Using Code Transformation for Consistent and Transparent Caching of Dynamic Web Content

Sara Bouchenak, Sumit Mittal, Willy Zwaenepoel

EPFL Technical Report ID: 200383,
Swiss Federal Institute of Technology (EPFL), Switzerland, 11th December 2003

Using Code Transformation for Consistent and Transparent Caching of Dynamic Web Content

Sara Bouchenak*
Sara.Bouchenak@epfl.ch

Sumit Mittal†
mittal@cs.rice.edu

Willy Zwaenepoel*
Willy.Zwaenepoel@epfl.ch

*EPFL, Switzerland
†Rice University, USA

Abstract

We present a consistent and transparent caching system for dynamic web pages produced by a server-side application using a back-end database. Cached pages always reflect current database values. No intervention from the programmer is necessary to implement caching. The system is an improvement on earlier methods that either did not guarantee consistency and/or relied on substantial programmer intervention.

The novel idea is that a compiler analyzes and transforms the server-side application code to include cache checks, inserts, and invalidations. In order to provide precise invalidations and attendant good hit ratios, we check the intersection of the database table columns used by the read and the write queries, augmented by uniqueness information from the database schema and comparison of the query selection predicates against values inserted in the database.

We use Java bytecode rewriting to implement the transformation of the server-side application. Using the Rubis benchmark, we demonstrate that transparent and consistent caching achieves substantial improvements in response time and throughput.

1 INTRODUCTION

An increasing percentage of web content is dynamically generated. Web servers for dynamic content are usually implemented using a three-tier architecture, using an HTTP server as a web front-end and provider of static content, an application server to execute the business logic of the application, and a database to store the dynamic content. Dynamic content generation places a significant burden on the servers, often leading to performance bottlenecks. As a result, various techniques have been studied for server-side acceleration of dynamic-content web sites, including replication and clustering of the three tiers, and caching of content at various levels. The use of these techniques is rendered more complicated by the dynamic nature of these services, requiring mechanisms to maintain consistency between various cached or replicated copies of the data.

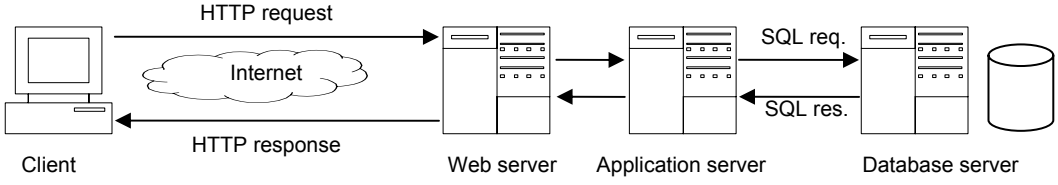

Figure 1. Architecture of dynamic web applications

This paper introduces a new method for server-side caching of dynamically generated web pages. The method provides both consistency and transparency with good performance. Strong consistency is achieved, i.e., the cached pages reflect current values in the database. The method is transparent in the sense that no effort is required from the programmer to achieve consistent caching. The key novel idea is the use of static analysis and transformation of the server-side application code. In other words, a compiler analyzes and transforms the code to perform cache checks, inserts and invalidations.

The new method compares favorably to various state-of-the-art dynamic caching approaches. Non-transparent approaches such as [6] [15] [10] [15] require extensive programmer intervention to indicate how cached data needs to be invalidated. A transparent approach such as [2] only provides time-lagged consistency. In contrast, the method described in this paper provides both transparency and consistency.

We have implemented our approach in an environment in which the server-side application is written as Java servlets using embedded SQL statements. Analysis and transformation is done at the level of Java bytecodes. Although our current implementation was carried out in this particular environment, the methods are applicable to many common environments used for dynamic content generation. The success of the methods depends on the accuracy of static analysis that can be achieved.

We have evaluated this implementation using standard software components, including Apache, Tomcat and MySQL, all running on current server-class PC hardware. We use the Rubis auction site as our application, which has been used for various studies of dynamic content sites. We compiled this application using the methods described in this paper. In addition, we have developed a hand-coded caching version of the application, which takes advantage of application-specific knowledge. This approach is included to provide a reasonable upper bound for the results that can be achieved using web page caching for this application.

Measurements of this implementation show that, in addition to consistency and transparency, our methods provide good performance. Average response time is reduced by up to 49% relative to no caching, and up to 62% with hand-coded caching. Under high load, throughput is improved by 38% relative to no caching, and is only 26% below that achieved by hand-coded caching. For our particular combination of hardware, software and application, using caching removes the bottleneck from the database server, explaining the performance improvement resulting from caching under high load. The differences in performance between the hand-coded and the automated caching versions are due to a cache-unaware coding style in the application and to imprecisions in our automated invalidation algorithm.

The rest of this paper is organized as follows. Section 2 describes how an application is transformed to perform caching. Section 3 describes the Rubis application, both the original version that was used as input to the compiler and the version in which caching was added by hand. Section 4 describes the experimental environment. Section 5 describes the performance results, including a discussion of the factors that affect performance differences between the various versions. Section 6 describes related work. Section 7 discusses some avenues for further work and draws conclusions.

2 METHODS

The cache consists of a set of web pages (or web page fragments). These web pages result from the execution of read-only web request handlers, i.e., request handlers in the server-side application that generate only read requests to the database. The goal is to avoid re-execution of read-only request handlers with the same set of arguments, and return the results from the cache instead. The results, if any, of request handlers that involve writes to the database are not cached, because they need to be re-executed on each subsequent invocation.

2.1 Cache structure

The key used for accessing the cache is the URI of the request with all its associated input information (arguments and cookies). Associated with each cache entry is *dependency information*.

Roughly speaking, this dependency information records which database queries were executed to obtain the cached web page.

2.2 Read-only request handler transformation

The compiler transforms the code of read-only request handlers to check for and to insert entries in the cache. In particular, for each read-only request handler it inserts a pre-processing step that uses the arguments of the current invocation (including cookies, if any) to check whether a valid entry exists in the cache for this request and this set of arguments. If so, the cached page is returned immediately as a response to the request, without any further execution. If not, the request handler is executed. The compiler inserts code to collect dependency information after each database query. At the end of the request handler, the compiler includes a post-processing step that inserts the page just produced in the cache, accompanied by the dependency information, collected during the execution of the request handler. In summary, each read-only request handler is transformed as seen in Figure 2.

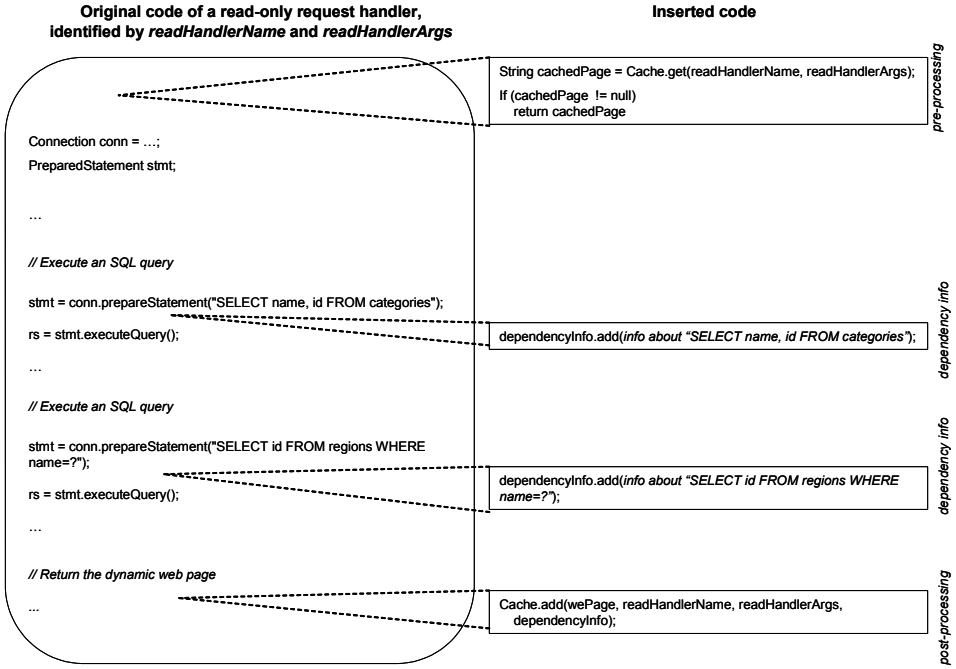

Figure 2. Read-only request handler transformation

2.3 Write request handler transformation

The compiler applies a similar transformation to write request handlers. After each write query (update, insert or delete), the compiler inserts code to collect *invalidation information* associated with this query. A post-processing step is added at the end of the write handler to invalidate cache entries, based on the invalidation information produced by the individual write queries (see Figure 3).

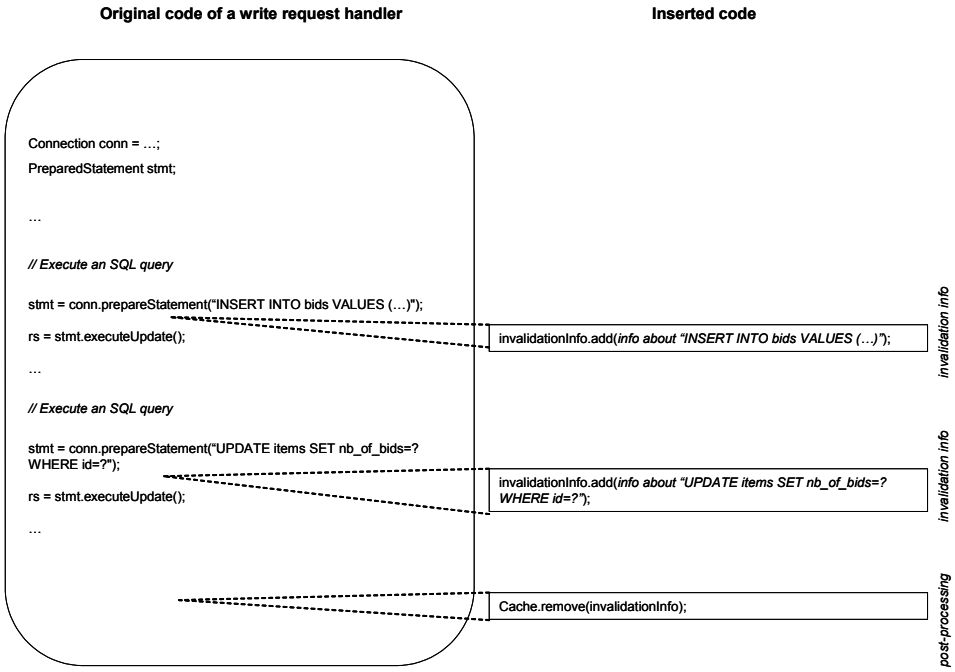

Figure 3. Write request handler transformation

2.4 Benefits of code rewriting

This code rewriting strategy allows us to obtain, without programmer intervention, a precise association between, on one hand, a request handler invocation (and its parameters) and the cache entry it produced, and, on the other hand, the queries executed during that invocation. This is a major advantage that allows us to build a consistent and transparent caching solution.

2.5 Computing invalidations

Ideally, we want to precisely determine whether the writeset (i.e., the set of database data items written) of a write request handler invocation intersects with the readset (i.e., the set of database data items read) of a read-only request handler invocation. If so, we need to invalidate the cache entry resulting from that read-only request handler invocation. On one hand, the determination of this intersection need not be entirely precise, as long as we conservatively invalidate each cache entry for which we cannot determine that the intersection is empty. If we conservatively invalidate an entry for which the intersection is empty, the execution remains correct, albeit perhaps less efficient, because what could have been a cache hit is turned into a cache miss. On the other hand, collecting all the information to precisely determine whether the intersection is empty may be prohibitively expensive. For instance, in some cases it may involve database accesses. In short, there is a tradeoff between the cost in time and space of collecting and storing dependency and invalidation information and the cache hit ratio. We recognize the following four distinct positions along this tradeoff: 1) using information about what (columns of) tables are accessed, 2) using database schema information, 3) using the query selection predicates, and 4) using information in the database.

In its simplest form, the dependency and the invalidation information simply consist of the tables read or written in the request handlers. A more selective approach is to record the columns read and written, and check the intersection of those two sets. Collecting this information requires minimal run-time processing.

The above approach can be refined by using uniqueness information from the database schema. The schema specifies which columns must have unique values. A special case of such a column is the primary key. It is frequently the case that read queries select a single row of a table based on a unique column (most commonly based on the primary key). Even more frequently, write queries operate on a single row by selecting it based on the primary key. Clearly, read and write queries in which the selection is made based on an equality test of a unique column with a constant can only intersect if that

constant is the same. By recording that constant with the dependency and the invalidation information, and doing a simple equality test, we can determine that queries do not intersect, even if they access the same columns. Since such queries (especially such update queries) are common, this refinement improves cache hit rate at a very modest runtime cost.

Using the selection predicate in the queries, further unnecessary invalidations can be avoided. For instance, if a read query contains a selection predicate based on an inequality involving one of the attributes in the table, and a write query contains a selection predicate based on an equality of that same attribute with a constant, emptiness of the intersection can be checked by recording the interval boundaries and the constant, and performing the comparison. Since both the write query and the read queries may involve complex predicates, this check can become expensive.

Our current implementation maintains a record of the columns read and written, refined by relying on uniqueness information from the database schema. We also plan to check the selection predicates of the read-only queries against insertion/update queries.

Finally, the cache could obtain extra information from the database to further reduce the number of invalidations. An example occurs when the read query is a join between two tables and the write query performs an insert into one of the tables. Without checking the other table for a row matching the value of the join attribute, we have to conservatively assume that the resulting page needs to be invalidated. We have for now rejected the idea of doing additional queries to the database to check the validity of cache entries as this seems to undo the main purpose of keeping a cache in the first place, although it may be useful if very complicated queries can be avoided in favor of simple ones.

2.6 Implementation Issues

2.6.1 Cache structure

Our cache structure is mainly composed of two data structures: a *cache entries table* and an *invalidation entries table*, as respectively described by Table 1 and Table 2. The *cache entries table* represents the cache itself, where the dynamically generated web pages are kept in memory for subsequent use. Our *Cache.get* method introduced in Figure 2 looks for an entry in this table, given a read-only request handler name and a set of arguments. If such an entry exists, the associated cached web page is returned as a result. When a new entry is added to the *cache entries table*, using the *Cache.add* method presented in Figure 2, the built dependency information is stored with this entry. This information mainly consists of: a set of table columns on which the read-only request handler depends, a flag that denotes that the column uniqueness was used in the read-only request handler, and the associated unique value.

URI (readHandlerName + readHandlerArgs)	**Cached web page**	**Dependency information**		
		isUnique	**{(unique column, unique value), …}**	**{non-unique column, …}**
Name1 + Args1	WebPage1	no	–	{column1, column2}
Name2 + Args2	WebPage2	yes	{(column1, value1)}	{column3}
…	…	…	…	…

Table 1. Cache entries table

The *invalidation entries table* is used for maintaining cache consistency. Indeed, when the *Cache.remove* method is requested (see Figure 3), given invalidation information, i.e., table columns, column and value uniqueness information, the *invalidation entries table* is used to find the cache entries that are associated with the given invalidation information. These entries are therefore invalidated, if the uniqueness optimization does not apply.

Column name	{readHandlerName + readHandlerArgs, …}
column1	{ Name1 + Args1, Name2 + Args2, …}
column2	{ Name1 + Args1, …}
column3	{ Name2 + Args2, …}
…	…

Table 2. Invalidation entries table

2.6.2 Application code transformation

In our implementation, the server-side application code uses Java servlets with embedded SQL statements. Thus, the request handlers discussed above are servlets [11]. The compiler first statically analyzes the application code in order to retrieve the SQL requests that are, directly or indirectly (through sub-sequent method calls), executed by the application's servlets. These SQL requests are recognized as being parameters of the *java.sql.Statement.executeQuery*, *java.sql.Statement.executeUpdate* or *java.sql.connection.prepareStatement* methods [12].

The compiler then rewrites the application code in order to add pre-processing, post-processing and per-query dependency and invalidation information collection, as depicted by Figure 2 and Figure 3. The code is also transformed in order to buffer the output strings (the dynamically generated web page) produced during read-only servlet execution such that they are available for insertion in the cache at the post-processing step.

We based the implementation of our static code analyzer and rewriter on the BCEL library (Byte Code Engineering Library) [3]. This library allows us to manipulate the application bytecode, thus providing us with an elegant way to transparently add consistent caching to web applications, only using the application jar file or binary Java code and not requiring the application source code.

3 APPLICATION

3.1 RUBiS benchmark

The RUBiS benchmark implements the core functionality of an auction site: selling, browsing and bidding [1]. It does not implement complementary services like instant messaging or newsgroups. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during visitor sessions, during a buyer session users can bid on items and consult a summary of their current bids, their rating and the comments left by other users. Seller sessions require a fee before a user is allowed to put up an item for sale. An auction starts immediately and lasts typically for no more than a week. The seller can specify a reserve (minimum) price for an item.

The database contains nine tables: *users*, *items*, *old_items*, *bids*, *buy_now*, *comments*, *categories*, *regions* and *ids*. The *users* table records contain the user's name, nickname, password, region, rating and balance. Besides the category and the seller's nickname, the *items* and *old_items* tables contain the name that briefly describes the item and a more extensive description, normally an HTML file. Every bid is stored in the *bids* table, which includes the seller, the bid, and a max_bid value used by the proxy bidder (a tool that bids automatically on behalf of a user). Items that are directly bought without any auction are stored in the *buy_now* table. The *comments* table records comments from one user about another. As an optimization, the number of bids and the amount of the current maximum bid are stored with each item to prevent many expensive lookups on the *bids* table. This redundant information is necessary to keep an acceptable response time for browsing requests. As users browse and bid only on items that are currently for sale, we split the *items* table in separate *items* and *old_items* tables. The vast majority of requests access the new items table, thus considerably reducing the database working set.

Rubis defines 26 interactions that can be accessed from the client's Web browser. Among the most important ones are browsing items by category or region, bidding, buying or selling items, leaving comments on other users and consulting one's own user page (known as *My eBay* on eBay [4]). Browsing items also includes consulting the bid history and the seller's information. We define two workload mixes: a browsing mix made up of read-only interactions and a bidding mix that includes 15% read-write interactions. The bidding mix is the most representative of an auction site workload.

The system is sized according to some observations found on the eBay Web site. We always have about 33,000 items for sale, distributed among eBay's 40 categories and 62 regions. We keep a history of 500,000 auctions in the *old_items* table. There is an average of 10 bids per item, or 330,000 entries in the *bids* table. The *buy_now* table is small, because less than 10% of the items are sold without any auction. The *users* table has 1 million entries. We assume that users give feedback (comments) for 95% of the transactions. The comments table contains about 500,000 comments. The total size of the database, including indices, is 1.4GB.

Rubis has been used in other dynamic content web site experiments. Its code is freely available from *www.objectweb.org*.

3.2 Hand-coded caching for RUBiS

We wrote a version of Rubis that uses caching. This hand-coded caching version relies on knowledge of the application, and is of course not transparent. For instance, the *SearchItemsbyCategory* read-only servlet builds a web page with the list of items that belong to a given category argument. And the *RegisterItem* write servlet adds a new item to the table of items. With the hand-code caching, the pages resulting from the *SearchItemsbyCategory* read-only servlet are invalidated if later a *RegisterItem* occurs with the same category argument. They are left valid if the argument of *RegisterItem* is different.

The cache structure of this hand-coded caching system is similar to the one for the automated cache. A cached web page key for lookup is the same as before, i.e., a URI (servlet name and arguments). The dependency information is simplified, and just contains a few servlet-specific values.

The code for the servlets is (by hand) transformed in a way similar to the way we automatically transform the code. In particular, read-only servlets have a pre-processing step, in which they check for a valid cache entry, and a post-processing step, in which they insert a page and its dependency information in the cache. A write servlet has a post-processing step, in which it potentially invalidates cache entries. Unlike the automated version, there is no per-query code added to either read-only or write servlets.

4 EVALUATION ENVIRONMENT

4.1 Client emulation

The Rubis benchmark comes with a client-browser emulator. A client session is a sequence of interactions for the same client. For each client session, the client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability to go from one interaction to another.

The think time and session time for both benchmarks are generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes, respectively. These numbers conform to clauses 5.3.1.1 and 6.2.1.2 of the TPC-W v1.65 specification [15]. We vary the load on the site by varying the number of clients.

All experiments start with a cache warmed up for 1 hour. The measurements are collected during the following 30-minute period.

4.2 Software

We use Apache v.1.3.22 as the Web server. We increase the maximum number of Apache processes to 512. With that value, the number of Apache processes is never a limit on performance. The servlet engine is Jakarta Tomcat v3.2.4, with the MM-MySQL v2.04 type 4 JDBC driver [8], running on Sun JDK 1.4.2. We use MySQL v.3.23.43-max [9] as our database server with the MyISAM tables. All machines run the 2.4.12 Linux kernel.

4.3 Hardware

Each machine has an Intel Xeon 2.4GHz CPU with 1GB ECC SDRAM, and a 120GB 7200pm disk drive. The database server runs on a separate machine because it is the bottleneck for the version without caching. The web server and the application server run on the same machine. This never causes a bottleneck on that machine. The client emulator runs on a separate machine. We have verified that in none of the experiments the client emulator is the bottleneck. All machines are connected through a switched 1GMbps Ethernet LAN.

5 EXPERIMENTAL RESULTS

We present results for the application without caching, with automated caching and with hand-coded caching. For the browsing mix there is almost no difference between the automated caching and the hand-coded caching versions, and the two perform much better than the version without caching. The throughput of the two caching versions continues to grow with increasing numbers of clients (see Figure 4). At 1500 clients, the largest client population measured, throughput is 17200 requests per minute.

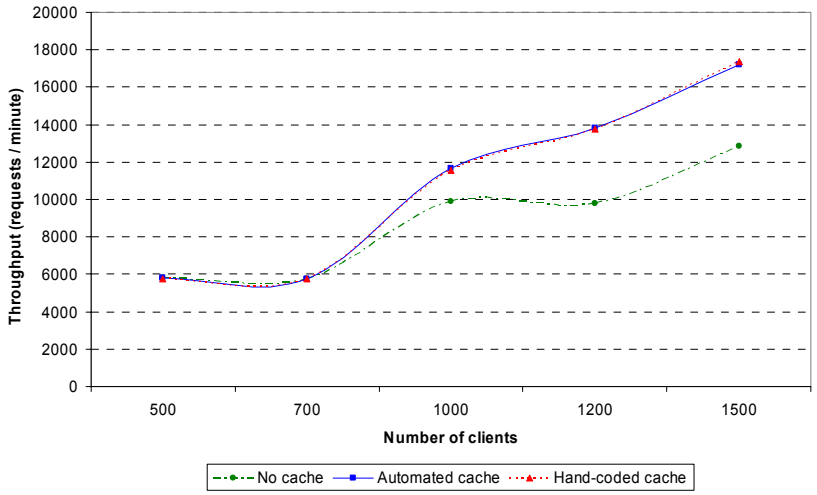

Figure 4. HTTP request throughput for RUBiS – Browsing mix

Similarly, average response time for the two caching versions remains low, from a low of 8 milliseconds at 500 clients up to a maximum of 31 milliseconds at 1500 clients (see Figure 5). In contrast, without caching, response time grows from 20 milliseconds at 500 clients to 1.8 seconds at 1500 clients.

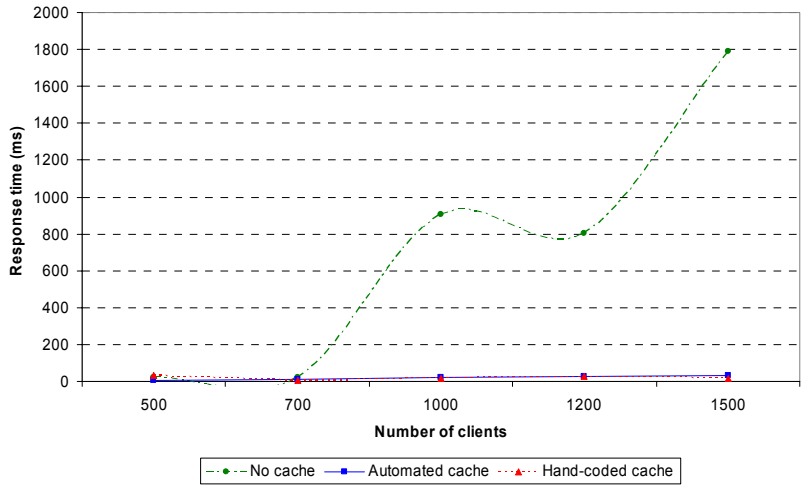

Figure 5. HTTP request response time for RUBiS – Browsing mix

These results can be first explained by comparing the hit rates in the automated cache and the hand-coded cache, which are almost the same as shown by Figure 6. The results can also be explained by the CPU utilization on the application server and on the database machine. Without caching, the CPU utilization on the database server reaches 99%. This bottleneck at the database explains the flat throughput curve and the rapid growth in response time. In contrast, with caching, the CPU utilization on the database server remains low, while it grows on the application server, with increasing number of clients. For hand-coded caching the maximum CPU utilization (80%) is a little lower than for automated caching (90%), reflecting the higher overhead of automated caching.

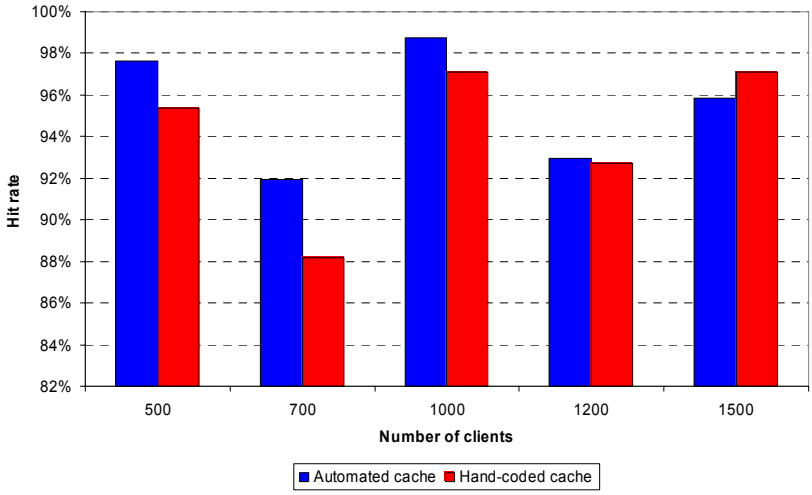

Figure 6. Cache hit rate for RUBiS – Browsing mix

The results for the bidding mix confirm the benefits of caching when the load increases, but, unlike for the browsing mix, the hand-coded caching performs better than the automated caching. At 1500 clients, the hand-coded caching version achieves 17340 requests per minute, the automated caching 14580, and the version without caching 10560.

Figure 7. HTTP request throughput for RUBiS – Bidding mix

The response time for both caching systems is always better than the one obtained without caching (see Figure 8). For instance at 1500 clients, automated caching achieves an average response time of 1.2 seconds, hand-coded caching 0.9 second, and no caching 2.3 seconds. The difference of performance between the automated caching and the hand-coded caching is due to the fact that the latter performs less cache invalidations on writes than the former, thanks to its knowledge of the application semantics. These different invalidation strategies also explain the difference in cache hit rates as shown by Figure 9.

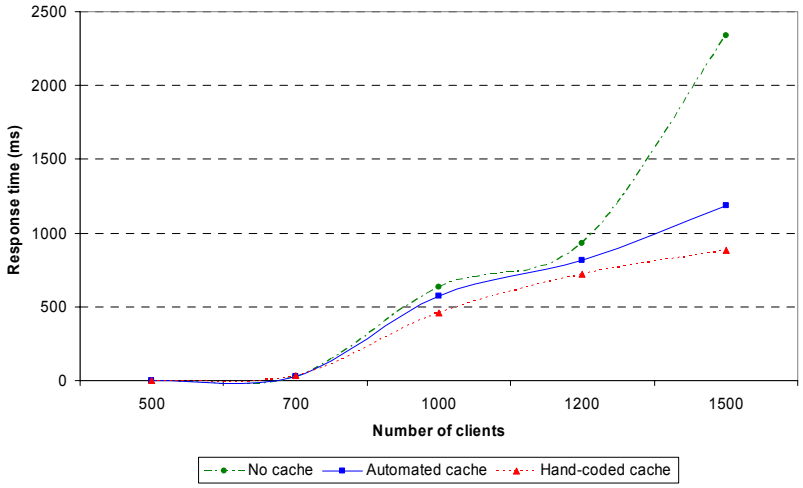

Figure 8. HTTP request response time for RUBiS – Bidding mix

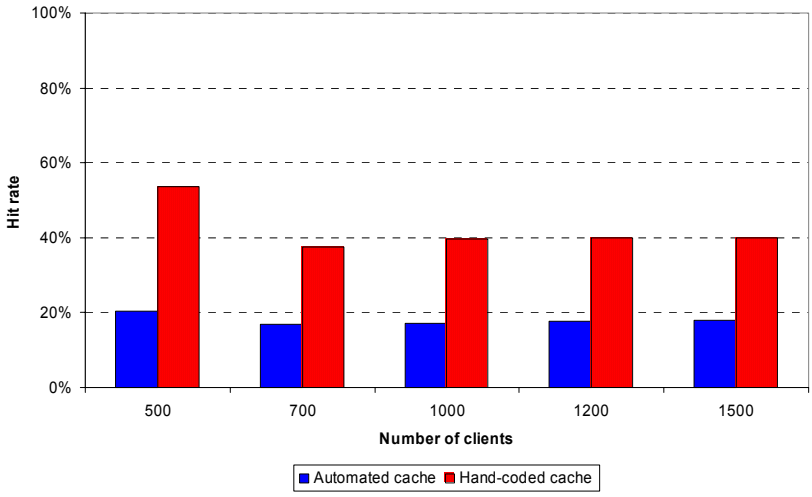

Figure 9. Cache hit rate for RUBiS – Bidding mix

6 RELATED WORK

Much work has been done on the caching of Web content. Most of it focuses on static content. Two types of caching have been applied to caching of dynamic Web content. In *database caching*, the results of previous queries to the databases are cached, either as query results or as materialized views [13] [7] [15]. Alternatively, in *dynamic web page caching* whole pages or page fragments (either in HTML or in XML) are cached. The two approaches are complementary and can be combined. This paper focuses solely on dynamic web page caching. Our work distinguishes itself from other work on dynamic web page caching in that we provide *both* consistent and transparent caching. Previous work in this area has only provided one of these two qualities. Among these works, we single out as representative examples CachePortal [2] (transparent but not consistent) and DynamicWeb [6] (consistent but not transparent).

Transparency in CachePortal is achieved by logging the HTTP requests to and responses from the Web server, the queries made to the database, and the updates made to the database, each of which with a (wall clock) timestamp [2]. CachePortal faces the same problem as our work: how to associate queries with requests in a transparent manner. Their approach to this problem is completely different. They conservatively assume that all queries in the database log with timestamps between the time of an HTTP request and its reply are associated with that request. In contrast, by transparently rewriting the application code, we get a precise association between requests and queries. In addition, they provide time-lagged consistency. Periodically, they read the database update log and perform invalidations according to the association between HTTP requests and queries. They evaluate their system only under low load with a synthetic application and a fixed hit rate. Under high load, the precision of our request-query association with the attendant reduction in number of invalidations should lead to higher rates and superior results.

DynamicWeb provides an API for specifying a dependency graph between certain events, in particular between write queries and cached web pages [6]. The occurrence of said events triggers invalidations. Invalidations are instantaneous and Dynamic WebCache therefore provides consistency, but at the expense of considerable effort from the application programmer. In our experience with the hand-coded caching system, precisely locating all dependencies even in the relatively simple Rubis benchmark (25 servlets and 4,600 lines of code) proved to be a major challenge.

Other non-transparent approaches include Weave [16], in which the programmer is required to use a specialized language to describe dynamic web pages and event handlers to specify invalidations, and various commercial solutions such as e.g., SpiderCache [10] and XCache [15], both of which provide an event API. In addition, these systems typically support periodic updates and therefore only time-lagged consistency.

Our work is orthogonal to issues of caching granularity. Partial page caching can be implemented using ESI or similar techniques [5]. Our analysis method can be extended to insert, check and invalidate a cache containing page fragments defined by ESI or similar annotations.

7 CONCLUSIONS AND FUTURE WORK

We have demonstrated consistent and transparent caching of dynamic web pages by transformation on the server-side application code. A compiler transforms the code to perform cache checks, insertions, and invalidations. In order to perform precise invalidations, we record the relevant aspects of the selection predicates in the queries used to produce a dynamic web page. Inserts are then checked against these selection predicates to determine if a cached web page needs to be invalidated.

We have built a prototype system that implements these methods for server-side applications written as Java servlets. We have demonstrated hit rates and attendant improvements in response time and throughput approaching those of a hand-coded caching version of the application, that took considerable effort to develop. We have also illustrated the reasons for the remaining gap between the hand-coded and the automated version.

One way to potentially close the gap is to use both a dynamic web page and a database query result cache. Although systems exist that incorporate both (e.g., Weave), none of them integrate the two caches. The database query result cache could be used to produce more precise automatic invalidations. If looking at the query predicates, as proposed in this paper, is not sufficient to assert the validity of a cache entry, the system could look into the query result to further eliminate unnecessary invalidations.

Our approach is completely transparent when all database updates go through the server-side application. If this is not the case, then transparency is difficult to achieve. Our approach can easily be extended with an API similar to the ones provided by Dynamic WebCache and Weave to allow an external entity to invalidate cache entries. This external entity could, for instance, work through database triggers. Given the high cost of trigger mechanisms, our approach would still offer an important performance improvement, if the majority of the database updates go through the application.

8 REFERENCES

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, USA, Nov. 2002.
http://rubis.objectweb.org

[2] K. S. Candan, W. S. Li, Q. Luo, W. P. Hsiung, D. Agrawal. Enabling. Enabling Dynamic Content Caching for Database-driven Web Sites. *ACM SIGMOD'2001*, Santa Barbara, CA, USA, 2001.

[3] M. Dahm. Byte Code Engineering. *Java-Information Tage (JIT'99)*, Düsseldorf, Germany, Sep. 1999.

[4] eBay Inc. http://www.ebay.com/

[5] Edge Side Includes. http://www.esi.org/

[6] A. Iyengar, J. Challenger. Improving Web Server Performance by caching Dynamic Data. *USENIX Symposium on Internet Technologies and Systems (USITS'97)*, Monterey, CA, USA, Dec. 1997.

[7] Q. Luo, J. F. Naughton. Form-Based Proxy Caching for Database-Backend Web Sites. *27th Very Large Data Bases Conference (VLDB'2001)*, Roma, Italy, 2001.

[8] MM-Mysql. MM MySQL JDBC Drivers.
http://mmmysql.sourceforge.net/

[9] MySQL. MySQL Open Source Database.
http://www.mysql.com/

[10] Spider Software. *SpiderCache Enterprise 2.0:Dynamic Content Delivered Faster*. Spider Software Technical White Paper, September 2001.
http://www.spidercache.com/

[11] Sun Microsystems. *Java Servlet Technology*.
http://java.sun.com/products/servlet/

[12] Sun Microsystems. *Java 2 Platform Standard Edition – API Specification*. http://java.sun.com/j2se/1.4.2/docs/api/

[13] TimesTen. *TimesTen Real-Time Event Processing System*. TimesTen White Paper, 2003. http://www.timesten.com

[14] Transation Processing Performance Council. *TPC-W: a transactional web e-Commerce benchmark*. http://www.tpc.org/tpcw/

[15] XCahe Technologies. XCache Overview. http://www.xcache.com

[16] K. Yagoub, D. Florescu, V. Issarny, P. Valduriez. Caching Strategies for Data-Intensive Web Sites. *26th Very Large Databases Conference (VLDB'2000)*, Cairo, Egypt, 2000.