads are the bottleneck, or when resilience to failures is important. It focuses on the relationship between servers and proxies, but is also useful for other cache system.

Although dynamic web pages can be cached, sometimes it gets better performance to generate the dynamic web pages than maintain a cached copy. In particular, dynamic web pages that change very fast like stock are not suitable to cache.

In this paper, we implement a caching system in Tomcat web server and deal with the cache problem of dynamic and personalised web pages. The goal is to find all relevant underlying data to dynamic web pages precisely. Thus, we explore the database, find all table names and field names. Then check if the MD5 of the requested content changes when the column of one field is changed. If the MD5 changes, it means the database field is relevant to the dynamic web pages. This is a brute-force method, but with some tricks we can make the process faster. In particular, most personalised website stores data in session objects. We use the data in the session objects to generate dynamic personalised web pages. These kinds of web pages are session related. We also solve the problem of session related dynamic web pages by signing in before accessing the MD5 of the requested content. Mapping session related dynamic web pages to database fields can be done by this method. When we emulate signing action, the data has already been loaded from the database to the session objects. Later, these session objects can be shared by the subsequent requests. Our proposed cache system is implemented in Tomcat web server and the dynamic web pages are JSP or servlet pages. Therefore, we name it Tomcat Dynamic Web Pages Caching System, or Tomcat Cache in short.

The rest of this paper is organised as follows. In Section 2, we explain the system design and implementation of our Tomcat caching system. In Section 3, we present the performance results. The related work is reviewed in Section 4. Last, we give our conclusions in Section 5.

2 System design and implementation

We first explain the process in Tomcat request-response loop with proposed Tomcat Cache System. The flowchart is shown in Figure 1. All steps are executed sequentially when each request is received by Tomcat web server. We use the *Borland Optimizeit Profiler* to trace all method calls happened inside Tomcat. It also allows you to integrate the *profiler* to most popular web servers easily, like *IBM webSphere 5*, *Oracle 9i*, *webLogic 8.1*, *BES 5.2.1*, *Jakarta Tomcat 5*, and *Jboss 3.2*. We configure it to integrate to *Tomcat 5* and run the *CPU profiler* to trace all method calls. At the same time we launch the standard WebBench benchmark suite, i.e., static benchmark suite (it accesses thousands of static files in the wetree directory). After WebBench finishes the static benchmark suite, we stop the *CPU profiler* and see the result. The result is something like Figure 2 which displays method call tracing of thread

http-80-Processor21. With *Borland Optimizeit Profiler* result, we can trace what happens inside Tomcat. It displays how much time a method is used in and under method and how much time a method is used in invoked methods and which methods are called by the current method.

Figure 1 Tomcat request-response loop with Tomcat cache system

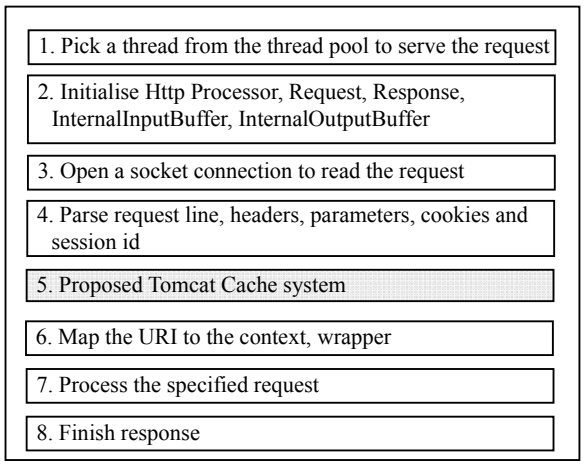


Figure 2 CPU profiler of http-80-Processor21 (see online version for colours)

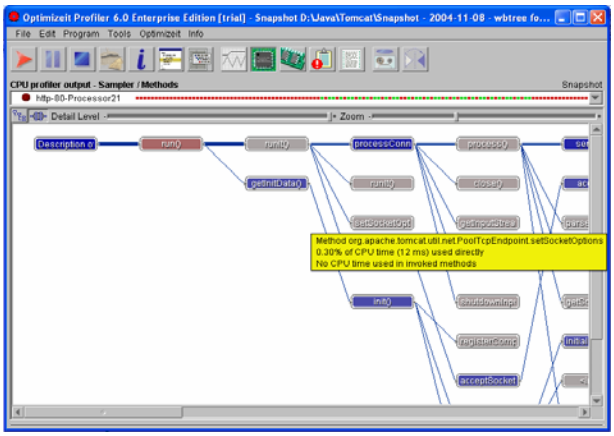
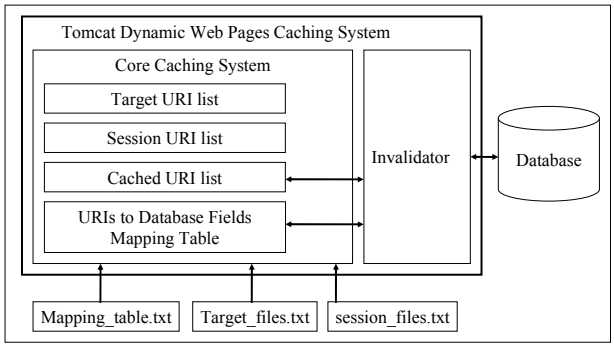


Figure 3 Architecture of Tomcat dynamic web pages caching system



The design idea of Tomcat Cache is to develop a general caching system inside Tomcat, not for any particular web application. The caching system is independent from web applications, such that we do not have any application-related code in Tomcat. The architecture of Tomcat Cache is shown in Figure 3. The Core Caching System and the Invalidator are main components.

At the initial state, Core Caching System loads mapping_table.txt, target_files.txt and session_files.txt into Target URI list, Session URI list and URIs to Database Fields Mapping Table separately. When Tomcat serves a request, Core Caching System would see if the request URI is in the cached URI list. If yes, it returns the cached web page. Otherwise, it generates the dynamic web page and stores it into the cache.

The Invalidator runs periodically to check the update log file of the database called Binary-Update-Log. If it finds some fields are changed, it tells Core Caching System to remove related URIs from cached URI list.

The main challenge of Tomcat Cache is to build a mapping between the cached web pages and database fields. DUP (Iyengar et al., 1999) builds the mapping manually. Our goal is to construct the URIs to Database Fields Mapping Table automatically by programs.

First of all, we extract the relevant information from the web access log to know which URIs are accessed in this web server. We exclude all static content (e.g., image files, html files) and extremely dynamic content (e.g., verify.jsp which is used to verify if user name and password are valid). We find the URIs of the dynamic web pages that is suitable to be cached in target_files.txt. We map them by the following steps:

Step 1 Start the Tomcat web server and MySQL database server. Backup all database tables.

Step 2 Get one URI from the target_files.txt.

Step 3 Get one field from the database fields, e.g., camera.price that is a combination of table name and a field name.

Step 4 Access the URI and record the MD5 of the requested content.

Step 5 Change all values of this field. If the type of the field is String we set it to null then recover it back to the original value next time. If the type of the field is int or float, we add 10 to it.

Step 6 Access the URI again and get the MD5 of the requested content. Then compare the new MD5 with the former MD5 and see if they are different. If yes, it means the field we changed is related to the URI.

Step 7 Go back to Step 3, until all database fields are examined.

Step 8 Go back to Step 2, until all URIs are examined.

Step 9 Restore the database and save the result to mapping_table.txt.

Using the steps described above, we can find the relationship between URIs and database fields. However,

some web pages, especially personalised web pages use session objects to save data grabbed from the database and create dynamic web pages by the information in the session objects. These web pages are designed to load personalised data from the database after user's login, then save these data to session objects for future use. This scheme is widely used and largely improves web server performance, because the web server does not need to access the database each time it creates a personalised web page but uses data stored in session objects to create personalised web pages. Therefore, web pages depend on the session objects and session objects depend on some database fields. We call the kind of relationship indirect mapping.

For example, when a user signs in, verify.jsp checks if the user is valid. If the user is valid, verify.jsp loads user related data from the database into the session objects. Then in the same session all requests can use the data in the session objects.

When we want to find the indirect relationship between URIs and database fields, we should emulate our web client logins to the website then access the URI content. But how does Tomcat maintain session connection? We use *Web Performance Trainer* 2.7 to record the process of the web pages access actions. We use *Web Performance Trainer* only for recording HTTP request header and response header. When *Web Performance Trainer* 2.7 starts the benchmarking process, it launches IE. Then records each step what IE does, including the session connection, the cookies, and the form inputs. After recording the web pages you want to benchmark, *web Performance Trainer* 2.7 displays each state of request and response. We use this tool to find out Tomcat session scheme.

After we know how Tomcat maintains a session, we can emulate our clients to use sessions and find out the indirect mapping. We map them by the following steps:

Step 1 Start the Tomcat web server and MySQL database server. Backup all database tables.

Step 2 Get one URI from the target_files.txt.

Step 3 Get one field from the database fields, e.g., camera.price that is a combination of the table name and a field name.

Step 4 Emulate login process by one of the user/password then get the value of JSESSIONID.

Step 5 Access the URI with the JSESSIONID cookie and record the MD5 of the requested content.

Step 6 Change all values of this field. If the type of the field is String we set it to null then recover it back to the original value next time. If the type of the field is int or float, we add 10 to it.

Step 7 Access the URI with the JSESSIONID cookie again and get the MD5 of the requested content. Then compare the new MD5 with the former MD5 and see if they are different. If they do, it means the field we changed is related to the URI.

Step 8 Go back to Step 3, until all database fields are examined.

Step 9 Go back to Step 2, until all URIs are examined.

Step 10 Restore the database and save the result to mapping_table.txt.

With session detection, we can build the relationship of indirect mapping.

We implement the Core Caching System inside Tomcat web server. When Core Caching System starts up, it loads target_files.txt to Target URI list, mapping_table.txt to URIs to Database Fields Mapping Table, and session_files.txt to Session URI list. Target URI list is used to check if the requested URI is our target URI. If it isn't, the request breaks the caching system checking condition, decreases the overhead of caching system. Session URI list is used to check if the request URI is session-related. If it is a session-related request, Core Caching System adds the session information to the id of cached object. Cached URI list is used to check if the request URI is cached. If the request URI is in the cached URI list, Core Caching System redirects the request URI to the cached object URI. When the invalidator detects some database fields changed, it notifies Core Caching System to remove the associating URI from the cached URI list. URIs to Database Fields Mapping Table is static and immutable, and provides the mapping information between URIs and database table fields. When a request is received, the Core Caching System checks if the content of the requested page is cached. If it finds a valid cached copy, it redirects the request URI to the URI of the cached content. Therefore, we implement the Core caching system after Tomcat parsed the request URI and before Tomcat maps the request to associating context, wrapper. This location is in the org.apache.coyote.tomcat5.CoyoteAdapter.service() method. In particular, we need to find out where we implement the Core Caching System. When a request is received, the Core Caching System checks if the response content of the request is cached. If it finds a valid cached copy, it redirects the request URI to the URI of the cached content. Therefore, we also implement the Core Caching System after Tomcat parsed the request URI and before Tomcat maps the request to associating context, wrapper. This location is in the org.apache.coyote.tomcat5.CoyoteAdapter.service() method.

Now we explain how to use Binary Update Log to implement invalidator. The Binary Update Log contains all SQL statements that update data. MySQL logs only statements that actually change the data. For example, a delete statement that fails to affect any rows is not logged. Update statements that set column values to their current values are also not logged. MySQL logs updates in execution order. Such that when the Invalidator parses the log, it skips all processed data and only parses new ones.

Launching MySQL with the --log-bin argument starts the Binary Update Log. If you do not specify the filename, MySQL uses the hostname-bin as the default filename, in

our example it is ren-bin. MySQL appends a numeric index to the end of the filename so that the file looks like ren-bin.001. When MySQL server restarts or refreshes or the log reaches the maximum log size, MySQL creates another log file like ren-bin.002 to log. MySQL also creates an index file that contains a list of all used binary log files. By default, the file is named something like ren-bin.index. You may change the file name or path of the index file with the --log-bin-index=file option.

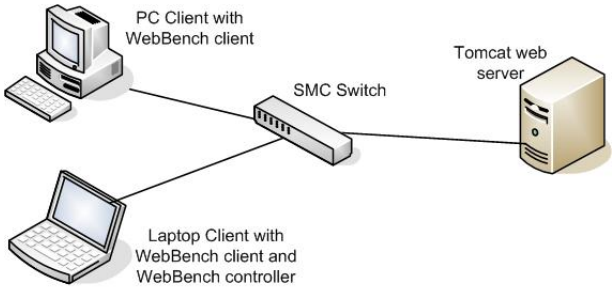
We implement the Invalidator Thread as a thread inside Tomcat so that it is easy to communicate with the Core Cache System. The thread starts to run when the RenTomcat.getRenTomcatInstance() is called in the first time. The Invalidator Thread runs periodically to check if some data is changed by the Binary Update Log. Binary Update Log logs all update SQL statements in execution order. Therefore, the Invalidator Thread does not need to parse through the whole log file, instead it starts to parse from the last time it checked. The incremental parsing improves performance a lot. The Invalidator Thread runs periodically, using the period saved in valiable TomcatConstant.POLLING_TIME. When it wakes up, it reads the Binary Update Log last parsed into a string. Then it finds the table name and field name from the update statements by regular expression. Actually, we should have more regular expression patterns to extract table name and field name from all possible update SQL statements. However, it depends on how the programmers write SQL statement codes.

3 Performance evaluation

In this section, we present our performance gain and the overhead of our web cache system. We use one PC and one laptop as the web clients and another PC as the web server. All PCs are connected by the SMC switch. The network configuration is as Figure 4. The system and software configuration for each component are as follows:

- DBMS: MySQL 4.0.15 is used as the database system and it runs on the same machine as Tomcat web server runs, AMD 2500+ with 1G byte main memory running Windows XP SP2.
- Switch: SMC SMC2804WBR.
- JVM: Sun Java(TM) 2 SDK, Standard Edition 1.4.2_03.
- Server: Tomcat 5.0.24 is the web server. We run it with the parameter JAVA_OPTS = -server –Xms128m –Xmx384m which makes it run faster. –server means it runs in server mode, JVM ignores keyboard, mouse events. –Xms128m means JVM runs with initial 128M byte main memory and –Xmx384m is the maximum available memory JVM that can be used.
- Laptop web client: We run WebBench controller and client on this laptop machine, Intel Pentium 4 1,600 MHz with 768M byte main memory.
- PC web client: We run WebBench client on this PC machine. It communicates with laptop WebBench controller and runs web load at the same time. Intel Pentium 4 3,000 MHz with 1G byte main memory.

Figure 4 Network configuration (see online version for colours)



WebBench (http://www.veritest.com/benchmarks) is used to emulate many clients that send requests to web servers. It is a closed-loop benchmark tool such that clients do not send requests faster than the server can respond. One controller is responsible for cooperating these clients to run one mix. After one mix finished, all statistic data items are collected at the controller and presented as a Microsoft Excel document.

We set the *Ramp Up*, *Ramp Down*, and *Length* to 5 sec, 5 sec and 30 sec, respectively. It means in each mix we record only 20 seconds. Because our website is session-related, we check the *Enable Cookie Support* option, such that WebBench treats each mix of a client as single session and they can share the same session objects. In the mix configuration, we configure WebBench to run each mix three times, each mix use two clients, i.e., PC client and laptop client. Engine-per-client is added by one incrementally until total engine-per-client reaches ten. Therefore, we have 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 clients tested.

Figure 5 shows the result of performance comparison according to requests per second. Original means the original Tomcat web server without web cache system, while web cache means Tomcat web server with proposed web cache system. We can see that Tomcat with proposed web cache system can serve more requests, up to 800 requests per second, while the original Tomcat can only serve 270 requests per second. Thus a 290% throughput improvement over the original web server is obtained. With web cache enabled, Tomcat does not need to request database each time a database-related request comes in and does not need to regenerate the dynamic web pages which contain the same data. Another observation is that web cache performs much smoother than the original. Because web cache serves dynamic web pages request by cached copy, it uses less system resource, like socket connections, process context switch, etc. While the original Tomcat generates database-related dynamic web pages, it queries the database then recomputes the dynamic web pages, spends much more CPU time.

Figure 6 shows the performance comparison in average response time. Note that web cache system is still more stable than the original Tomcat. When 20 clients request the original web server, the average response time is worse than 90 milliseconds. Figure 7 shows that the original curve of response time standard deviation grows much faster when clients increase. We can conclude web cache system is more stable than original Tomcat web server.

Figure 5 Performance comparison on requests per second (see online version for colours)

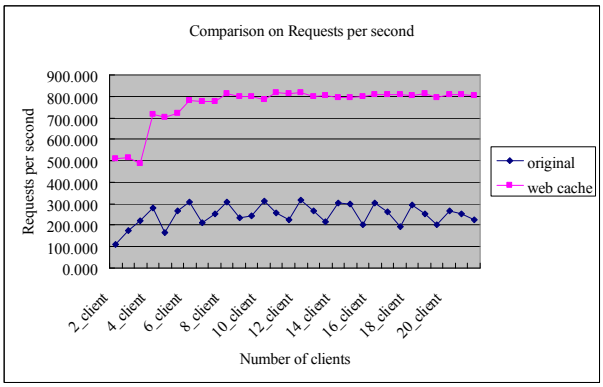


Figure 6 Response time comparison (see online version for colours)

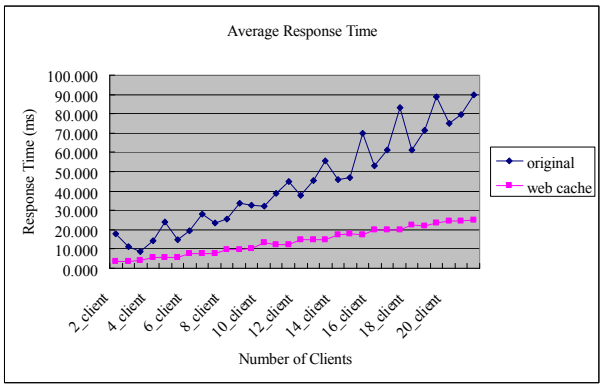
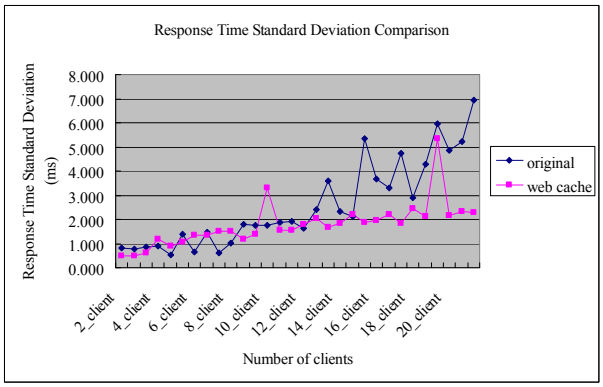


Figure 7 Response time standard deviation comparison (see online version for colours)



4 Related work

There are various ways to accelerate access of dynamic web pages. Oracle Database Cache (http://www.orafaq.com/faqicach.htm) is a database-replicated cache directly associated with Oracle9i Database, not with any particular application. A copy of the data is loaded from the database into the database cache, which is on the middle tier and, therefore, closer to the application for faster access. Queries are made locally to the database cache instead of across the network to the original database, such that saving costly round trips and eliminating network latency. However, it cannot reduce the redundancy arising from the web server to the database server computations. Furthermore, due to the consistency between the data cache and the original database, it requires heavy database-cache synchronisation overhead.

Another scheme is Oracle Web Cache (http://www.oracle.com/technology/products/ias/web_cache/index.html). It is a HTTP-level cache, maintained outside the application, providing very fast cache operations. It is a pure, content-based cache, capable of caching static or dynamic data with time-based, application-based, or trigger-based invalidation of the cached pages. However, it does not provide a mechanism through which updates in the underlying data can be used to identify which web pages in the cache to be invalidated. The use of triggers for this purpose is very inefficient and may introduce a large overhead on the underlying database system, defeating the original purpose.

DUP (Iyengar et al., 1999) maintains data dependence information manually between cached objects and the underlying data which affect their values in a graph. In the proposed scheme (Challenger et al., 2004), they improved the system to general multitier architectures and add a set of algorithms that have been deployed for high-performance serving of dynamic content to many clients at highly accessed websites. The disadvantage of DUP is that it maintains the dependence graph manually, such that each time a new data source or some new web pages are added to the website, the administrator should rebuild the graph. It takes a lot of time, and not efficient.

Class-based Cache (Zhu and Yang, 2001), offers a complementary solution for applications that require coarse-grain cache management. Their approach allows users to specify coarse-grain dependence among underlying datasets and groups of dynamic pages called URL classes which share common URL patterns or client information. As a consequence, application programmers do not need to enumerate data dependences for individual pages. This scheme also supports group-based invalidation of pages which share common patterns and these patterns can be unknown to the cache in advance. This solution in general fits into applications with relatively slower changing data.

There are two schemes proposed by Li (2004) and Candan et al. (2001). They further develop another method to build the dependence graph automatically. They build a query logger into the JDBC driver as a JDBC wrapper, then each time the Java web servers do database queries, the

sniffer can capture the queries that are sent to the database by the web server. They also develop another request logger to capture current requests. Then combine these two loggers into a sniffer module. When the web server receives a dynamic web page request which queries database, the request logger captures the current request URL and the query logger extract the SQL statements, then sniffer module maps the two information to build a Request-to-Query Mapping table which they called QI/URL map. The disadvantages of this method include:

1 They implement the cache system outside the application server. This makes something difficult like getting the information of request URI, cookies, and session objects.

2 Not everything they captured by the query logger is relevant to the requested web page. For example, real world applications do deploying 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 user. All logging database operations that do not affect the freshness of a cached page are captured also by the query logger. Such that they develop a GUI tool to distinguish some related data source from non-related one. However, it seems their automatical method goes back to manual method.

WebGraph (Mohaptra and Chen, 2002) develops a framework called WebGraph that helps in improving the response time for accessing dynamic objects. The WebGraph framework manages a graph for each of the web pages. The nodes of the graph represent weblets, which are components of the web pages that either stay static or change simultaneously. The edges of the graph define the inclusiveness of the weblets. Both the nodes and the edges have attributes that used in managing the web pages. Instead of recomputing and recreating the entire page, the node and edge attributes are used to update a subset of the weblets are then integrated to form the entire page. In addition, WebGragh can achieve the lower response time.

The performance comparison of proposed dynamic web technologies can be found in Titchkosky et al. (2003).

5 Conclusions and future work

We present a method to solve the problem of mapping request URIs to the underlying database fields. Furthermore, with session objects detection mapping, dynamic web pages that use numerous session objects to save personalised data can also be detected and cached. This important feature helps a lot real world web applications that offer personalised service. In addition to the framework design, we implement the system in Tomcat. Our experimental results show that our proposed caching system improves the Tomcat throughput on dynamic web pages by up to 290%.

Future work includes extending our solutions to improve the performance of session-based dynamic web pages. In current solution, if one of the personalised underlying data changes, all the other cache web pages of session-based dynamic web pages would be invalidated. This decreases performance of cache system greatly. In order to solve this problem, we should build application-based cache system to improve the performance. So, the cache system would be able to distinguish the relation between underlying data and cached web pages. Our performance evaluation setting is based on the LAN network. In the future, we will benchmark it in a WAN network environment which can be archived by WAN emulation (Williamson et al., 2002).

Acknowledgements

We are grateful for the supporting resources provided by NSC in Taiwan. This work was supported by the National Science Council, Republic of China, under Grant NSC-96-2221-E-006-190-MY3.

References

Bhide, M., Deolasee, P., Katkar, A., Panchbudhe, A., Ramamritham, K. and Shenoy, P.J. (2002) 'Adaptive push-pull: disseminating dynamic web data', *IEEE Trans. Computers,* Vol. 51, No. 6, pp.652–668.

Candan, K.S., Li, W-S., Luo, Q., Hsiung, W-P. and Agrawal, D. (2001) 'Enabling dynamic content caching for database-driven web sites', *SIGMOD Conference*.

Challenger, J., Dantzig, P., Iyengar, A., Squillante, M.S. and Zhang, L. (2004) 'Efficiently serving dynamic data at highly accessed web sites', *IEEE/ACM Trans. Netw,* Vol. 12, No. 2, pp.233–246.

Iyengar, A., Challenger, J. and Dantzig, P. (1999) 'A scalable system for consistently caching dynamic web data', *INFOCOM,* pp.294–303.

Li, Q. and Moon, B. (2001) 'Distributed cooperative apache web server', in *Proceedings International WWW Conference*, p.10, Hong-Kong.

Li, W-S., Hsiung, W-P., Po, O., Hino, K., Candan, K.S. and Agrawal, D. (2004) 'Challenges and practices in deploying web acceleration solutions for distributed enterprise systems', *WWW Conference,* pp.297–308.

Mohaptra, P. and Chen, H. (2002) 'WebGraph: a framework for managing and improving performance of dynamic web content', *IEEE Journal on Selected Areas in Communications (JSAC)*, September, Vol. 20, No. 7, p.1414.

Titchkosky, L., Arlitt, M.F. and Williamson, C.L. (2003) 'A performance comparison of dynamic web technologies', *SIGMETRICS Performance Evaluation Review,* Vol. 31, No. 3, pp.2–11.

Williamson, C.L., Simmonds, R. and Arlitt, M.F. (2002) 'A case study of web server benchmarking using parallel WAN emulation', *Performance Evaluation*, September, Vol. 49, Nos. 1–4, pp.111–127.

Zhu, H. and Yang, T. (2001) 'Class-based cache management for dynamic web content', *INFOCOM,* pp.1215–1224.

Websites

Oracle Database (iCache) Cache FAQ,
http://www.orafaq.com/faqicach.htm.

Oracle Web Cache,
http://www.oracle.com/technology/products/ias/web_cache/.

WebBench 5.0,
http://www.veritest.com/benchmarks.