A Simple and Effective Caching Scheme for Dynamic Content*

Karthick Rajamani
Dept. of Elec. and Comp. Engg., Rice University
karthick@rice.edu

Alan Cox
Dept. of Computer Science, Rice University
alc@rice.edu

September 18, 2000

Abstract

As web sites increasingly deliver dynamic content, the process of content generation at request time is becoming a severe limitation to web site throughput. Recent studies have shown that much of the dynamic content is, however, better characterized as *pseudo-dynamic*, i.e., a dynamic composition of stored or static data. Consequently, caching the generated web pages may increase the web server's throughput if there is some temporal locality in the request stream.

In this paper, we perform a quantitative analysis of the benefits of caching for dynamic content using the e-commerce benchmark, TPC-W, as the workload. We implement caching through a simple and efficient Apache extension module, *DCache*, that can be easily incorporated into the current infrastructure for dynamic content delivery. Our DCache module uses conventional expiration times and our own *request-initiated* invalidation scheme as the methods for keeping the cache consistent. It also supports site-specific optimization by providing a mechanism to incorporate the priorities of specific web pages into the caching scheme. Our experiments show that we can obtain over 3 times the non-caching throughput with our caching approach.

1 Introduction

Recent studies of web-site content [13, 14] show that an increasing proportion of the content delivered is dynamic, i.e., content generated at the time of the request. The main bottleneck for such servers is content generation at request time. If all dynamic content were such that it could only be generated at request time, then there would be no other solution to improve performance but improving the speed of content generation. However, many recent studies [13, 14, 7] and an examination of popular web sites show that much of the dynamic content is really *pseudo-dynamic* in nature.

Pseudo-dynamic content is a dynamic composition of stored or static data. Two common examples of pseudo-dynamic content include user-specific web pages and selective content display (through queries on the data). In these cases, much of the request-time processing can be done ahead of time, based on past requests. This is because there is a lot of locality in the tailored content for the request stream, just as there is in requests for static objects. And, similar to the

* *Rice University Computer Science Technical Report TR 00-371*

exploitation of this locality by caching for static content, we can cache the pseudo-dynamic content also for improved performance. This is the basic premise behind caching for *dynamic* content.

One of the most popular approaches for dynamic content delivery is to query a database backend for request-specific data and present the data, suitably augmented, as the dynamic response to the request. This is a very common practice with e-commerce sites where the display and sales is through the web interface, while the customer, inventory and cache flow management is carried out through transactions on backend database systems where all the real data is resident. In such a system, the database querying and result transformation into web-pages at request time become the major bottlenecks in the system. A common approach, and popular one among database system vendors, is to use increasingly powerful database backends. An alternative one is to use caching of the dynamic content.

The issues involved in content caching for such a system are:

- What to cache, when to cache, and where to cache?
- How to keep the cache consistent, i.e., how to guarantee that a cached response does not deliver stale data ?
- How to do it with minimal resources, i.e., get maximum bang for the buck ?

The first two items in the above list are very much site dependent. The system supporting the site can at best provide a variety of alternatives with flexible mechanisms to choose between them. We provide and use mechanisms for these two that efficiently support our test application in this paper - these mechanisms can be effectively used for any similar e-commerce application. We address the third issue in detail - how to implement a low cost caching scheme with minimal alterations to the existing system? How good a performance can such a system achieve?

To address these questions we have built a caching system on top of 3 very popular open source software packages - the Apache web server [1], the MySQL database server [8], and the PHP web-scripting/application development language [9]. Our caching system is integrated into the existing infrastructure as an *extension module* to the Apache web server. Ours is a low cost, least intrusive, and minimal addition approach to caching. We utilize the spare resources on the web-server to cache dynamic content (acting just as if the server was serving more static content) to reduce the load on the database server - this increases the capacity to serve more requests, reduces user-response times etc. To evaluate our system we use the TPC-W benchmark [12] that has been specifically designed for evaluating entire systems that support e-commerce.

Our experiments give surprisingly good results. We obtain tremendous improvement in performance, measured as the number of completed interactions - over 3 times the non-caching performance. In addition, we provide simple mechanisms to maintain the validity of cached dynamic content which can be adopted among a large class of web-sites.

2 Caching Dynamic Content

Our approach to dynamic content caching integrates the caching technology with the web-server. Whenever, any request for dynamic content arrives at the web-server it gets transparently handled by our caching module that is part of the web-server. To do this, we provide the caching subsystem as an add-on-module for the web-server. Our caching approach could work with any general-purpose web-server. However, to better understand the details of our implementation it is necessary to have a basic knowledge of the architecture of our web-server, Apache.

2.1 Architecture of Apache

Here we provide a brief introduction to the architecture of Apache [1, 11]. On Unix systems, Apache uses a multi-process model to multiplex its operations. A single parent process is responsible for supervision, while all the child processes participate in serving incoming requests. The request handling task is broken down into multiple phases (nine in the current version). The core server code can handle basic functionality for each phase of a HTTP request. However, *extension modules* are used to further customize how Apache functions for each of these phases. The modules do so by registering short code routines called *handlers* that Apache would invoke at the appropriate moment. A module can register handlers for multiple phases. And multiple modules can register handlers for the same phase. Apache invokes the multiple handlers for a phase in the reverse order in which they are registered. If no modules are registered for a particular phase, it will be handled by a default routine from the Apache core.

Our caching module, **DCache**, registers a handler for the *content delivery* phase last, for all request types that require dynamic content e.g. pages with server-side scripting like Perl, PHP etc. This means, that whenever a request enters the content delivery phase for a page requiring dynamic content, Apache would first invoke the handler registered by our DCache module.

Apache also provides facilities for modules to invoke *internal requests* and *subrequests*. An internal request is processed almost like any other request, but is actually a request generated within Apache. A subrequest is a special form of internal request. At any phase of request handling, a handler can create a subrequest for a file or URI. Other handlers would service this subrequest just as if it came from a client outside the server. After the subrequest is serviced the control is then returned to the invoking handler.

The *subrequest facility* allows the DCache module to service any request for dynamic content by using pre-registered handlers of other modules for that particular content type. Once the DCache module intercepts (or receives first) the request for a dynamic content page, it invokes the handler registered by a module for that content type to obtain the response for that request.

2.2 DCache - The Cache Module for Dynamic Content

DCache provides dynamic content caching by caching responses to previous requests for the *same* content. It classifies requests into two broad categories:

- Cacheable requests - For this category, if a previous response has been cached by DCache, it is served from the cache. If not, it makes a subrequest that will be served by the appropriate handler. Once the subrequest returns, the DCache module gathers the response (generated during the subrequest) and caches it. A request cached by the DCache module can have an expiration time. If so, once the expiration time has elapsed, any new request for that content results in a subrequest to obtain fresh content for that request.

- Non-cacheable request - For this category, the content handler of the DCache module just *declines* the request. This results in Apache invoking the previously registered (next in priority) handler of another module for the content handling phase for that request.

Once the response for a request gets cached it is likely that the cached response would turn out to be invalid for the request at a later point in time - basic nature of dynamic content. DCache supports two procedures for invalidating cached responses. One is through expiration times and the other is through a process we call *request-initiated* invalidation. The first is straight-forward: when the response is generated for a dynamic content request, the appropriate handler also generates an expiration time for the request. This information is passed to the DCache's content handler through the headers generated for the subrequest. DCache then stores this expiration time along with the cached response. When a *cacheable* request arrives after the expiration time has elapsed, DCache generates a new subrequest for it, instead of returning the cached response. For *request-initiated* invalidation the site builder registers rules with the DCache handler to identify cached responses that are made stale by the servicing of a specific request - these can be in the form of simple if-then rules e.g.

if *request* is for document A - with *args* x,y,z
then *invalidate* cached responses B, C, ..

Whenever a new request is received, whether it belongs to the cacheable or non-cacheable category, in addition to handling the specific request, any request-initiated invalidations registered for it are also carried out by the DCache module.

2.3 Cache implementation

This section details our current implementation of the actual cache. The multi-process nature of the Apache server and the support of our specific Unix platform strongly influences this design of the cache. Given the multi-process nature of Apache, we need a cache that should be shareable between multiple processes. This is to make the response cached by one Apache process available for use by another Apache process. The two obvious choices to obtain this sharing between processes is - shared memory and memory mapped files. There is little to choose between the two in terms of access overhead in our Unix platform, FreeBSD.

Memory-mapped Cache

We go with memory mapped files to share the cached responses as it makes it easier to implement the DCache as a transparent module that intercepts the dynamic content requests. Apache supplies a *file descriptor* to each module's content handler to which the handler writes the response. This file descriptor is normally that of the connection to the client, and thus the response written by the handler is directly sent to the client. The DCache replaces this file descriptor with that of the memory mapped file when making the subrequest (as explained in 2.1). Thus the 'real' handler for a particular dynamic content type does not need to know the presence of the DCache module to supply it with the response to cache. Once the response has been written into the memory mapped file, it is sent to the actual connection by the DCache module.

The cache is composed of an in-memory hash table and memory mapped files that contain the cached responses. There is a fixed number of buckets in the hash table, one for each cached response. The cached response corresponding to each hash bucket is backed by a specific memory mapped file. A hash function maps the URL (with the arguments if necessary) of the cache request to a specific hash bucket. This hash function can be site-specific for better characteristics (a default hash function is also supplied with the system).

The hash table also needs to be shared between the Apache processes. We do this, again, by having it backed by a memory mapped file. However, the mapping is done such that the updates to the table are not reflected to disk, thus retaining the performance advantage of an in-memory hash table.

Synchronization

We need efficient synchronization between the multiple apache processes when they attempt to concurrently read/write to the cached responses or to the hash table. We use efficient atomic reader-writer locks implemented with code borrowed from the FreeBSD source. There is a lock per hash bucket to eliminate unnecessary synchronization overheads. The locks themselves are shared by allocating them at the beginning of the shared mmap'd file for the hashtable.

Cache Replacement

As a result of the our cache design, we have the flexibility to have a different replacement policy per response. Since the hash-function can be site-specific (user-supplied) it can regulate what set of responses/requests get mapped to what hash buckets. High-priority (frequently requested) pages can have their own hash buckets, while multiple lower priority pages can be hashed to the same hash bucket. This is a simple but very effective mechanism to provide a differentiated cache priority scheme to account for the different levels of request priority that a web-site might like to support.

3 Related Work

3.1 Keeping caches consistent

The following papers address the issue of maintaining caches for dynamic data consistent. The Data Update Propagation (DUP) algorithm [3, 4] maintains an object dependency graph (ODG) that links the underlying data with the objects (web-pages) dependent on the data. Challenger et. al. discuss the application of DUP to enabling a cache invalidation scheme, where the invalidations can be generated as a result of database triggers. The dependencies between the underlying data and the objects are to be specified by the application programs in general while some could be generated dynamically. The caches in these systems are at the web-site where the dynamic content is also generated. However, they are managed on separate cache servers, distinct from the web-server.

Challenger et. al. [5] extend the use of the ODG for a publishing system for creating dynamic content. Web pages are composed from *fragments*. Relationships between Web pages and fragments are represented through ODGs. Users specify the composition of web pages by creating templates in a markup language. Templates are parsed to determine inclusion relationships between fragments and web pages - which are represented in the ODG. A *Trigger Monitor* process uses the ODG to update stale pages.

We are distinguished from much of this work by our lightweight solution to the problem. Our DCache approach requires minimal modifications to the existing infrastructure for serving dynamic content while obtaining to a similar extent, the benefits of dynamic content caching.

3.2 Caching Dynamic Content at Proxies

The papers in this section discuss a somewhat orthogonal issue of caching dynamic content at proxies. Smith et. al. propose Dynamic Content Caching Protocol [10] that allows individual content generating applications to specify how their results should be cached and/or delivered. They use HTTP/1.1 extensions for the *Cache-Control* directive to specify the validity of a cached result for multiple requests. Our system can easily incorporate their work for exploiting cached response equivalence for multiple requests.

Cao et. al. [2] use applets to manage cache content at proxies. The web-servers control the caching behavior of the proxies for their content with specific mobile code associated with their content. Updates and invalidations of cached content can be carried out by request-initiated execution of the corresponding applet code.

4 Experiments and Results

4.1 Platform

We analyze and evaluate our caching strategy on the following test-bed:

- **System:** In all our experiments we use three machines, one for the web-server, another for the database server and a third one for the client-browser program that emulates multiple concurrent clients accessing the web-site. We use three identically configured workstations for this purpose. Each of the workstations has an AMD Athlon 800Mhz processor, 256MB SDRAM, and a 30G ATA-66 disk drive. They are connected through 100MBps Ethernet LAN that saw no additional network activity during our experiments. FreeBSD 4.0-STABLE is the operating system.

- **Application Software:** We use Apache v.1.3.12 [1] for our web-server, configured with the PHP v.4.0.2 module [9] providing server-side scripting for generating dynamic content. MySQL v.3.23.22-beta [8] is our database server. PHP has a client API for MySQL through which the web-server interacts with the database server.

- **Test Software:** We use the TPC-W benchmark [12] as the test software. The benchmark has been designed specifically for evaluating the performance of complete systems for e-commerce. The benchmark is implemented as a combination of database tables, web-pages on the server that query the database and generate dynamic content, and a browser-emulator program with multiple request generation modules. There is detailed specification to control the characteristics of each of the components. More detail on TPC-W follows in the next section.

4.2 TPC-W Benchmark implementation

The TPC-W benchmark from the Transaction Processing Council is a transactional web benchmark specifically designed for evaluating e-commerce systems. The workload is designed to evaluate a breadth of system components among which *dynamic page generation with database access and update* is of maximum relevance to our work. The performance metric reported by TPC-W is the number of *web interactions per second* - **WIPS**.

Multiple interactions are used to simulate the activity of a retail store. The store size is chosen from among a set of scale factors which is the number of items in the inventory. TPC-W simulates three different interaction profiles or mix by varying the ratio of *browse* to *buy*: specifically **browsing**, **shopping**, and web-based **ordering**.

Table 1 lists all the 14 different interactions and their proportions in the different profiles. The column, *Time*, in the table refers to the average time taken (in milliseconds) for each interaction on our test-bed. This gives an idea of the relative complexity of the interactions, and combined with the proportions an idea as to the difference in the load for the different profiles. The last column, *Cached*, indicates whether we support caching for that interaction or not. We implement each of the interactions as a separate PHP page, that can take arguments supplied using the HTTP [6] methods GET or POST. Each interaction also involves the request for multiple embedded images. Except for *Order Inquiry*, all other interactions can potentially query the database server.

Table 2 lists the 8 different database tables and their sizes in our test-bed. The sizes include that of the necessary indexes on each of the tables to make the queries in the interactions efficient. In addition to the data in the tables, there is an additional image component - the images are

Web Interaction	**Browsing**	**Shopping**	**Ordering**	**Time (ms)**	Cached ?
Browse	**95%**	**80%**	**50%**		
Home	29.00%	16.00%	9.12%	26	Yes
New Products	11.00%	5.00%	0.46%	48	Yes
Best Sellers	11.00%	5.00%	0.46%	668	Yes
Product Detail	21.00%	17.00%	12.35%	25	Yes
Search Request	12.00%	20.00%	14.53%	22	Yes
Search Results	11.00%	17.00%	13.08%	356	Yes
Order	**5%**	**20%**	**50%**		
Shopping Cart	2.00%	11.60%	13.53%	37	No
Customer Registration	0.82%	3.00%	12.86%	19	Yes
Buy Request	0.75%	2.60%	12.73%	47	No
Buy Confirm	0.69%	10.20%	10.18%	54	No
Order Inquiry	0.30%	0.75%	0.25%	3	No
Order Display	0.25%	0.66%	0.22%	77	No
Admin Request	0.10%	0.10%	0.12%	21	Yes
Admin Confirm	0.09%	0.09%	0.11%	8869	No

Table 1: Web Interaction Mix and Characteristics

Table Name	Number of Rows	Table Size (KB)
Customer	2,880,000	1341139
Address	5,760,000	861567
Orders	2,592,000	269090
Order_Line	7,782,313	988555
CC_XACTS	2,592,000	252800
Item	10000	6368
Author	2500	954
Country	92	8

Table 2: Database Table Characteristics

meant to represent data flow of different media on the web. For our test-bed, the image component comes to an additional 183MB of data, and is resident on the Apache server.

The interactions *Home*, *Best Sellers*, *New Products*, *Search Request*, and *Search Result* have a *Promotional Processing* component - displays 5 promotional items on the response page (extracting their information at random from the database) - which is presumably the advertisement component for the e-commerce environment. The specifications state that this component be executed for every request of that interaction. If enforced it would make all these pages virtually un-cacheable. We relax this constraint, by adding an expiration time for each of these interactions - that would reasonably mimic a real-world situation. Once elapsed, a request for any of these interactions would re-evaluate the Promotional Processing component for that interaction, the response of which would again be cacheable for the duration of an expiration period.

We cache responses for *BestSellers*, *Customer Registration*, *Home*, *New Products*, *Product Detail*, *Search Request*, *Search Result*, and *Admin Request* interactions. Each interaction can have multiple cached responses, each one for a specific set of arguments. The other interactions are either not cacheable because they update the database, or are deemed as not very useful to cache (because of the lack of locality in their requests).

The interactions *Buy Confirm* and *Admin Confirm* update the database in a manner that would invalidate specific cached responses for the *Best Sellers*, *New Products*, *Product Detail*, and *Admin Request* interactions. These invalidations are carried out before the invalidating request is served, as explained in the section (2.2) on DCache implementation.

Our TPC-W implementation differs from the specifications in the following ways:

- We do not implement secure communications layer and authentication system for sales transactions - this affects only a very small percentage of the total number of interactions and would not affect our results for caching.
- MySQL does not, yet, support full-fledged ACID transactional semantics - hence, we programmatically provide the isolation semantics, and not with standard transactional support.
- TPC-W specifies an average *session length time* of 15 minutes, and an average *think time* for every interaction of 7 seconds. A session is a sequence of interactions emulated as those for the same visitor to the site. For each session, the client-browser emulator opens a persistent HTTP connection to the web-server and closes it at the end of the session. The session length time is generated from a random distribution with the specified mean. Once elapsed, a new session is started by the same emulated client. The think time is the time at the end of each interaction that the emulated client should wait, before sending the first request of the next interaction - this simulates its namesake for a real visitor to the site. It is also generated from a random distribution with the specified mean.

  We use a mean session length time of 2 minutes, and a think time of 1 second for all our experiments. This results in a higher load on the servers for the same number of emulated clients than with the specified mean values (approximately 7 times higher) - this allows us to run our heavy-load experiments with a lesser number of emulated clients. More importantly it allows us to have shorter runs that have the same effect for our purposes as longer ones, with the TPC-W specified mean values.

4.3 System characteristics

In this section we present an analysis of the increase in load seen on the servers with increase in the number of concurrent clients. In Table 3 we present the CPU load on our servers as we vary the number of emulated concurrent clients. For all the three interaction profiles we see the database server get more loaded than the web-server as the number of clients increase. For the Browsing and Shopping profiles, the web-server's CPU utilization actually drops from 10 to 100 clients. This is because the database server is so overloaded that it cannot maintain the same level of activity for the web-server as at the 10 client level. In the case of the Ordering profile, such marked degradation is absent. However, there is a much higher increase in the database server's CPU utilization than the web-server's with increasing number of clients. The results clearly demonstrate that the database server is the primary bottleneck in the system. Our caching solution becomes the perfect match for this situation by directly addressing the bottleneck. It does so by using the spare resources on the web-server to offload computation from the database server.

4.4 DCache vs No-Cache

In this section, we present the comparison between our caching implementation and the system without caching. We use the total number of completed interactions in the same interval in time as

	Browsing			Shopping			Ordering		
	1	10	100	1	10	100	1	10	100
Apache	3.1%	13.3%	10.8%	3.1%	21.4%	15.4%	3.2%	24.2%	33.2%
MySQL	9.6%	92.6%	98.3%	7.1%	69.6%	98.2%	4.8%	41.4%	87.1%

Table 3: Percentage CPU Utilization on Servers with Increase in Clients

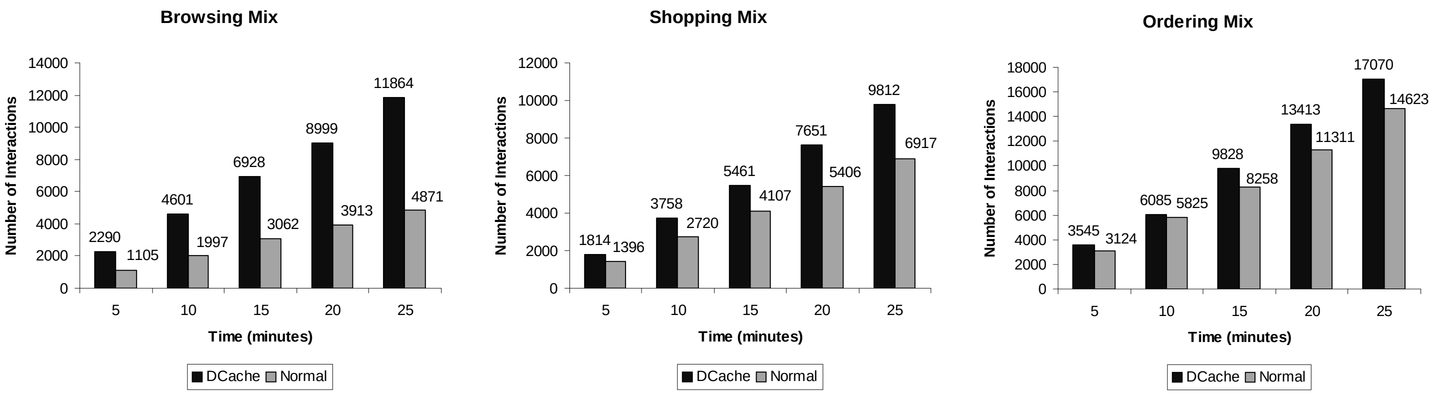

Figure 1: Number of Interactions vs Time with and without Caching

the measure of comparison between the two. We do the comparison with 100 emulated concurrent clients - a situation where the database server becomes the bottleneck for the non-caching/normal implementation. This is to show the effect of caching exactly at a point when a solution to the performance bottleneck becomes necessary. Also the expiration time used for cached pages is 2 minutes - the value for the mean session length time, i.e., during a single session an emulated visitor is likely to see the result of just one promotional processing for a specific interaction. We believe this to be a conservatively low value for the expiration period, and thus, under less constrained situations we would get bigger gains for caching.

Figure 1 summarizes the results for the three different profiles. Our cache implementation is labeled *DCache*, and the regular one as *Normal*. It is immediately apparent that caching is beneficial for all profiles. But, the gains vary significantly due to the different mix of interactions for each. The number of interactions for caching range from 207%, after 5 minutes, to 244%, after 15 minutes, of that of the no-cache situation for Browsing. For Shopping the range is from 130% to 142%, and for Ordering the range is from 113% to 117%. More importantly, for all the three profiles the improvement increases steadily with time. Browsing, Shopping and Ordering exhibit an order of decreasing benefit from caching - this is directly due the the decreasing order of the proportions of costly, but cacheable, interactions in them.

We performed other measurements at lightly loaded conditions which indicate that our caching implementation has no noticeable overheads at such conditions, while providing similar but smaller gains.

In our test-bed, we do not emulate the caching of images at the clients. This potentially increases the load on the Apache Server (images are resident on the Apache server). However, since the bottleneck happens to be the database server, the effect of this on our caching results is not significant.

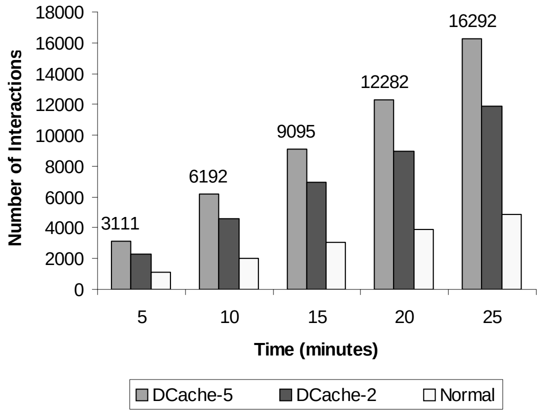

Figure 2: Number of Interactions vs Time for different expiration times

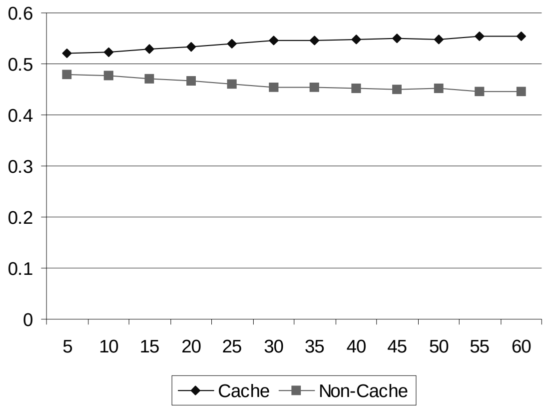

Figure 3: Change in the proportion of cached and non-cached responses with time

4.5 Experiments analyzing DCache Characteristics

In this section, we analyze the behavior of our caching solution in greater detail. Since the caching behavior is most prominent with the Browsing profile, the results presented in this section are for that profile. In Figure 2, we present the numbers for an expiration time of 5 minutes, along with the already presented ones for 2 minutes and the no-cache situation. There is considerable increase in performance for relatively small increase in expiration time - this is related to the load on the system, which in this situation is again kept at 100 emulated concurrent clients. With the expiration time at 5 minutes the number of interactions completed is around 136% of those at 2 minutes. This corresponds to a factor *of over 3 times* the performance with the non-caching implementation.

In the previous section, we noticed a growth in the improvements from caching with time. Here, we present the results of a run in a different manner that should make the reason for this improvement clearer. Figure 3 has the the ratio of the interactions served from cache and the others to the total number of interactions on the y-axis. On the x-axis is the time in minutes. The fraction of interactions served from the cache increases with time, and correspondingly the number not served from cache decreases. Since, the ones served from cache obviously take less time/and load on the servers, as their fraction grows with time, so does the improvement from caching.

5 Conclusions

We summarize our unique contributions as follows:

- We present the first quantitative study of the benefits of dynamic content caching with a specific workload, TPC-W. Our study show that we can obtain over 3 times the normal performance under heavy load with caching.
- We provide a simple and easily adoptable mechanism for obtaining caching for dynamic content. Our DCache module is a minimal extension to popular existing infrastructure for delivering dynamic content.
- Our *request-initiated* invalidation approach is an effective method for detecting cache content invalidations for a large class of web-sites. Any site that has a majority of its updates

submitted through its web-server, can effectively adopt this approach to deliver consistent data, while benefiting from caching.

- We propose a flexible, site-specific optimization capability for caching - By allowing the site-administrators/programmers to register the hashing function for mapping the cached response to a cache slot, we incorporate their priorities for the site's pages into our caching scheme.

We plan to extend our work with an exhaustive analysis of the limits of our current implementation for caching. We also plan to study other kinds of dynamic content workloads in this context. The next in dynamic content caching, we believe, is to extend the caching approach to multiple levels. While we currently cache complete responses, it is easy to envision scenarios where caching database query results, specific fragments of responses, or compiled script-caching would be valuable. A multi-level caching scheme that can choose the optimal level of caching for each response is our next goal.

References

[1] The Apache Software Foundation. http://www.apache.org/.

[2] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *In Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, 1998.

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

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

[5] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A Publishing System for Efficiently Creating Dynamic Web Data. In *Usenix Symposium on Internetworking Technologies and Systems*, March 2000.

[6] Hypertext Transfer Protocol - HTTP/1.1. http://www.w3c.org/Protocols/rfc2616/rfc2616.html.

[7] Evangelos P. Markatos. On Caching Search Engine Results. In *Technical Report 241, ICS-FORTH*, January 1999.

[8] MySQL. http://www.mysql.com.

[9] PHP Hypertext Preprocessor. http://www.php.net.

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

[11] Lincoln Stein and Doug MacEachern. *Writing Apache Modules with Perl and C.* O'Reilly, 1999.

[12] Transaction Processing Council. http://www.tpc.org/.

[13] Craig E. Wills and Mikhail Mikhailov. Examining the Cacheability of User-Requested Web Resources. In *Proceedings of the 4th International Web Caching Workshop*, April 1999.

[14] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-Based Analysis of Web-Object Sharing and Caching. In *Usenix Symposium on Internetworking Technologies and Systems*, 1999.