Controlled Caching of Dynamic WWW Pages

Costas Vassilakis[1], Giorgos Lepouras[1]

[1]University of Athens, Department of Informatics and Telecommunications
Panepistimiopolis, TYPA Buildings, Athens 157 71 Greece
{C.Vassilakis, G.Lepouras}@di.uoa.gr

Abstract. In order to increase flexibility and provide up-to-date information, more and more web sites use dynamic content. This practice, however, increases server load dramatically, since each request results to code execution, which may involve processing and/or access to information repositories. In this paper we present a scheme for maintaining a server-side cache of dynamically generated pages, allowing for cache consistency maintenance, without placing heavy burdens on application programmers. We also present insights to architecture scalability and some results obtained from conducted experiments.

Introduction

Content offered by WWW servers can be classified in two categories, *static* and *dynamic*. Static content comprises of HTML pages and images stored in files; dynamic content is created by the WWW servers upon receipt of a relevant request, through the execution of server-side program/script (CGI, ASP, JSP etc). Usually, these programs accept some parameters, included in the request and extract data from an information repository (e.g. a database) to formulate the HTML page that will be sent back to the requestor. While in the early days of WWW dynamical content was rare, its share is constantly increasing due to the need for increased flexibility in HTML page formulation and the desire to provide up-to-date information.

Dynamic content, however, increases server workload, since it necessitates the execution of a (possibly costly) code fragment, while additional overheads (environment setup, housekeeping etc.) are inherent in such an approach. This workload may prove overwhelming for heavily accessed servers, and techniques used for static HTML pages (e.g. client-side caching, proxy servers) may not be employed for reducing it. This is due to the fact that such techniques do not take into account the particularities of dynamic content, such as the existence of parameters and the possibility that either data, or even the program that were used to formulate the page may have changed in the meantime. This paper presents a scheme for maintaining a *server-side cache* for dynamic content, which may reduce drastically server workload, without compromising the advantages offered by the usage of dynamic content.

In the rest of this paper, section 2 covers related work and outlines open issues. System architecture is presented in section 3, and cache consistency maintenance is discussed in section 4. Section 5 focuses on scalability and resource utilization and section 6 presents experimental results. Section 7 concludes and outlines future work.

Related Work

In [11] a cost model for the materialization of web views, i.e. subsets of a server's dynamically computed information content, is presented. In [12] pre-generation of Web pages is discussed and changes in the information repository are addressed, but no algorithm for selecting the most appropriate pages for caching is presented. Caching of dynamic content is also discussed in [3], [13] and [6] and [9] presents techniques for reducing a web server's workload, regarding dynamic content creation.

In [15], two categories of dynamic content caching are described, *active cache* and *server accelerators*. In *active cache* [4] servers supply cache applets that are attached to documents and are executed when a user "hits" a cached object. Although this scheme reduces network load, it has a significant CPU cost. On the other hand, *web server accelerators* reside in front of web servers. Such techniques were used to improve performance at the Web Site for the 1996 Olympics [10] and for the 1998 Winter Games [6]. In the 1998 Games the Data Update Algorithm was used to maintain data dependence information between cached objects and the underlying data, enabling cache hit rates close to 100% compared to 80% of an earlier version. However, this approach suffers when it comes to web sites that offer a large number of dynamic pages [19], because it depends on keeping up to date a fine-grain graph that describes dependencies among each web page and the underlying data. In [19] a different approach is proposed, where the caching mechanism resides behind the web server and the fine grain method described in [6, 10] is supplemented with coarse grain dependencies between data and groups of dynamic pages. The caching algorithm uses a URL class based invalidation method and selective precomputing.

Commercial products implementing caching of dynamic content exist as well. Cold Fusion [1] offers the capability to designate programs that produce non-changing dynamic pages, and Cold Fusion engine arranges so that outputs from this program are cached and reused, while XCache [18] can be integrated into the IIS Web server to maintain a cache of dynamically generated Pages, providing an additional API for update applications so as to inform the cache engine for outdated pages.

All techniques presented insofar, however, require from application programmers an amount of additional development and maintenance for adapting programs to the caching scheme, for reasons of update propagations and consistency management. Moreover, invalidated pages are designated using proprietary specifications or APIs, increasing thus overall system complexity. Finally, most implementations are coupled to specific web servers, reducing interoperability and portability.

Maintaining a Cache of Dynamically Generated Pages

A server-side cache of dynamic content may be maintained in a controlled fashion using the architecture depicted in Figure 1. The proposed scheme complements the existing web site installation with two additional software modules, the *cache manager* and the *update manager*, which are responsible for maintaining the cache and ensuring that the cache is consistently updated or invalidated, when data is modified. The proposed scheme places the new modules as a *front end* to the existing installation, without any need to affect the latter; some additions may only be needed for update programs, as explained later. This characteristic was one of the design goals, since site administrators would be reluctant to employ a scheme that would require major changes to their site.

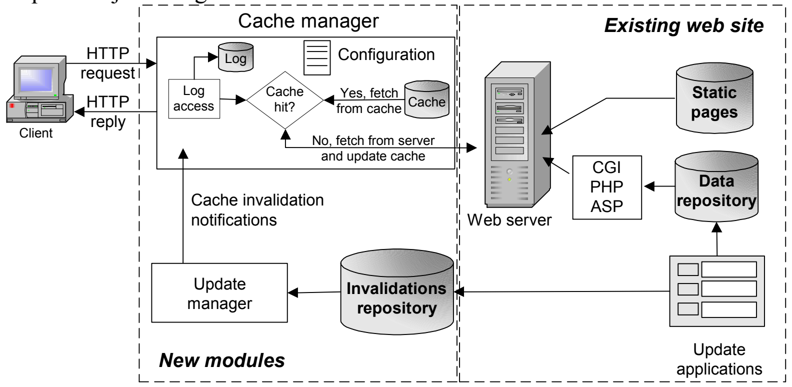

Fig. 1. Architecture for maintaining a cache of dynamic content

Upon start-up, the Cache Manager reads its configuration, which is derived from the class-based cache management for dynamic web content, described in [23]. Under this scheme, web pages are grouped into classes, based on page URLs and client information. The URL information used for classifying pages includes network paths, program names and parameters. Client information may complement the specification of a class and include cookies, client domain names or IPs, browser information (e.g. HTTP_ACCEPT_ENCODING, HTTP_USER_AGENT), or any other piece of information designated in the HTTP protocol for the server-side program environment. Figure 2 presents example cache specifications for three classes. We note that our approach does not use the *Dependence* specification described in [23], since a different mechanism (see section 4) is used for change tracking and page invalidations.

```
URL-Class:
  http://www.example.org/php/comments.php?op=submit
   Cachable: No
   Server: local-www-srv:80
URL-Class: http://www.example.org/php/search.php
   Cachable: no
   Server: local-www-srv:80
URL-Class: http://www.example.org/php
   Cachable: yes
   Page-ID:_cookie:LANGUAGE
   Server: local-www-srv:80
```

Fig. 2. Cache specification for two document classes

The first class designates that pages generated by server-side program (SSP) *comments.php* with the parameter *op* set to the value *submit* are not eligible for caching (due to side effects of execution); the second class denotes that search results are not cacheable either (due to the small probability that two users will use the same search text). The third class defines that pages generated by SSPs within the *php* catalogue are cacheable, but additionally states that the page contents depend on the value of the *LANGUAGE* cookie. In all three cases, the *Server* lines (an addition to the specifications presented in [23]) designate that the web server producing these pages is named *local-www-srv* and may be contacted at port *80*. The name *local-www-srv* need not be known to the clients, since only the Cache Manager uses it.

After initialisation, the Cache Manager awaits for web requests. For each request, it checks whether the requested page has been cached or not. A *page*, in the context of dynamic content, is identified by the URL, the parameters passed to the SSP and any cookies and HTTP variables pertaining to the class in which the request falls. If the page is found in the cache, it is returned to the client immediately. If not, the Cache Manager requests the page from the web server and forwards the results to the client. In order to rapidly determine if the target page exists in the cache, the Cache Manager employs the algorithm described in [16], which computes an MD5 checksum of the requested URL and matches it against the checksums of the cached pages.

Subsequently, the Cache Manager inspects its configuration to determine whether the page just served is eligible for caching. In this case, the page is placed in the cache, so that it becomes available for later requests. If the page is not eligible for caching, the results returned to the client are simply dropped. In the case that the cache is full and a page needs to be inserted to it, the Cache Manager removes some previously cached pages from the cache to make space available for the new page. The page(s) that will be removed are determined using the Greedy Dual-Size [5] algorithm, and more specifically the variation aiming to maximize the hit ratio, since it has been found to achieve superior performance, compared to other algorithms within the context of web caching [2].

The Cache Manager cooperates with the *Update Manager* module, described in section 4, to guarantee that the pages sent to the clients are not outdated, due to changes in the data or the server-side programs that generate the pages.

Handling Data and Program Changes

Pages generated by SSPs and stored within the cache may become obsolete, since the underlying data, or even the server-side programs themselves may be modified. In these cases, the affected pages in the cache must be located and either be removed, or re-generated. The responsibility for update tracking is assigned to the *Update Manager*, which is a distinct software module that may be run on the same machine as the cache manager or on a different one. The *Update Manager* monitors the *invalidations repository*, to locate *patterns* describing the pages that have become obsolete. Each such pattern contains the following information:

1. A designation of the program(s) that have generated the affected pages.
2. The parameters and the respective values that must match the server-side program's invocation data, so as to consider the page invalid.

Figure 3 presents some examples of invalidation patterns.

Each invalidation pattern is complemented with a *criticality designation*, which draws values from the domain *{hard, soft}*. A value of *soft* indicates that the outdated version of the page *may* be used as a reply to the requesting client, in the event of high server load, whereas a *hard* criticality designation states that sending outdated pages is unacceptable, and the respective cached pages should be invalidated immediately. Updates having a *soft* criticality designation are assigned to a low-priority thread of the Cache Manager, which uses idle CPU cycles to refresh outdated pages.

(Path, Parameters)	Remarks
("http://www.example.org/categories.php", "catid=2")	All pages generated by *categories.php* with the parameter *catid* equal to *2*, regardless of the values of other parameters or cookies.
("http://www.example.org/admin/*", "")	All pages generated by any program in the *admin* directory, regardless of parameters.
("http://www.example.org/index.php", "__cookie:LANGUAGE=en")	All pages generated by *index.php* with the cookie *LANGUAGE* set to the value *en*.

Fig. 3. Examples of patterns within the invalidations repository

Once the Update Manager detects a new invalidation entry within the invalidations repository, it reads the respective data and propagates them to the Cache Manager, so that the respective page invalidations or regenerations may be performed. The Cache Manager communicates with the Update Manager using a private protocol, which is not needed to be known to the applications that modify the data repository, contrary to the practice employed in [23] and [10]. These applications need only insert the corresponding invalidation entries to the invalidations repository (which may be a database table, an operating system file etc.), using familiar programming techniques.

Using an additional repository for registration of invalidation patterns and the introduction of the update manager were design decisions, aiming to relieve the update application programmers from using proprietary APIs, the need to know details about the number of Cache Managers and their addresses, or handling communication errors. Furthermore, using a standard information repository for invalidations, such as a database, facilitates interventions from the Web administrators that could not be foreseen at the time of update application development. For instance, in order to force the invalidation of all pages generated by the */categories.php* script (e.g. due to changes in the script), the Web administrator may append the respective entry to the invalidations repository by using e.g. SQL:

```
insert into cache_invalidations(script_path, params, criticality)
values('http://www.example.org/categories.php', '', 'hard');
```

If invalidation information were communicated to the cache manager through an API, it would be necessary to write compile and execute a –simple– program that would arrange for sending the respective information. Moreover, invalidation records use a format very familiar to web programmers and administrators; therefore it is expected that web site maintenance will run more smoothly, compared to cases that employ proprietary protocols or specifications.

Finally, if the data repository supports active features, application programmers may be totally relieved from the burden of registering the invalidation patterns within the invalidations repository. This approach is outlined in section 4.2.

Locating affected pages within the cache

When the Cache Manager receives an invalidation notification, it must locate all pages matching the patterns for the script path and the parameters and either remove them from the cache or refresh them (in immediate or deferred mode, as specified by the *criticality* designation). Since the model presented in this paper allows for specifying *parameter subsets* or *wildcards* within the script paths, the MD5 checksum of the URL, stored for the purpose of serving client requests, cannot be used to locate the affected pages, since hash functions can only be used for full key search [8]. To avoid scanning the full table of cached pages, which would be expensive, the following techniques are used, together with the primary MD5 checksum hashing:

1. An additional MD5 checksum is computed on the program path portion of the URL for each page within the cache, and a hash structure is maintained based on this checksum. This hash structure is used to locate the pages generated by a specific server-side program, when the *script path* field of an invalidation entry does not contain wildcards. For each of these pages, standard substring searching is used to further filter pages matching the specified parameters, if necessary.
2. A secondary B-tree index is built on the *script path* field of the cached pages. This index is used to locate the affected pages when the invalidation entry's *script path* field *does* contain wildcards. Since the maintenance of this index is more costly than the maintenance of MD5 hash tables, an administrative option is provided to disable it; in this case, however, wildcard matching in script paths is not supported.
3. An *optimisation hint* may be attached to an invalidation entry, specifying that the page designation is *complete*, i.e. it includes *all* parameters to the SSP. In this case, the primary MD5-checksum hashing algorithm may be used to locate the *single* page that has become obsolete, minimizing thus housekeeping overheads.

Separating invalidations from update applications

In the approaches presented in the literature insofar, application programs that modify the data repository need to notify in some way the Cache Managers regarding the pages that have become obsolete. This is a burden for programmers and introduces another source of application maintenance activities, when the mapping between data items and dependent web pages changes. In these cases, all programs modifying the respective data need to be tracked and updated.

The approach presented in this paper allows for exploitation of active features offered by the data repository to relieve programmers from the extra coding and maintenance. Such active features may be found in many DBMSs, e.g. *triggers* in Oracle, *rules* in INGRES etc. These active features may be used to implement automatic entry addition to the invalidation repository when the underlying data changes. For instance, the PL/SQL code

```
create trigger category_invalidations
after insert or delete on stories for each row begin
insert into cache_invalidations (script_path, params, criticality)
values('http://www.example.org/categories.php', 'catid=' || catid,
      'hard');
end;
```

arranges for automatically inserting into the cache invalidations repository the appropriate record for invalidating the pages generated by the script *categories.php*, when its *catid* (category id) parameter matches the category id of an inserted or deleted item in the *stories* table.

Similar results may be obtained by using *stored procedures* within the information repository, in order to modify the data, instead of direct SQL (or other information repository-dependent) statements. For instance, the addition of a story may be performed using the statement

```
exec procedure insert_story(800, 3, 'New story', 'Very interesting');
```

where *insert_story* may be defined as

```
create procedure insert_story(sid number, cid number, title varchar2,
body varchar2) as begin
insert into stories values(sid, cid, title, body);
insert into cache_invalidations(script_path, params, criticality)
values(' http://www.example.org/categories.php', 'catid=' || cid,
   'hard');
end;
```

Both the trigger-based and the stored procedure-based techniques are feasible because invalidations are stored within a data repository, instead of being communicated to the Cache Manager via a custom interface, which –in general- cannot be directly used from within a database. Moreover, both cases present the additional advantage of *centralising* the mapping between updated data and affected pages; thus when this mapping changes, only a single modification is required (within the database schema), rather than one update for each affected application.

Architecture Scalability and Resource Utilisation

When the number of requests to the web site increases, the cache manager may become the performance bottleneck of the installation. This may be tackled by using an *array of cache managers* rather than a single one, as depicted in Figure 4. Client requests may be distributed amongst the servers within the array using any appropriate technique (e.g. clustering, round-robin DNS, or specialised hardware [7]).

In this configuration, cache managers maintain independent caches, serving client requests using the algorithm described in section 3. Although in some cases it would be preferable for a Cache Manager to fetch a pre-generated page from a *sibling cache*, rather than from the web server, the current implementation does not support ICP [19], cache digests [14], or any protocol for communication between cache servers.

When updates to underlying data take place, the invalidations repository is populated with the appropriate entries, as presented in section 4. The Update Manager monitors new additions and arranges for communicating the relevant data to the Cache Managers, whose addresses are registered within the Update Manager's configuration file. Each Cache Manager proceeds then to the invalidation of the affected pages, following the algorithm described in section 4.1. The Update Manager is the sole responsible for handling communication errors, retransmissions to failed nodes etc.

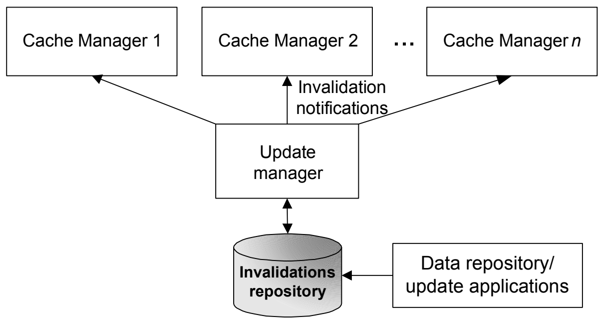

Fig. 4. Load balancing through introduction of multiple caching servers

We would like to note that although the web server, the Cache Manager and the Update Manager appear as distinct modules in Figure 4, it is possible that two of these modules (or even all of them) run on the same machine, in order to save resources. In medium configurations it is expected that only one Cache Manager will be used and the Update Manager will be hosted in the same machine as the cache manager. In large installations, the Cache Manager might be placed in a separate machine. Finally, a single Cache Manager may be used to cache multiple web sites, if this is desirable.

Experiment Results

In order to assess the various aspects of our approach, a number of experiments were conducted. The experiments were selected so as to vary in different aspects, such as update frequency, average page size, update propagation criticality and the existence of personalised pages. For each experiment, stress tests were performed under different loads, so as to quantify the benefits derived from using the caching scheme.

The first experiment was conducted on a web portal implemented using the PHP-Nuke system. In this experiment, a number of stories were added daily; when a story was added, the main page (*/index.php*) should be refreshed instantly, whereas pages hosting news falling in specific categories (*/categories.php*) were updated with softer time constraints. Personalised pages (implemented via cookies) were not cached, as log analysis showed that each such page was used only 1.2 times in average.

The second experiment was conducted for a sports site, offering “live” information about the scores of soccer and basketball games. Pages served from this site were classified in three categories: (a) a summary page for all soccer games (b) a summary page for all basketball games (c) one detail page for each match (basketball or soccer). In this experiment, the soccer summary page was updated once every two minutes, in average, whereas the basketball summary page was updated constantly as game scores evolved. Out of these two summary pages, only the first one was cached. Regarding detail pages, the average update rate for soccer pages was once every 3’:30”, while the corresponding rate for basketball match pages was once every 28”. Refresh information sent to clients dictated page refreshing every 30” for both soccer and basketball pages. “Impatient” users, however, often requested page refreshes every 3”-5”, especially during the last few minutes of games.

Both experiments were conducted using two site configurations. In the first configuration, a single machine hosted all software modules, i.e. the web server, the Cache Manager and the Update Manager. In the second configuration, two machines were used, with the first one running the Cache Manager, and the second one hosting the Update Manager and the web server. The results from these experiments are summarised in Figure 5.

Single-server	*Cache hit ratio*			*Avg. response time reduction*		
# clients	*50*	*250*	*500*	*50*	*250*	*500*
Web portal	62%	73%	78%	61%	72%	67%
Sports site	43%	60%	65%	51%	62%	59%
Dual-server	*Cache hit ratio*			*Avg. response time reduction*		
# clients	*50*	*250*	*500*	*50*	*250*	*500*
Web portal	62%	73%	78%	59%	70%	76%
Sports site	43%	60%	65%	49%	59%	66%

Fig. 5. Experiment results using a dual-machine configuration

In these results, it is worth noting that in the single-machine configuration, although the cache hit rate increases when the number of users grows from 250 to 500, the average response time reduction drops. This is due to *server saturation*, since the server cannot handle all three tasks under this heavy load. The situation is remedied when the cache manager is separated from the web server, in the second configuration, where the load is balanced between the two servers.

Conclusions – Future Work

This paper proposes a caching scheme for dynamic web pages, which isolates the caching mechanism from the web server. Although this separation duplicates the parsing of URLs and headers, it presents a number of advantages:

- the caching server can interoperate with any web server, thus no web-server specific implementations are required
- the two servers (web and caching) may reside on different systems
- any upgrade/modification to any of the servers does not affect the others.

The architecture presented minimises the additional workload for dynamic page developers, by allowing separation of the rules describing the dependencies between underlying data and derived pages from the application programs that perform the data updates. These dependencies can be stored within the database schema, in a centralised fashion, leaving update applications intact. This requirement can be easily met when the database offers triggering mechanisms, or stored procedures.

The caching scheme was tested in several cases, proving quite efficient and well behaved in cases of information repository or server-side program updates. The results showed that the proposed caching mechanism offers improved network performance with minimum CPU overhead. It should, however, be noted that the results are dependent on the nature of the dynamic pages. If the underlying data change too often (e.g. stock exchange), the caching scheme will probably not improve the web site performance. Further research is underway to form a set of guidelines and tools to aid developers in assessing the suitability of dynamic content for caching.

Future work will focus on supporting cache population using anticipatory rules, to cater for expected access patterns over time periods, and extending cache techniques to the client side, so as to limit the required network traffic. Cooperation between sibling caches, and its effect on performance is another area that will be investigated.

References

1. Allaire Corporation: Cold Fusion White Paper. Version 4.0. (1999). `http://www.allaire.com`
2. Arlitt M., Friedrich R., Jin T.: Performance Evaluation of Web Proxy Cache Replacement Policies. Proceedings of the 10th International Conference, Tools '98, Palma de Mallorca, Spain, (1998) 193-206
3. Atzeni P., Mecca G., Merialdo P.: Design and Maintenance of Data Intensive Web Sites. Proceedings of the Conference of Extending Database Technology, Valencia, Spain (1998)
4. Cao P., Zhang J., Beach K. Active Cache: Caching Dynamic Contents on the Web. Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98). (1998) 373-388.
5. Cao P., Irani S.: Cost-Aware WWW Proxy Caching Algorithms. Proceedings of USENIX Symposium on Internet Technologies and Systems. Monterey, California (1997) 193-206
6. Challenger J., Iyengar A., Dantzig P. A Scalable System for Consistently Caching Dynamic Web Data. Proceedings of the IEEE Infocom 99 Conference. New York, (1999)
7. Coyote Point Systems Inc. The Equalizer Network Appliance. Available through http://www.coyotepoint.com/equalizer.shtml
8. Elmarsi R., Navathe S. Fundamentals of Database Systems. Benjamin/Cummings Publishing Company Inc. (1994)
9. D. Florescu et al. Run-Time Management of Data Intensive Web-sites. Proceedings of the Workshop of Web and Databases (WebDB 99), Philadelphia, Pensylvania. (1999)
10. Iyengar A., Challenger J. Improving Web Server Performance by Caching Dynamic Data. Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California (1997).
11. Labrinidis A., Roussopoulos N. On the Materialization of Web Views. Proceedings of the Workshop of Web and Databases (WebDB 99), Philadelphia, Pensylvania (1999)
12. Pröll B. et al. Ready for Prime-Time – Pre-Generation of WEB Pages in TIScover. Proceedings of the Workshop of Web and Databases, Philadelphia, Pensylvania (1999)
13. Sindoni G. Incremental Maintenance of Hypertext Views. Proceedings of the Workshop of Web and Databases (1998)
14. Hamilton M., Rousskov A., Wessels D. Cache Digest specification - version 5. Available through http://www.squid-cache.org/CacheDigest/cache-digest-v5.txt
15. Wang J. A Survey of Web Caching Schemes for the Internet. ACM Computer Communication Review, (29) (1999) 36--46
16. Wessels D. Squid Internet Object Cache. http://www.squid-cache.org
17. Wessels D., Claffy K. Internet Cache Protocol (ICP), version 2 – RFC 2186. Available through http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2186.html
18. XCache Technologies, XCache Product Information. http://www.xbuilder.net/home/
19. Zhu H., Yang T. Class-based Cache Management for Dynamic Web Content. IEEE INFOCOM, 2001.