

TECHNIQUES FOR EFFICIENTLY SERVING DATA AND DYNAMIC DATA AT WEBSERVERS USING INTERNET AND INTRANET TECHNOLOGY

Prof. S. N. Gujar¹, Prof. G.R.Bamnote², Prof. R.S.Apare², Prof. M.A.Pund², Mr. S.R.Gupta²

¹Smt. Kashibai Navale College of Engineering, Pune, India

Email: satish_gujar@yahoo.com

²Smt. Kashibai Navale College of Engineering, Pune, India

Email: ravi_apare@yahoo.co.in

²Prof. Ram Meghe Institute of Technology and Research, Badnera India

Email: { pundmukesh@rediffmail.com , sunilguptacse@gmail.com , grbamnote@rediffmail.com }

ABSTRACT

This paper presents a new approach for consistently caching dynamic Web data in order to improve performance. The algorithm, which we call Data Update Propagation (DUP), maintains data dependence information between cached objects and the underlying data which affect their values in a graph. When the system becomes aware of a change to underlying data, graph traversal algorithms are applied to determine which cached objects are affected by the change. Cached objects which are found to be highly obsolete are then either invalidated or updated.

1. INTRODUCTION

Many Web sites today have an increasing need to serve dynamic content. Dynamic content is important for Web sites that provide rapidly changing information, e.g., sports Web sites must provide the latest information about sporting events, and financial Web sites must provide current information about stock prices. If the pages for such Web sites are generated dynamically by a server program that is executed every time the pages are requested, the program can return the most recent version of the dynamic content, whereas if files are created to serve the pages statically, it may not be feasible to keep the files up to date. This is particularly true if there are a large number of files that need to be frequently updated. Dynamic content is also important for creating Web pages on the fly from databases. Search engines satisfy queries dynamically from databases. Web pages corresponding to product catalogs are often created dynamically from databases. Information personalized to individual users is also frequently created dynamically.

2. REDUCING DYNAMIC DATA OVERHEAD

2.1. Caching: Caching has been successfully deployed to improve Web performance for static content, but dynamic objects are harder to cache because they change frequently. The DUP algorithms are developed to solve problems related to determining what dynamic pages should be cached and when a cached page has become obsolete. The analysis shows that, despite the higher update rates for dynamic content, the number of

requests for popular dynamic pages far exceed the number of updates to those pages. Hence, judicious use of caching can significantly reduce the number of times such pages have to be regenerated by a server.

2.1.1 Data Update Propagation Algorithm: Data update propagation (DUP) determines how cached Web pages are affected by changes to underlying data which determine the current values of the pages. For example, a set of several cached Web pages may be constructed from tables belonging to database. In this situation, a method is needed to determine which Web pages are affected by updates to the database. That way, caches can be synchronized with databases so that they do not contain stale data. Furthermore, the method should associate cached pages with parts of the database in as precise a fashion as possible. Otherwise, objects whose values have not changed may be mistakenly invalidated or updated from a cache after a database change. Such unnecessary updates to caches can increase miss rates and hurt performance. DUP maintains correspondences between objects which are defined as items which may be cached and underlying data which periodically change and affect the values of objects. Although an entity may be both an object as well as underlying data, objects and underlying data could also be different, which would mean that underlying data are not cacheable. In this system, caches may contain both entire HTML pages and fragments of HTML pages. It is possible for a cached HTML fragment f to affect the value of cached HTML page. In this situation, f would constitute both an object and underlying data. In a simpler system, caches may consist entirely of HTML pages and underlying data may consist entirely of parts of databases. In this case, the underlying data and objects would be disjoint. The system maintains data dependence information between objects and underlying data. When the system becomes aware of a change to underlying data, it queries the dependence information which it has stored in order to determine which cached objects are affected. Caches use dependency information to determine which objects need to be invalidated or updated as a result of changes to underlying data. The cache architecture centers around a cache manager which is a long-running daemon process managing storage for one or more

caches. Application programs communicate with cache managers in order to add or delete items from caches. Application programs are also responsible for communicating data dependencies between underlying data and objects to cache managers. Such dependencies can be represented by a directed graph known as an object dependence graph (ODG), wherein a vertex usually represents an object or underlying data. An edge from a vertex u to another vertex v denoted (u, v) indicates that a change to u also affects v . Node u is known as the source of the edge, while v is known as the target of the edge. For example, if node $go2$ in Figure 1 changes, then nodes $go5$ and $go6$ also change. By transitivity, $go7$ also changes. Edges may optionally have weights associated with them which indicate the importance of data dependencies. In Figure 1, the data dependence from $go1$ to $go5$ is more important than the data dependence from $go2$ to $go5$ because the former edge has a weight which is 5 times the weight of the latter edge. Weights are correlated with the importance of data dependencies.

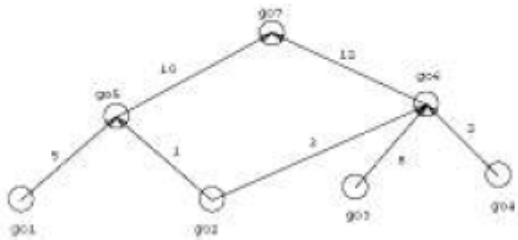


Figure 1: An object dependence graph (ODG).

In many cases, the object dependence graph is a simple object dependence graph having the following characteristics:

- Each vertex representing underlying data does not have an incoming edge.
- Each vertex representing an object does not have an outgoing edge.
- All vertices in the graph correspond to underlying data (nodes with no incoming edges) or objects (nodes with no outgoing edges).
- None of the edges have weights associated with them.

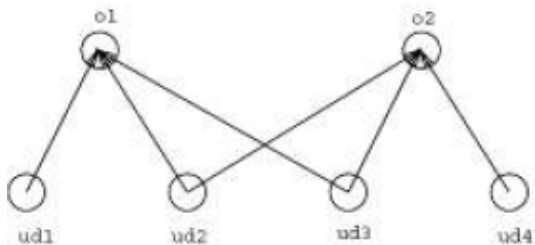


Figure 2: A simple object dependence graph.

2.1.2 DUP for Simple Object Dependence

Graphs: The application program must determine an appropriate correspondence between underlying data and vertices of the object dependence graph G . For example, a vertex corresponding to underlying data may represent a database table. Another vertex

corresponding to underlying data may represent portions of several database tables. There are no restrictions on how underlying data may be correlated with nodes of G . The application program has freedom to pick the most logical and/or efficient system.

Each object has a string obj_id known as the object ID which identifies the object. Similarly, each node representing underlying data has a string ud_id known as the underlying data ID which identifies it. The application program informs cache managers that an object has a dependency on underlying data via an API function:

```
add_dependency(obj_id, ud_id)
```

Whenever underlying data corresponding to a node in G changes, an application program notifies cache managers via an API function:

```
underlying_data_has_changed(u ud_id)
```

The cache managers then invalidate all cached objects having dependencies on underlying data. Referring to Fig 2, $underlying_data_has_changed(ud4)$ would cause $o2$ to be invalidated. The function call:

$underlying_data_has_changed(ud2)$ would cause both $o1$ and $o2$ to be invalidated. Cache managers maintain cache directories containing information about cached objects. Directory information for a cached object with one or more dependencies on underlying data includes the object ID and an incoming adjacency list containing all underlying data ID's corresponding to underlying data which affect the value of the object. Figure 3 depicts the incoming adjacency lists corresponding to the graph in Figure 2. Cache managers also maintain hash tables containing pointers to outgoing adjacency lists for nodes in G corresponding to underlying data. Hash tables are indexed by underlying data ID's. Each outgoing adjacency list contains the object ID's of objects whose values depend on the underlying data represented by the underlying data ID.

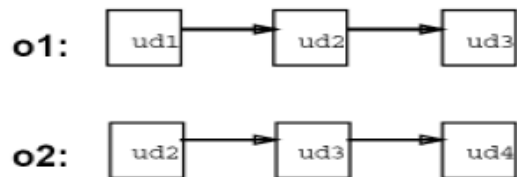
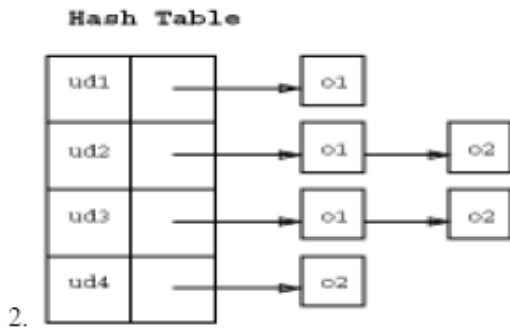


Figure 3: Incoming adjacency lists corresponding to the graph in Figure 2.



2. Figure 4: The hash table corresponding to the graph in Figure 2.

Figure 4 depicts the hash table corresponding to the graph in Figure 2.

An invocation of the **API** function:

add_dependency(obj_id,ud_id)

adds a new edge to G by adding *obj_id* to the outgoing adjacency list for and to the incoming adjacency list for *obj_id* An invocation of the API function:

underlying_data_has_changed(ud_id)

is implemented by invalidating all objects on the outgoing adjacency list for *ud_id*. It is sometimes desirable to delete nodes from G. For example, all dependencies underlying data node could become obsolete in which case the node would no longer be needed. Similarly, an object could go away which would make its node in G unnecessary. Object nodes are removed from G by removing the object ID from outgoing adjacency lists for all nodes on the incoming adjacency list for the object and removing the incoming adjacency list for the object. Underlying data nodes are removed from G by removing the underlying data ID from incoming adjacency lists for all nodes on the outgoing adjacency list for the underlying data node and removing the hash table entry for the underlying data node.

2.2. Prefetching: One of the key techniques use to obtain high cache hit ratios is to calculate and cache new versions of frequently referenced pages, relative to their update characteristics, immediately after it is determined that the pages are obsolete instead of invalidating the pages and waiting for them to be loaded on demand. Consequently, once a frequently requested page is cached, the large number of future requests for he ge always result in a cache hit. This technique is effective because popular dynamic pages are often requested far more frequently than they are updated, as demonstrated by analysis of request and update patterns. In more recent HAWS, whenever data changed, the DUP algorithms are used to first identify the pages affected by the data change. The appropriate pages were regenerated and the stale pages replaced in cache in a single atomic operation. Cache misses were almost never observed; therefore, even during peak periods, the system was not particularly busy and had considerable excess capacity. When a page changed,

the system would regenerate the page once and store the updated page in multiple caches.

3. PERFORMANCE ANALYSIS:

3.1 Performance measurements from an actual system:

For performance measurement of the system, we used Web Tress benchmark. The benchmark sends requests to the system. It acts as simulated web browser. It is capable to use 10 simulated web browsers. The test was conducted for 3 minutes by using 10 clients. Each client sent random requests to the web server. The test was conducted as follows.

1. Test in which the requests for dynamic pages were made to web server without using cache manager.
2. Test in which the requests for dynamic pages were made to cache manager.

The hits and misses were measured properly by cache manager. The Web server was having mixing of JSP and HTML pages. Out of that we used only two pages for update. Each simulated client clicked for these two pages. We updated these two pages after the interval of 2 seconds.

3.2.1 Analysis of the system without using cache manager:

The detailed log of test when it was conducted without cache manager is given below:

Completed Clicks: 10 with 0 Errors (=0.00%)

Average Click Time for 10 Users: 9 ms Successful clicks per Second: 1.97 (equals 7,091.11 Clicks per Hour)

Results of complete test

** Results per URL for complete test **

URL#1 (): Average Click Time 274 ms, 330 Clicks, 0 Errors

Total Number of Clicks: 330 (0 Errors)

Average Click Time of all URLs: 274 ms.

We observed that without using cache manager the average hit time was 274 ms. This was because for each time the client requested a dynamic web page, the server executed that page and gave reply back.

The Table No.1 shows that, all clients made 34-35 requests out of that each one got 33-3.1 Performance measurements from an 34 hits. The average click time was in between Actual System 201-325 ms. The data transfer rate was in between 14.75 to 23.82 kbit/s.

User No.	Clicks	Hits	Errors	Avg. Click Time [ms]	Bytes	kbit/s
1	35	34	0	325	20,395	14.75
2	35	34	0	313	20,395	15.35
3	34	33	0	323	19,795	14.86
4	34	33	0	308	19,795	15.59
5	35	34	0	267	20,395	18.01
6	34	34	0	262	20,395	18.30
7	35	34	0	241	20,395	19.91
8	35	34	0	230	20,395	20.90
9	34	33	0	228	19,795	21.08
10	35	34	0	201	20,395	23.82

Table No. 1: Result per user without using cache manager.

3.2.2 Analysis of the system using cache manager:

Now detailed log of test when the test was conducted with cache manager is given below. The considerable change has observed. In this test the requests were made to cache manager. We used one dynamic page for analysis. The clients were used to request that page only. It was also considered that the page modifies after 2 seconds. Thus, continuously the updated after the interval of 2 seconds and requests were made to that page. All client clicks resulted in cache hits. Only for two to three times the requested page resulted in miss, this was due to the delay in invalidation and updation of the page in cache.

The detailed analysis of the system using cache manager is given as follows:

** Results per URL for complete test **

URL#1 (): Average Click Time 5 ms, 350 Clicks, 0 Errors

Total Number of Clicks: 350 (0 Errors) Average Click Time of all URLs: 5 ms

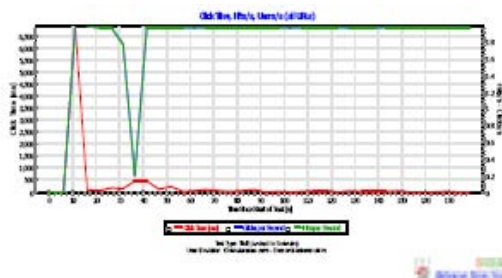


Figure 5. Click Time, Hits/s, Users/s without using cache manager

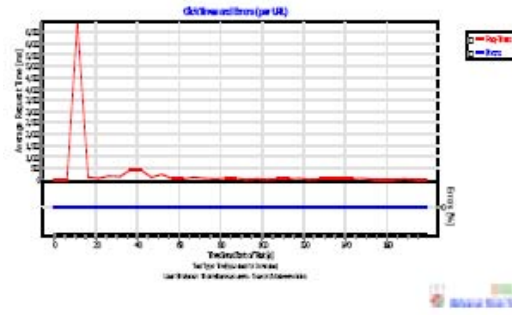


Figure 6 :Click times and Misses without using cache manager

It is observed that by using cache manager and successfully updating dynamic pages in cache by using DUP algorithm, the average hit time is 5 ms. All client requests were successfully satisfied from cache. There were no cache miss, though the dynamic page is made obsolete in cache and requested from the server for each 2 seconds time interval.

The Table No.2 shows that, made 37-36 requests out of that each one got 35-36 hits. The average click time is in between 4 -10 ms. The data transfer rate is in between 489.73 to 1,092.09 kbit/s.

User No.	Clicks	Hits	Errors	Avg. Click Time [ms]	Bytes	kbit/s
1	37	36	0	10	21,600	489.73
2	37	36	0	4	21,600	1,075.26
3	37	36	0	4	21,600	1,071.74
4	37	36	0	4	21,600	1,092.09
5	36	35	0	5	21,000	1,010.80
6	36	35	0	5	21,000	960.56
7	36	35	0	5	21,000	1,019.49
8	36	35	0	5	21,000	1,052.72
9	36	35	0	5	21,000	953.34
10	36	35	0	5	21,000	1,020.85

Table No. 2: Result per user using cache manager.

The time taken by first test was 90,497 ms and the total clicks made by clients are 330, the average click time is 274 ms. Whereas the time taken by second test which was carried out by using cache manager is 1824 ms and clicks made by the clients are 350, while the average click time is 5 ms.

Comparing two tests it is found that the page was difference between average click time is

269 ms and the difference between data transfer rate is 954.60 kbit/s.

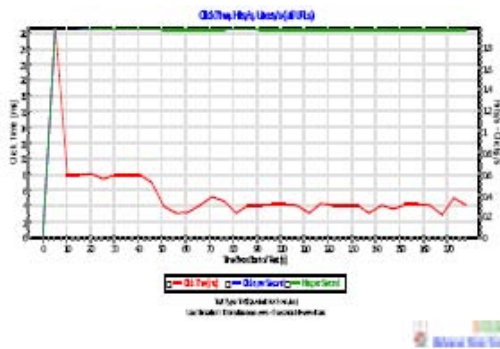


Figure 7. Click time, Hit/s, Users using cache manager

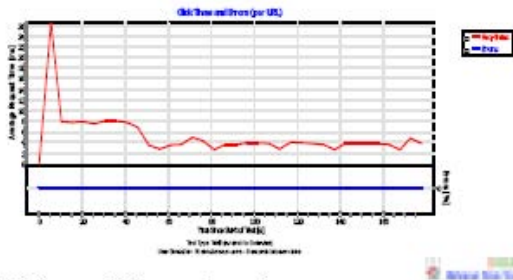


Figure 8. Click time and Misses using cache manager

4. CONCLUSION:

The system presented is general multitier architecture and a set of algorithms that have been deployed for high-performance serving of dynamic content to many clients at HAWS. This system design is based on analysis of the properties of the request and update patterns at HAWS environments and on the analysis of the performance properties of the architectures and algorithms. Analysis of the system design based on data and measurements from real deployments at HAWS demonstrate significant performance benefits, which have proven to be consistent with considerable experience using this system design at subsequent HAWS. In particular, the system is able to achieve cache hit ratios close to 100% compared with 80% for the system which did not use these architectures and algorithms. Proper deployments result in cached data which is almost never obsolete by more than a few seconds, if at all. Moreover, this system design serves dynamic content at the performance level of static content. There has been heavy demand for technologies to ensure timely delivery of fresh dynamic content to end-users. Fragment based generation and caching of dynamic web content is widely recognized as an effective technique to address this problem. Manual fragmentation of web pages by a web administrator or web page designer is expensive and error-rone; it also does not scale well. A fragment is considered to be interesting if it is shared among multiple pages or if it has distinct lifetime or personalization characteristics. This scheme is based on analysis of the web pages dynamically generated at given web sites with respect to their information sharing

behavior, personalization properties and change patterns.

5. FUTURE WORK:

There are a number of future research directions to extend and improve this work.. The present system is having only one cache and all clients communicate with that cache. It is also possible that the design can have multiple caches. Each cache containing the same copy of the object. When underlying data changes all copies of object in all caches invalidates and the new object copy is updated in all caches .Thus, multiple caches can be used for speedup and fast delivery of web content to the clients.

6. REFERENCES:

[1] A. Iyengar and J. Challenger, "Improving Web server performance by caching dynamic data," in Proc. USENIX Symp. Internet Technologies and Systems, Dec. 1997, pp. 49–60.
 [2] A. Iyengar, E. MacNair, and T. Nguyen, "An analysis of Web server performance," in Proc. IEEE GLOBECOM, Nov. 1997, pp. 1943–1947.
 [3] J. Challenger, A. Iyengar, K. Witting, C.Ferstat, and P. Reed, "A publishing system for efficiently creating dynamic Web content," in Proc .IEEE INFOCOM, Mar. 2000, pp. 844–853.
 [4] J. Challenger, A. Iyengar, and P. Dantzig, "A scalable system for consistently caching dynamic Web data," in Proc. IEEEINFOCOM, Mar.1999, pp. 294–303.
 [5] A. Aho, J. Hopcroft, and J. Ullman, "The Design and Analysis of Computer Algorithms." Reading, MA: Addison-Wesley, 1974.
 [6] A. Iyengar, J. Challenger, D. Dias, and P.Dantzig, "High-performance Web site design techniques," IEEE Internet Computing, vol. 4, no. 2, pp.17–26, Mar./Apr. 2000.
 [7] J. Song, A. Iyengar, E. Levy, and D. Dias, "Architecture of a Web server accelerator," Computer. Networks, vol. 38, no. 1, pp. 75–97, 2002.
 [8] M. S. Squillante, D. D. Yao, and L. Zhang, "Web traffic modeling and server performance analysis," in Proc. 38th IEEE Conf. Decision and Control (CDC), 1999, pp. 4432–4439.
 [9] J. Challenger, P. Dantzig, A. Iyengar, M. S.Squillante, and L. Zhang, "Efficiently serving dynamic data at highly accessed Web sites," IBM Research Div., Yorktown Heights, NY, Tech.Rep. RC22823, 2003.
 [10] V. Holmedahl, B. Smith, and T. Yang, "Cooperative caching of dynamic content on a distributed Web server," in Proc.7th IEEE Int. Symp. High Performance Distributed Computing, July 1998, pp. 243–250.
 [11] Laxmish Ramaswamy, Arun Iyengar, LingLiu Fred Dougli " Automatic [12]Detection of Fragments in Dynamically Generated WebPages".
 [12] Document Object Model-3CRecommendation. <http://www.w3.org>.