NOTE TO USERS

This reproduction is the best copy available.

UMI®
RICE UNIVERSITY

A Consistent and Transparent Solution for Caching Dynamic Web Content

by

Sumit Mittal

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Alan L. Cox, Chair
Associate Professor of Computer Science
Rice University

Dr. Willy Zwaenepoel
Professor and Dean
Faculty of Computer & Communications Sciences
Swiss Federal Institute of Technology

Dr. Scott Rixner
Assistant Professor of Computer Science
Rice University

Dr. T. S. Eugene Ng
Assistant Professor of Computer Science
Rice University

HOUSTON, TEXAS
JUNE, 2004
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 1425852
Copyright 2005 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346
A CONSISTENT AND TRANSPARENT SOLUTION FOR CACHING DYNAMIC WEB CONTENT

Sumit Mittal

Thesis: Master of Science
Computer Science
Rice University, Houston, Texas (June 2004)
A Consistent and Transparent Solution for Caching Dynamic Web Content

Sumit Mittal

Abstract

Caching is an effective means for reducing load on web servers, especially for those that dynamically generate documents in dynamic web applications. While adding caching to a web application can greatly reduce response times for requests, the logic to ensure consistency with the backend database requires considerable effort to develop. Much of the complexity is in minimizing unnecessary page invalidations, a key goal for improving the cache hit rate and response times. In this thesis I explore a range of invalidation policies that are progressively more precise. A policy is more precise than the other if it produces less false positives (removal of valid pages). A contribution of this work is in achieving precise invalidations at the application server layer automatically.

To explore these issues, I introduce AutoWebCache, a system for adding server-side caching for dynamic content automatically to web applications having a back-end database. To achieve automation, it uses aspect-oriented programming for injecting the cache code into the application. Dependencies between the read and write requests are determined automatically, during run-time. Formulating the dependencies requires SQL query analysis be performed at run-time, which is costly. I demonstrate how to reduce this dynamic analysis overhead through effective caching of intermediate analysis results. In two e-commerce benchmarks, RUBiS and TPC-W, I show my method can be highly effective, reducing the response times, 63% and 97%, respectively.
Acknowledgments

I am grateful to my advisors Dr. Alan L. Cox and Dr. Willy Zwaenepoel for their guidance, constant encouragement and motivation. Their interest and confidence in the work was always a source of inspiration. Many thanks to my other committee members: Dr. Scott Rixner and Dr. Eugene Ng, for their valuable feedback and suggestions.

I also thank Dr. Sara Bouchenak and Dr. Steven Dropsho who have worked with me on this project from across the Atlantic. Their insight and contributions have played a significant part in this project. Special thanks goes to all my friends in the Systems group at Rice: Ajay, Amit, Animesh, Anupam, Anwis, Atul, Johnny, Khaled, and Santa. All have contributed to my thesis in unique ways. Aravind, while working from EPFL, was always a help with his knowledge of the benchmarks and was gracious in the sharing of machines we used for simulations.

Lastly, my family members deserve special credit for keeping me inspired and motivated. I dedicate this work to their support.
Contents

Abstract ........................................... ii
Acknowledgments ................................... iii
List of Illustrations ............................... vi
List of Tables ...................................... vii

1 Introduction ...................................... 1

2 Background ....................................... 5
   2.1 Dynamic Web Applications .................... 5
   2.2 Structure of Database Queries .................. 6
   2.3 Aspect-oriented Programming and AspectJ .... 9
       2.3.1 Introduction to AspectJ .................. 10

3 Design of the AutoWebCache ................... 12
   3.1 Overview ..................................... 13
   3.2 Consistency Model ............................. 15
   3.3 Cache Structure ................................ 16
   3.4 Invalidation Strategies ....................... 21
   3.5 Query Analysis Engine ....................... 23
       3.5.1 Parsing ................................... 23
       3.5.2 Intersection Testing ..................... 24
       3.5.3 Analysis Caching ......................... 25
       3.5.4 Generation of Extra Queries ............. 25
   3.6 Code Injection with AspectJ ................... 26
3.7 Limitations of Automatic Caching ............................................. 29
3.7.1 The Hidden State Problem ................................................. 29
3.7.2 Transparent Updates to Database ....................................... 30
3.7.3 Lack of Knowledge of Application Semantics ......................... 31
3.8 A General Invalidation Strategy ............................................. 31

4 Evaluation and Results ......................................................... 34
4.1 Evaluation Environment ..................................................... 34
4.1.1 Benchmarks .............................................................. 34
4.1.2 Client Emulator .......................................................... 35
4.1.3 Software ................................................................. 35
4.1.4 Hardware ................................................................. 36
4.2 Results ........................................................................... 36
4.2.1 RUBiS ....................................................................... 37
4.2.2 TPC-W ..................................................................... 42
4.2.3 Web Page Cache Size Requirement .................................. 46
4.2.4 Query Analysis Caching .................................................. 47

5 Related Work .................................................................... 49

6 Conclusions and Future Work .............................................. 52

Bibliography ........................................................................ 55
Illustrations

2.1 Architecture of Dynamic Web Applications .................. 5

3.1 Design of the Automatic Caching System .................. 14
3.2 Use of AspectJ to Achieve Transparency .................. 28

4.1 Response Times for RUBiS Browsing Mix .................. 37
4.2 Cache Hit Rates for RUBiS Bidding Mix .................. 38
4.3 Response Times for RUBiS Bidding Mix .................. 38
4.4 Breakdown of Hits by Request Type in RUBiS .................. 39
4.5 Breakdown of Overall vs Miss Response Time in RUBiS ........... 41
4.6 Extra Query Overhead for RUBiS Bidding Mix .................. 41
4.7 Cache Hit Rates for TPC-W Shopping Mix .................. 43
4.8 Response Times for TPC-W Shopping Mix .................. 43
4.9 Breakdown of Hits by Request Type in TPC-W .................. 44
4.10 Breakdown of Overall vs Miss Response Time in TPC-W ........... 45
4.11 Extra Query Overhead for TPCW Shopping Mix .................. 46
4.12 Cache Size for RUBiS .................................. 47
4.13 Cache Size for TPCW .................................. 47
Tables

2.1  A Database Table .................................................. 7

3.1  Cache entries table. Uses the request URI as a hash index to lookup the
      cached web page for the request. ................................ 17

3.2  Instances table. Uses the full SQL query string (a.k.a, query type) as the
      index to return the complete set of instances for the type. The
      instance-specific information are the dynamic values used by the instance
      and a link (the URI) to the cached web page. .................... 17

3.3  Write query analysis results table. Uses the full SQL query string
      (a.k.a, query type) as the index to return the summary of the query’s
      analysis, which is the internal representation of the query and the list of
      read query types with which the write could intersect, and extra queries
      needed for additional values (only for AC-extraQuery). .......... 18

3.4  Read query analysis results table. Uses the full SQL query string (a.k.a,
      query type) as the index to return the summary of the read query’s
      analysis, which is the internal representation of the query. The
      dependency relationships with writes are kept in Table 3.3. .......... 18

4.1  Query analysis cache statistics for RUBiS @ 1000 clients and TPC-W @
     400 clients .......................................................... 48
Chapter 1

Introduction

Dynamically generated web content represents a large portion of web requests, and the rate at which dynamic documents are delivered is often orders of magnitude slower than static content [5, 8]. Web sites for dynamic content are usually based on a multi-tier architecture implemented using several middleware systems: an HTTP server as a web front-end and provider of static content, an application server to execute the program generating the dynamic content, and a database to store the non-ephemeral data required by the application program. 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 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.

This thesis introduces AutoWebCache, a new method for server-side caching of dynamically generated web pages. This method provides both consistency and transparency with good performance. AutoWebCache maintains strong consistency, 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, other than ensuring that the application requests do not rely on hidden state to generate their response, i.e., all information required to form the web page is contained in the URI itself. Arguments from the URI in conjunction with the state of the database should determine the contents of the response.
We also ensure that all updates to the database go through the application server. Transparency is achieved through the use of AspectJ to automate the insertion of cache lookup and coherence operations at compile time.

In contrast to caching static content web pages, caching pages with dynamic content requires monitoring activity to the database. To maintain consistency, we need to detect modifications to the database that would change the content of a cached page when it was next requested, thus making the cached version obsolete. The effectiveness of the cache depends on how precisely invalidations can be made for cached pages, without generating false positives. A false positive is said to have been generated if a valid page is deemed invalid. Highly precise detection of when pages must be invalidated would offer the highest hit rate. On the other hand, a policy that generates too many false positives is easy to formulate and apply, but suffers from low cache hits. The trade-off is the performance cost of achieving a certain level of precision.

These actions are similar in intent to query caching at the database, but the target environment for the cache, the application server, only has access to the database via the query interface. Thus, many of the query cache design methods that require knowledge of the database implementation internals cannot be employed in this environment. Instead, this work relies on a form of symbolic analysis to generate tests for when invalidations must be performed. A contribution of this work is to explore a range of invalidation policies that increase precision by progressively more aggressive symbolic analysis of the SQL queries.

Another goal of this work is to minimize the involvement of the application designers in integrating the cache. I achieve an automated process of injecting the cache API calls using AspectJ, an aspect-oriented programming (AOP) environment [10]. The advantage of this method is simplicity. The disadvantage of this approach is that all query analysis must be done dynamically at run-time. This overhead can be considerable, especially for aggressive
invalidation protocols that require greater analysis for more precise information. Another contribution of this work is a demonstration of how to make this considerable run-time analysis overhead negligible by caching query analysis results.

To summarize, I present a caching model for dynamic web content - one that is both consistent with the backend database and also transparent to the developer. The model is unique because the existing systems offer only one of the two at a time. Specifically, my contributions through this thesis involve studying several strategies for determining pages to be invalidated in the cache and also development of a methodology for transparently injecting all caching logic into the application at run time.

The AutoWebCache method compares favorably to various state-of-the-art dynamic caching approaches. Many non-transparent approaches require extensive programmer intervention to indicate how cached data needs to be invalidated [7, 15, 16, 18]. A transparent approach may only provide time-lagged consistency [2]. In contrast, the method described in this thesis provides both transparency and consistency with precise invalidations.

I have evaluated AutoWebCache using standard software components, including the Apache web server, the Tomcat servlet server, and the MySQL database server, all running on current server-class x86 hardware. In my evaluation, I use the RUBiS auction site [1] and the TPC-W online bookstore [3] as my dynamic web content applications.

I compare five caching versions varying in degrees of selectiveness with which consistency is maintained, ranging from having no cache to aggressive hand-coded versions developed for each benchmark. The hand-coded version is a labor-intensive exercise that 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 the applications. The evaluation compares these two base versions to the three versions that can be selected in AutoWebCache. In all the AutoWebCache versions the information
required to detect conflicts is completely generated at run-time with no prior static analysis of the SQL queries at compile time. In contrast, the hand-coded versions have less overhead because this same information is statically determined (via the programmer) and is hard-coded into the application.

Measurements of AutoWebCache show that, in addition to consistency and transparency, my method provides good performance. In RUBiS, the average response time improves by 63% relative to no caching and compares well with the 71% improvement obtained from the hand-coded version. In TPC-W, the response time improvements are even more dramatic (> 95%) because the database easily becomes overloaded if there is no cache, but even at low loads I see minimum improvements of 50%. The benefits from caching are not distributed evenly among the different requests, with some requests benefiting from high web page hit rates in the cache and others having low hit rates. Furthermore, most performance benefits come from caching a few dominant (long latency) servlet types. Finally, I show that the overhead of doing run-time query analysis can be made negligible through caching intermediate results.

The rest of the thesis is organized as follows. Chapter 2 provides some background material. Chapter 3 describes the design principles of my caching system and provides implementation details. Chapter 4 describes the experimental environment and presents performance results. Chapter 5 describes related work, and Chapter 6 discusses further work and draws conclusions.
Chapter 2

Background

This chapter presents some basic background material required for understanding the work. In the first section, I present the architecture of dynamic web applications. Then I describe the structure of database queries that are used to access and modify the database in such applications. Finally, I proceed to give an overview of aspect-oriented programming. Informed readers are free to skip one or more of the sections that follow.

2.1 Dynamic Web Applications

![Diagram of Dynamic Web Applications]

Figure 2.1: Architecture of Dynamic Web Applications

Figure 2.1 shows an overview of a typical web page delivery mechanism for web sites with backend database systems. In a standard configuration, at the very front, there is a web server that acts as an interface for receiving all requests from the users and relaying back the generated responses. When a user asks for a particular page, the request and its
associated parameters, such as the product name and item number, are passed by the web server to an application server. The application server executes the business logic of the application. It performs the necessary computation to identify what kind of data it needs from the database and, then, sends appropriate queries to the database. Most e-commerce sites today utilize database management systems (DBMSs) to maintain business related data, such as prices, descriptions, and quantities of products. After the database returns the query results to the application server, the application uses these to prepare a page and passes it to the web server, which then sends it to the user.

In contrast to a dynamically generated page, we have a static page. As the name suggests, such a page is generated once and not each time on demand. A static page has no dynamic content involved and can be served to a user in a variety of ways. For instance, it can be placed in a proxy cache, a web server cache or a user side cache so that it can be delivered promptly following a request.

Caching of dynamically generated pages presents an interesting challenge. Most e-commerce web sites are sensitive to the freshness of the information provided to the user. It is important to note that the web servers, the application servers and the database servers are independent components. We therefore require an efficient mechanism to make database content changes reflect in the cached dynamic content pages. I explore such a mechanism in this thesis.

2.2 Structure of Database Queries

A database management system (DBMS) consists of:

- A collection of interrelated and persistent data (referred to as the database (DB)).

- A set of application programs used to access, update and manage that data (which
Table 2.1: A Database Table

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
<th>Major</th>
<th>City</th>
<th>GradePoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>John</td>
<td>Chemistry</td>
<td>Houston</td>
<td>3.9</td>
</tr>
<tr>
<td>Watson</td>
<td>Mary</td>
<td>Physics</td>
<td>Boston</td>
<td>3.3</td>
</tr>
<tr>
<td>Parker</td>
<td>Peter</td>
<td>Chemistry</td>
<td>Los Angeles</td>
<td>3.6</td>
</tr>
</tbody>
</table>

form the data management system (MS)).

The goal of a DBMS is to provide an environment that is both convenient and efficient to use while storing and retrieving data. Databases are usually designed to perform concurrency control if the system is shared by users.

A database most often contains one or more tables. Each table is identified by a name (e.g. Customers or Orders). Tables contain records (rows) with data.

Table 2.1 is an example of a table called Students. The table contains three records (one for each person) and five columns (LastName, FirstName, Major, City, and GradePoint).

SQL is an ANSI (American National Standards Institute) standard computer language for accessing and manipulating database systems. SQL stands for Structured Query Language. SQL statements are used to retrieve and update data in a database. With SQL, we can query a database and have a result set returned.

A query like this:

- SELECT LastName FROM Students WHERE Major=Chemistry

obtains the LastName for all the students in the table Students whose major is Chemistry. The result set is:
On the other hand, the query:

- SELECT FirstName, LastName FROM Students WHERE 3.5 < GradePoint < 3.7

returns the result set:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>LastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>Parker</td>
</tr>
</tbody>
</table>

The **WHERE** clause in the query is a boolean expression that evaluates to either **true** or **false** for each row in the table. All the rows evaluating to **true** are returned.

The SQL language also includes a syntax to update, insert, and delete records:

- **UPDATE** - updates data in the database table

  A query like this:

  ```sql
  UPDATE Students SET City=Chicago WHERE LastName=Watson
  ```

  sets the city to Chicago for all students whose last name is Watson.

- **DELETE** - deletes data from a database table

  The query
- DELETE FROM Students WHERE FirstName=Mary

deletes all the rows from Students where the first name is Mary.

- **INSERT INTO** - inserts new data into a database table

The query

- INSERT INTO Students VALUES (Brown, Alice, Biology, Austin, 3.4)

inserts a new record into the Students table for a student having last name as Brown, first name as Alice, majoring in Biology, living in Austin and having a grade point of 3.4.

### 2.3 Aspect-oriented Programming and AspectJ

Object-oriented programming has become a dominant programming paradigm whereby a problem is decomposed into objects that encapsulate both behavior and data in a single entity. Object-oriented methodologies reduce the complexity in writing and maintaining applications. They improve the ability of a software developer to achieve clear *separation of concerns*, or "the ability to identify, encapsulate, and manipulate only the parts of software that are relevant to a particular concept, goal, or purpose" [14].

Although object-oriented programming offers greater ability for separation of concerns, it still has difficulty localizing concerns which do not automatically fit into a single program module, or even several closely related program modules. Concerns can range from high-level notions such as security and quality of services to low-level notions like buffering, caching, and logging. Such functionality might not be contained within a single class but distributed across many classes or often the whole of application.
We say that two concerns \emph{crosscut} if the methods related to those concerns overlap. For example, two objects encapsulating different data entities might require the same logging behavior or the same buffering mechanism. Through object-orientation, however, such code ends up being scattered across many modules.

Aspect-oriented programming is a method of programming that modularizes orthogonal functionality in an application having such a crosscutting structure. It encapsulates behaviors that affect multiple classes into reusable modules. Crosscutting concerns are separated into single units called \emph{aspects}. An aspect is a modular unit of crosscutting implementation. A single aspect can contribute to the implementation of a number of procedures or modules increasing reusability of the code. For the purpose of this thesis, caching code exhibits precisely the properties desired for applying aspect-orientation - each request requires cache lookup, addition of an entry to the cache and removal of an entry from the cache. Each behavior is described by its own aspect.

\subsection{Introduction to AspectJ}

AspectJ is an implementation of aspect-oriented programming environment (AOP) for Java [10]. AspectJ provides a simple language with which to specify points of interest in the code, defined by \emph{pointcuts}, and the code to inject at these points, called \emph{advice}. Pointcuts are specified using a combination of regular expression matching and object type information. The initial execution points of the requests are identified with \texttt{javax.servlet.http.HttpServlet}'s \texttt{doGet()} and \texttt{doPost()} methods. The pointcut for \texttt{doGet()} can be defined as follows:

\begin{verbatim}
pointcut
  atDoGet(HttpServletRequest request,
           HttpServletResponse response):
\end{verbatim}
The point cut identifier is labeled `@DoGet` and specifies two parameters to which the advice should have access. These two parameters are arguments (`args()`) to the `doGet` call. The pointcut is at the execution point of the `doGet` method (i.e., inside the method rather than the call site before its execution).

While pointcuts indicate where points of interest are in the program, advice specifies what actions should be taken at the selected sites. Advice can be injected before, after, or around the pointcuts. The names are self explanatory. The differences are that before advice executes before the pointcut and does not have access to return values, if any, at the pointcut. After advice can access return values. Around advice is used to inject actions in lieu of the actions of the pointcut. Around advice is the only advice which can affect the flow of the original code or the values used.

For more details on AspectJ, interested readers should visit `http://aspectj.org`. 

```java
args(request, response)
&&
execution(* doGet(HttpServletRequest,
            HttpServletRequest))
&&
excludeAspectCode();
```
Chapter 3

Design of the AutoWebCache

The policy used for maintaining cache coherence is the driving factor in the design of a web caching system. A caching system that permits web pages to contain out-of-date (inconsistent) data with the database for a window of time [2] eliminates much of the complexity of maintaining coherence. In contrast, a design that maintains strong coherency between the cached web pages and the backing database must resolve a number of issues that have ramifications on both the overall complexity of the cache and its performance overhead.

The challenge is particularly acute for automating a strong coherency protocol because selectivity in doing invalidations is essential to the effectiveness of caching. While simplistic protocols offer easy implementation, they risk excessive invalidations. On the other hand, highly sophisticated invalidation protocols that maximize the hit rate through precise invalidations require considerable analysis of the SQL database queries to develop precise tests of overlap between read-only (read) and database modifying (write) queries. If the application designer is burdened with the task of performing this analysis then aggressive protocols can be hard-wired into the application code that are both precise and have little overhead. The challenge I address with the AutoWebCache design is how to automate the implementation of precise invalidation protocols such that the overhead is negligible and no semantic knowledge from the developers is needed.
3.1 Overview

In brief, a web page must be invalidated when a write to the database results in modifying the data set (returned by a prior read query) used to generate the content of the now cached page. Determining which pages to invalidate can be difficult. In such cases, one can conservatively invalidate those pages that might be affected. However, this can result in a poor hit rate. More sophisticated invalidation detection policies could give us higher hit rates than those that conservatively invalidate many valid pages.

There are various levels of precision for detecting if a write query intersects with a read query. A simple method is to check if the columns used in the read query are also updated in the write query. This column-only check may result in many false positive indications that an intersection exists when, in fact, there is no intersection. However, this method is safe since invalidating cached entries cannot violate correct coherency, though performance may suffer.

To make the test for intersection more precise, selection criteria in the read query instance’s WHERE-clause can be re-evaluated using values from the write query in the hope of showing that the element of the write does not satisfy the selection criteria of the read query. For example, if the read’s predicate requires that $T.a=X$, but for the write query $T.a=Y$ and $X \neq Y$, then the queries cannot intersect.

Invalidates can be made even more precise by executing extra queries. Continuing with the prior example, if the value of the field $T.a$ is not specified in the write query itself, then an extra query can be made to the database for the value. This value would help us in reducing the number of (valid) pages that need to be invalidated. Of course, the benefits of invalidating fewer pages must compensate for the cost of these additional queries.

In this work I explore the proper balance between invalidation precision and its associated performance cost. Precision is determined by the invalidation strategy. Its cost
is determined by the **detail of query analysis** to extract the information and relationships needed by the strategy, as well as the dynamic costs of intersection detection itself. While these two points are my primary focus, I also present the implementation details of the cache data structure and explain how the cache API calls are transparently injected into the application server code. An overview of the design is shown in Figure 3.1.

In the rest of this chapter, I first describe the consistency model provided by my caching system. Then, I explain the data structure used to maintain the cache of web pages. Next, I describe the invalidation strategies used to determine the validity of pages. I then proceed to outline the query analysis needed to generate dependencies between the queries. Finally, I describe the procedure for automatically injecting the caching logic into the application, without any effort from the developer. The last two sections of this chapter present some
limitations of automatic caching mechanisms and outline a general strategy for detecting page invalidations, respectively.

### 3.2 Consistency Model

Addition of caching logic to the application should not change the behavior of the original system. Formally, we say that the caching system is completely consistent if it

1. Provides the same guarantees to the clients as the original application.
2. Maintains the model used by the server to generate web pages.

The original system provides the following guarantee to the client

- If the user issues one or more read requests accompanied by a write request, the effects of write might not be visible in responses of the read requests. However, the effects of write are visible (to all users) in all the read requests issued after response to the write is available.

This guarantee is provided by the caching system as well.

I consider the following model to be used by the application to generate web pages:

- The web pages generated always reflect the current state of the database.
- The system provides no guarantees about the order in which concurrent requests are executed by the application server.

Addition of a web page cache to the application maintains this model. Specifically,

- Any pages that become invalid as a result of modifications to the database are immediately removed from the cache.
• The order in which concurrent requests complete might change due to introduction of the cache. However, since the order is not guaranteed by the original application itself, the caching system does not introduce any violations in this context.

Therefore, my system provides complete consistency, both with respect to the client and the server.

3.3 Cache Structure

The design of the cache data structures is common to all invalidation strategies studied in this thesis since each strategy requires similar functionality: cache lookup, addition of an entry and deletion of an entry from the cache. The AutoWebCache is composed of four primary storage structures. These structures are a cache entries table for web pages (Table 3.1), a query instances table (Table 3.2), a write query analysis results table (Table 3.3), and a read query analysis results table (Table 3.4). I now discuss each in turn.

The cache entries table represents the cache of web pages itself. Web pages in the cache entries table are indexed with a hash of the URI of a client request. If such an entry exists, the associated cached web page is returned as a result. On a cache miss, the request code is run and the newly formed page is added to the cache.

Additionally on a cache miss, the SQL queries executed to generate the page are captured and processed for the required information needed to support intersection testing and invalidations. Each SQL query string is tokenized and processed to extract the list of tables referenced, the list of columns read/written, the field values if specified in the string (i.e., constant), and the WHERE-clause expression. The WHERE-clause is the predicate expression that needs to evaluate to true for the row in the table to be selected/updated.

Consider a simple query example

SELECT T.a FROM T WHERE T.b=10
Table 3.1: **Cache entries table.** Uses the request URI as a hash index to lookup the cached web page for the request.

<table>
<thead>
<tr>
<th>Index: URI (readHandlerName + readHandlerArgs)</th>
<th>Cached web page</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI₁</td>
<td>WebPage₁</td>
</tr>
<tr>
<td>URI₂</td>
<td>WebPage₂</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3.2: **Instances table.** Uses the full SQL query string (a.k.a. query type) as the index to return the complete set of instances for the type. The instance-specific information are the dynamic values used by the instance and a link (the URI) to the cached web page.

<table>
<thead>
<tr>
<th>Index: SQL string</th>
<th>&lt;Value vector, URI&gt; pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadQueryType₁</td>
<td>&lt;instance values₁₁, URI₁&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;instance values₁₂, URI₂&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;instance values₁₃, URI₃&gt;</td>
</tr>
<tr>
<td>ReadQueryType₂</td>
<td>&lt;instance values₂₁, URI₄&gt;</td>
</tr>
<tr>
<td>ReadQueryType₃</td>
<td>&lt;instance values₃₁, URI₅&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;instance values₃₂, URI₆&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;instance values₃₃, URI₇&gt;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Table 3.3: **Write query analysis results table.** Uses the full SQL query string (*a.k.a.*, query *type*) as the index to return the summary of the query’s analysis, which is the internal representation of the query and the list of read query types with which the write could intersect, and extra queries needed for additional values (only for *AC-extraQuery*).

<table>
<thead>
<tr>
<th>Index: SQL string</th>
<th>Intermediate form</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>WriteQueryType₁</td>
<td>AnalysisResultsₜ₁</td>
<td>{&lt;ReadQueryType₁,extraQry₁&gt;,&lt;ReadQueryType₃,extraQry₃&gt;}</td>
</tr>
<tr>
<td>WriteQueryType₂</td>
<td>AnalysisResultsₜ₂</td>
<td>{&lt;ReadQueryType₂,extraQry₂&gt;}</td>
</tr>
<tr>
<td>WriteQueryType₃</td>
<td>AnalysisResultsₜ₃</td>
<td>{&lt;ReadQueryType₁,extraQry₁&gt;}</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Table 3.4: **Read query analysis results table.** Uses the full SQL query string (*a.k.a.*, query *type*) as the index to return the summary of the read query’s analysis, which is the internal representation of the query. The dependency relationships with writes are kept in Table 3.3.

<table>
<thead>
<tr>
<th>Index: SQL string</th>
<th>Intermediate form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadQueryType₁</td>
<td>AnalysisResultsᵣ₁</td>
</tr>
<tr>
<td>ReadQueryType₂</td>
<td>AnalysisResultsᵣ₂</td>
</tr>
<tr>
<td>ReadQueryType₃</td>
<td>AnalysisResultsᵣ₃</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
In this case, the columns selected are \{T.a\}, the table concerned is T and the WHERE-clause is T.b=10

Furthermore, the WHERE-clause is converted to an abstract syntax tree form for easy evaluation. This information represents the intermediate form of the query string and is stored as the analysis results for the query in either Table 3.3 or Table 3.4 depending on the query type (write or read). I call the original SQL query string the query type and use it as the hash index into the tables. Every occurrence of a query string is an instance and includes a vector of dynamic values set at the time of the occurrence. Thus, while the number of encountered query instances may be large, the number of unique query types is small. The number of unique query types is dependent on the number and type of requests offered by the application. The number of query instances, on the other hand, depends on the domain of arguments permitted for each request, which can be large.

The dependency relationships between write and read query types are kept with the write query type in Table 3.3. If the new query type is a write then it must be tested against each of the read query types in Table 3.4. If it is possible that a read query type could intersect with the write query type, then its index is added to the dependencies list for the write type. Similarly, if the new query type is a read then it must be tested against each of the write query types in Table 3.3 for possible intersection and added to the appropriate dependency lists. In addition, for the AC-extraQuery version, any extra queries for additional information needed for the invalidation tests are paired with their read query type in this table.

Intersection testing is performed by re-evaluating the WHERE-clause of the read query type using known values specified in the write query type. Intuitively, this check determines if the modification would either add or subtract an element from the data set of a read query if the read query were to be executed again. Missing values conservatively are
assumed to take values that maximize the possibility of the evaluation resulting in returning \texttt{true}, in which case the read query type is added to the appropriate dependency list. Tables 3.3 and 3.4 are updated only when a new SQL string is encountered. Since the number of unique query types in the e-commerce applications that we have reviewed is small, these two tables become stable quickly and are essentially static after only a short warm-up period. During the actual measurement interval, these two tables act as cache of the analysis results.

The instances table (Table 3.2) maintains the details of the specific dynamic instances of read query types. Indexed by the SQL string (its type), the vector of dynamic values used in the instance are paired with the URI of the web page generated. When a write query occurs, Table 3.2 is accessed with each of the read types in its dependency list. For each dependent read type, the set of dynamic instances of that type is returned (which also defines the set of cached pages that could be affected). For each instance returned, the read type's WHERE-clause (available from Table 3.4) is evaluated using the instance's dynamic values and those of the write instance. If the evaluation returns \texttt{true}, the associated URI is used to locate the page in the web page cache for invalidation.

The number of instances associated with a read query type may be large. In general, an intersection test must be performed for each instance. To minimize the number of these evaluations, I implement an optimization for the common case of conjunctive equality tests, \textit{e.g.}, \texttt{T.a==X AND T.b==Y}. It is sufficient for expressions of this form to know if the pair \{X, Y\} was used to form any page. Part of processing the read queries identifies such cases in the WHERE-clauses and instructs the value set be added to an evaluation map at each occurrence. Upon a write that must check for intersection, it suffices to form its own value set and simply perform a constant time look-up in the hashmap to see if this same set has occurred, instead of evaluating each of the instances from the instance table.
Tables 3.1 and 3.2 are modified continuously as pages are added and deleted from the cache. In contrast, Tables 3.3 and 3.4 which hold results of the query analysis are modified very infrequently, only when a new query string (type) occurs. In the results of Section 4.2 I show that these latter two tables are the key feature that enable good performance despite performing the relatively complex \textit{(i.e., time-consuming)} query analysis at run-time.

### 3.4 Invalidation Strategies

Invalidations are generated when a write to the database intersects data sets used to form cached pages. The data sets are defined by the selection criteria \textit{(i.e., WHERE-clauses)} of the read queries and intersection can be directly tested by re-evaluation of the predicate with values from the write. Pages whose data sets intersect with the write must be deleted from the cache.

Different invalidation strategies can be defined that vary in the level of detail with which intersection tests are performed. I explore three invalidation strategies: \textit{AC-columnOnly}, \textit{-valueBased}, and \textit{-extraQuery}. \textit{AC} stands for Automatic Cache, to emphasize the automation of cache code injection. They all assume the same cache structure as discussed in the prior section.

**AC-columnOnly.** One of the most simple tests is to compare if any of the columns used in the read query are updated by the write query. This test can be done once during query analysis since column names are fully specified in a query and no run-time checks need to be made. Of course, in the event of an insert or delete such a test will always return positive for intersection.

I add an optimization to this basic strategy. It is often the case in e-commerce applications that a read is predicated on a column that is a primary key, such as an item ID,
which is unique. Thus, an insert of a new element can never intersect with a prior read whose predicate selected a single existing row based on the unique primary key. This case is detected during query analysis and the read query is not added to the list of dependent predicates for the write.

**AC-valueBased.** In this strategy, during analysis, a list of read predicates sharing columns with the write query are added to its list of dependent queries. On the occurrence of a write, each of its dependent read query predicates are re-evaluated for intersection with the dynamic values from the write, which are available from the query itself (or implied as defaults). Also required are the dynamic values used in the read queries at the time they were executed. Query analysis specifies with the cached query analysis results which dynamic values from the query instance must be saved when the page is cached.

**AC-extraQuery.** Finally, query analysis can detect if values needed to evaluate a read predicate are not available in the write query. In this case, the analysis engine can generate an extra query to retrieve the missing values not specified in the write query. Consider the following example:

- Read query R - SELECT T.a FROM T WHERE T.b=?
- Write query W - UPDATE T SET T.a=? WHERE T.c=?
- Extra query R_x - SELECT T.b FROM T WHERE T.c=?

The read query is conditioned on T.b, but the value of this field is unknown in the write query. Performing an extra query for this value during an intersection test allows for more precise invalidations. The process of generating the extra queries is a simple mechanical process. The fields listed in the SELECT portion of the extra query are those
that are a) in the read predicate but not known from the write and b) are used in equality
tests (the only relation we consider currently). The predicate of the extra query is simply
the predicate of the write query (T. c=? ) filled in with its dynamic values. For further
details on generation of extra queries, see Section 3.5.4. It must be emphasized, however,
that the cost of doing the extra query should be compensated by the reduction in the number
of valid pages removed from the cache.

These three invalidation strategies are strictly progressive in their preciseness of de-
termining invalid pages, with AC-columnOnly being the least precise and AC-extraQuery
the most precise. AC-columnOnly generates invalidations based on simple column testing.
AC-valueBased makes use of the values assigned in the predicate expression of the queries,
besides column testing. AC-extraQuery goes a step further and obtains all missing values
through extra probes to the database. To minimize the number of extra queries issued, this
strategy issues a single database query for all the missing column values. Having obtained
all these values, it proceeds to determine whether an invalidation should occur or not.

3.5 Query Analysis Engine

The query analysis engine has the task of specifying the predicates to be cached and the
dependencies between queries. This analysis is performed on queries automatically during
run-time and the results are cached in Tables 3.3 and 3.4. The analysis requires parsing the
query string, intersection testing, and analysis caching.

3.5.1 Parsing

SQL queries are given as a string with a vector of dynamic values to be inserted. Parsing
separates a query into the fields being selected, the tables being referenced, and the selec-
tion criteria of the WHERE-clause. The query is output into a format usable for internal
evaluation and further processing. The tables and fields for the SELECT are stored in hash tables. The WHERE-clause is parsed to identify values set as constant and those to be set dynamically (delineated by a '?' in the string, as well as converting the expression into a form appropriate for evaluation (see Section 3.5.2 for these details).

### 3.5.2 Intersection Testing

The queries making up RUBiS and TPC-W benchmarks rely heavily on selection criteria using the equality operator. Since intersection evaluation is performed repeatedly, I optimize AutoWebCache performance by limiting the evaluation to simple equality expressions using conjunction (e.g., A=X AND B=Y) and assign wildcard values to other types of expressions. The wildcard marker is defined so that it combines in evaluations to maximize the return of true, indicating possible intersection which is the conservative result. The importance of conjunctive equality tests in the two benchmarks appears to be a result of the IDs of customers, items, and categories being the focus of most queries and suggests most e-commerce applications will be similarly structured.

The analysis also specifies which fields to use as the index into the evaluation map in the optimization discussed in detail in Section 3.3 for conjunction equality expressions. This optimization enables a constant time detection of intersection over any number of query instances.

Of course, the evaluation engine can be extended to support evaluation of more complex expressions in the WHERE-clause, if needed. An abstract syntax tree-like structure with tuples (Vectors) can be used consisting of an operation object and the appropriate number of operands, e.g., (operation, operand, operand). Operands may be references to simple objects such as an integer, or can be references to other tuples to create arbitrarily complex nested expressions. If a type of sub-expression is not supported then such expressions can
always be replaced by a wildcard marker.

3.5.3 Analysis Caching

The overhead to analyze a new query string can be considerable. Fortunately, new analyses need only be done when new query strings (types) are encountered. In practice, the number of unique sites in the code that specify SQL query strings is small, 32 types for RUBiS and 24 types for TPC-W. Thus, the analysis results can be cached in a separate small query analysis cache, Tables 3.3 and 3.4, whose contents will become effectively unchanging once all query strings are seen. The details of these tables are covered in the cache data structure discussion of Section 3.3. I show in Section 4.2 that the caching of analysis results is critical to overall performance and makes the impact of query analysis on performance negligible.

3.5.4 Generation of Extra Queries

Intuitively, we want to find out all the missing values needed to compute whether an invalidation occurs or not. For this, on each occurrence of a write query, we aim to find values of all variables occurring in the read predicate. We then would evaluate the read predicate with these values. If the predicate evaluates to true, we have an intersection leading to invalidation. We looked at a simple example earlier, I present some more in this subsection.

- Read query R - SELECT T.a FROM T WHERE T.b=?
- Write query W - DELETE FROM T WHERE T.c=?

In this case, we want to find out whether one (or more) of the rows deleted have the value of T.b same as that occurring in the read query. For this, we issue an extra query

Extra Query $R_x$ - SELECT T.b FROM T WHERE T.c=?
Now, consider the following pair of queries:

- Read query R - SELECT T1.a, T2.x FROM T1,T2 WHERE T1.b=? AND T2.y=?

- Write query W - UPDATE T1 SET T1.a=? WHERE T1.c=?

In this case, we select columns from two tables, but update only one of these. The predicate in the read query needs two values, T1 . b and T2 . y. We, however, need to fetch only T1 . b because it belongs to the table being updated. We get this value by issuing an extra query

Extra Query $R_x$ - SELECT T1.b FROM T1 WHERE T1.c=?

Finally, for the following pair of queries:

- Read query R - SELECT T1.a, T2.x FROM T1,T2 WHERE T1.b=? AND T1.c=T2.y AND T2.z=?

- Write query W - UPDATE T1 SET T1.a=? WHERE T1.d=?

The read predicate in this case involves values of two columns T1 . b and T2 . z, and a condition T1 . c=T2 . y. We fetch the value of T1.b as before, but also use the condition T1 . c=T2 . y to restrict the set obtained. The extra query is

Extra Query $R_x$ - SELECT T1.b FROM T1, T2 WHERE T1.c=T2.y AND T1.d=?

### 3.6 Code Injection with AspectJ

AspectJ is an implementation of aspect-oriented programming environment described in Section 2.3. A single file of 142 lines of AspectJ specification suffices to direct where to
inject the cache API calls and to specify the necessary data gathering code. AspectJ relies on a combination of regular expression matching and type information to match patterns in the source code at compile time. Points in the code that match the specification patterns trigger AspectJ to insert the associated caching code. Requests are detected by matching to the \texttt{doGet()} and \texttt{doPost()} methods. For AutoWebCache, insertion points are only required for a small set of standardized java library methods. Thus, the specification file is general for e-commerce applications.

To ensure that I do not break the original application when the cache code is injected, AutoWebCache follows a simple policy. The policy is that the original request code is either run in its entirety or it is bypassed completely. Modifications within the original control flow or to values used are not permitted. To inject the AutoWebCache, I place \textit{around} advice within the bodies of the \texttt{doGet()} and \texttt{doPost()} methods. Before the original body of the method is executed, the advice accesses the cache with the URI request. A web page is returned if the cache contains the page. In this case, none of the code in the original method body is executed.

On a cache miss, the original application code is executed by specifying the AspectJ call \texttt{proceed()}. The complete body of the original method is run. Just prior to the exit of the method, the advice injects a cache post-processing call to add the page to the cache if the request is read-only.

Also on a cache miss during the regular processing, additional advice code records the activity of the request; specifically, for each \texttt{PreparedStatement} instance I record the SQL query string and the associated values set (via its \texttt{set*()} methods, where * is the value type: \texttt{int}, \texttt{String}, etc.). The queries and values are captured with \textit{after} advice at the appropriate call pointcuts.

The specific points within the application request code I instrument are the following:
Figure 3.2: Use of AspectJ to Achieve Transparency

- `Connection.prepareStatement(String)` : I capture the SQL query string here.

- `PreparedStatement.set*()` : I capture the dynamic values in the SQL query here.

- `PreparedStatement.execute*()` : I need the point at which the query is actually executed to mark when query formation is complete and the query can be analyzed.

The SQL strings and values are sent as parameters to the cache’s `queryAnalysis()` method at a point when I know the `PreparedStatement` is fully formed, e.g., immediately after its `PreparedStatement.executeQuery()` method. The run-time analysis
parses the SQL query and detects data dependencies with other queries as well as recording specific values used in the query.

Overall, AspectJ is used to provide wrappers to capture activity between the web and the application server and also between the application and the database server. Labelling the intercepting wrappers of Section 3.1 with AspectJ leads us to Figure 3.2.

My AspectJ file assumes standard interfaces for communication between the servers. Any changes or addition to the set of interfaces must be accompanied by the corresponding changes or additions in the AspectJ code.

The code transformation is performed at compile time and requires the source code or byte code of the application. Because its specification model is simple and intuitive, I was able to quickly automate the instrumentation process.

3.7 Limitations of Automatic Caching

3.7.1 The Hidden State Problem

Implied in the design of AutoWebCache is that the HTTP request contains all the information necessary for the request to create the web page. Thus, identical requests (which will map to the same cache entry) result in the same page being generated. Any other state that affects the web page content is considered hidden state, e.g., static variables, random number generators (TPC-W’s promotional frames), and timeouts (TPC-W’s best-sellers request). It is the responsibility of the application developers to ensure read requests do not rely on hidden state, or requests that do are declared as uncacheable via an input file read by the cache at initialization.

Generally speaking, a request is said to contain a hidden state if two invocations with the same arguments and having the same database state produce two different web pages.
An SQL level analysis of each request does not provide us with the semantics of the request, and hence, the inability to detect such hidden state automatically. For automatic detection, we have to explore the requests for their semantics, possibly using flow analysis. It is not clear, however, if we can guarantee the detection of non-cacheable requests arising from hidden states. Intuitively speaking, any analysis that aims to do so would have to be intelligent enough to detect the non-determinism present because of the use of random numbers, timeouts and the like. Acquiring this intelligence is an interesting topic of research by itself, and warrants more thorough investigation. The general solution to this problem is likely untractable in the foreseeable future, but compiler analysis can address specific cases. For example, the compiler might be instructed to look out for certain keywords and regular expressions in the application code.

Lastly, a hidden state problem is present in any solution that transparently adds caching logic to the application. It is not an artifact of the solution I propose and in this context can be viewed upon as being present in any automatic solution.

3.7.2 Transparent Updates to Database

The current implementation of my system assumes that all updates to the database go through the application server. My approach is completely transparent in this case. If this is not the case then transparency is difficult to achieve. However, the approach can easily be extended with an API similar to the ones provided by Dynamic WebCache [7] and Weave [18] to allow an external entity to invalidate cache entries or simply flush the cache. This external entity could, for instance, work through database triggers. If external updates are rare then there should be little impact on performance.
3.7.3 Lack of Knowledge of Application Semantics

The hand-coded caching strategy computes all of the invalidation dependencies by hand, and the cache calls are inserted manually in the application code. While generating these dependencies, we can take advantage of application level semantics not available to automatic strategies. Consider the following pair of a read and an insert query:

- Read - SELECT * FROM bids, items WHERE bids.item_id=items.id

- Insert - INSERT INTO items {value set}

The table bids contains information about bids on all available items, whereas the table items contains a description of these items. From our knowledge of the application, we know that unless an item exists in the items table, there can be no entry for it in the bids table. Hence, in the above pair, when we insert a new entry into the items table, we know from this knowledge that this would not affect the read query - since this is a first time item, there would be no entry in the bids table with item_id equal to the id of the item. The expression bids.item_id=items.id can never be satisfied by a new item entering the items table.

This information, however, is not available to the automatic caching strategies, which must either conservatively invalidate the results of the read query or do extra work at run time to determine the non intersection between the two queries.

On the other hand, if in the above example, the bids and the items tables were unrelated, the optimization for the hand-coded scheme would not hold.

3.8 A General Invalidation Strategy

We have seen three invalidation detection strategies: those based on columns, dynamic values or extra queries to the database. In this section, I outline a general invalidation
strategy.

First, consider two queries, a read and an update:

- Read - SELECT * FROM T WHERE EXP₁
- Update - UPDATE T SET T.a=5 WHERE EXP₂

The write query can affect the results of the read query iff EXP₁ ∧ EXP₂ can evaluate to \text{true}. More generally, to detect intersection we write the expressions in a conjunctive normal form (CNF) and check the satisfiability of the resulting formula.

Sometimes it is possible to determine that the formula is unsatisfiable even without knowing all the run-time values used for the query instances. Consider the following set:

- SELECT T1.a FROM T1,T2 WHERE 10 < T1.b < 20 AND T1.c=8 AND T2.d > 70
- UPDATE T1 SET T1.a=5 WHERE 30 < T1.b < 40

Although the update query doesn’t tell me anything about the values of T2, and so I have to assume that the last two clauses of the selection predicate are true, the intersection is bound to be empty. On the other hand, if the update query is

- UPDATE T1 SET T1.a = 5 WHERE 15 < T1.b < 25

I need to check the value of T1.b and T1.c to see if the read query is affected.

Similarly for the pair

- SELECT * FROM T WHERE T.a=5
- UPDATE T SET T.b=12 WHERE T.a=10

We can say that the queries would not intersect, but for the pair

- SELECT * FROM T WHERE T.a=?
• UPDATE T SET T.b=12 WHERE T.a=?

We have to rely on the dynamic values of column 'a' to be able to determine a true intersection.
Chapter 4

Evaluation and Results

This chapter presents the environment used for evaluation of my caching system and invalidation strategies. I also describe the benchmarks used for evaluation and present a set of results obtained for each.

4.1 Evaluation Environment

4.1.1 Benchmarks

I evaluate the caching configurations on two benchmarks, RUBiS and TPC-W. RUBiS implements the core functionality of an auction site: selling, browsing and bidding [1]. TPC-W implements an on-line bookstore [3]. The two benchmarks differ significantly in where they stress the system. In both applications, clients generate requests to a web and application server which then shuttles SQL queries to a database server to gather values for generating dynamic web page content. RUBiS is designed to stress the web and application server with many requests that can be quickly satisfied by the database server. In contrast, TPC-W stresses the database server by issuing requests that generate complex SQL queries.

TPC-W has three different workload mixes that vary in the ratio of reads to writes. The browsing mix is 95% read-only, the shopping mix is 80% read-only, and the ordering mix is 50%. There are two mixes for RUBiS, a browsing mix that is 100% read-only and a bidding mix with 85% read-only queries. RUBiS is a less well-known benchmark than TPC-W but offers an interesting contrast. RUBiS has been used in other dynamic
content web site experiments and its code is freely available from www.objectweb.org. I use an implementation of TPC-W originally proposed by the University of Wisconsin [9] but modified to support web page frames.

### 4.1.2 Client Emulator

Both benchmarks use a client-browser emulator to generate requests. 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 of going from one interaction to another.

The think time and session time 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 v1.8 specification [3]. I vary the load on the site by varying the number of clients. For RUBiS, experiments start with a cache warmed up for 15 minutes. The measurements are collected during the following 30-minute period. For TPC-W, warm-up is 60 minutes and the measurement interval is 30 minutes.

### 4.1.3 Software

I use Apache v1.3.22 as the Web server. I 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 MySQL v2.04 type 4 JDBC driver [12], running on Sun JDK 1.4.2. I use MySQL v3.23.43-max [13] as my database server with the MyISAM tables. All machines run the 2.4.20 Linux kernel.
4.1.4 Hardware

Each machine has an Intel Xeon 2.4GHz CPU with 1GB ECC SDRAM, and a 120GB 7200 RPM disk drive. Three machines are used: one for the client emulator, one for both the web and application server, and one for the database server. I have verified that in none of the experiments the client emulator is the bottleneck. All machines are connected through a switched 1Gbps Ethernet LAN.

Choosing one machine for both the web and application server does not influence the results in the context of caching. Separating the two on different machines would add the same communication overhead in both the cached as well as the original uncached system. In a scenario such as ours, where there is only one web server and one application server, the web server acts as an interface to the application server. It is, therefore, not uncommon to keep them on the same machine.

4.2 Results

This section presents the experimental results of the evaluation of the three proposed caching schemes: AC-columnOnly, AC-valueBased, and AC-extraQuery. I include two additional comparison points: No cache and Hand-coded cache. These last two represent practical lower and upper bounds on performance. The hand-coded cache, like the AC-extraQuery scheme utilizes extra queries to minimize the number of pages invalidated. However, as the name suggests, all the dependencies are generated manually and hard-coded into the application. In all simulations, the cache size is not limited since memory is not a bottleneck.
4.2.1 RUBiS

Figure 4.1 shows the response times in RUBiS running the browsing mix which is read-only and, thus, does not generate cache invalidates. I ran with 50, 100, 300, 500, 600, 700, 800, 900, and 1000 clients. For the browsing mix, the cache hit rate is over 90% and will tend to 100% on even longer runs. Shown are only three cache configurations: *No cache*, *Hand-coded*, and *AC-columnOnly*. Since the different caching schemes primarily differ in how invalidates are done, without updates all the cache configurations are similarly effective. Caching can clearly provide a performance advantage.

Using the bidding mix, which has updates, differentiates the cache designs. Figure 4.2 is a bar chart of the hit rates for each cache design when running the bidding mix, for 1000 clients.* *AC-columnOnly* that uses the simplest, and least precise, invalidation method has the lowest hit rate at 16%. The web pages making up this fraction are rarely

---

*In all reporting of hit rates, only the population of dynamic content pages is included; static content pages are not cached. They are generated by the application upon each request.*
Figure 4.2: Cache Hit Rates for RUBiS Bidding Mix

Figure 4.3: Response Times for RUBiS Bidding Mix
Figure 4.4: Breakdown of Hits by Request Type in RUBiS

if ever invalidated. If detecting invalidates by column intersection is made more selective through dynamic checks of known value fields as is done in AC-valueBased then the hit rate increases to 39%. AC-extraQuery further improves the hit rate to 52% by actively querying the database for values that can reduce the possible region of intersection with existing cached pages and achieves essentially the precision of the Hand-coded cache. Although hit rates are reported for 1000 clients, they remain essentially the same for other client numbers.

Shown in Figure 4.3 is the effect of caching on response times. The effectiveness of caching ranges from none to quite significant as the precision of invalidates increases. In fact, the selective caching of AC-extraQuery matches that of the Hand-coded version (their plots lie on top one another). A breakdown of the requests where each cache design is effective reveals an interesting behavior in the progression of improvements.

In Figure 4.4 I chart the percentage that each request type occurs in the RUBiS bidding mix; for example, the first request type listed, About Me that generates information about
the current user, makes up about 4% of all requests. The AutoWebCache versions progressively generate more refined invalidates. Thus, the sets of requests that hit in the cache form a subset relationship. Because of this, I can conveniently subdivide each bar in Figure 4.4 into five categories: hits that occur in the \textit{AC-columnOnly} design (the base hit rate), additional hits occurring in the \textit{AC-valueBased} design, those in the \textit{AC-extraQuery} cache, then those in the \textit{Hand-coded} cache, and finally the remainder are the misses occurring in all cache versions.

What is immediately obvious from the figure is that different request types can be effectively cached as the invalidation policy becomes more selective. The simplest caching scheme \textit{AC-columnOnly} effectively caches the requests \texttt{Browse Cat} and \texttt{Browse Rgn}. The more complex caching designs cannot improve the hit rate here. However, with \textit{AC-valueBased}, \texttt{Search Cat}, \texttt{View Item}, and \texttt{View User} become cacheable. The most selective caching scheme \textit{AC-extraQuery} makes caching effective for the two request types \texttt{About Me} and \texttt{Search Rgn}, while improving caching for \texttt{Search Cat}. Requests like \texttt{Buy Now}, \texttt{Put Bid} and \texttt{Put Comment} remain largely uncachable, essentially suffering from cold misses. \textit{Hand-coded} caching performs almost the same as \textit{AC-extraQuery}.

The reason \textit{AC-columnOnly}'s 16% hit rate has small performance benefits in Figure 4.3 is because it caches requests with low latency that make up only a small portion of the overall average response time of \textit{No Cache}. This fact is clearly shown in Figure 4.5 that graphs the response time for misses of each request type (the top of the bars) over and above the average response times (the dark portion of each bar) including both cache misses and hits, for the caching scheme using \textit{AC-extra query}. The miss response time of the two categories \texttt{Browse Cat} and \texttt{Browse Rgn} are relatively minor compared to the other requests, thus caching them has little benefit. In contrast, the full set of request types \textit{AC-valueBased} effectively caches are somewhat more costly and because of their frequency
Figure 4.5: Breakdown of Overall vs Miss Response Time in RUBiS

Figure 4.6: Extra Query Overhead for RUBiS Bidding Mix
they make up almost a fourth of the No Cache average response time. The advantage of AC-extraQuery is that the set of requests it effectively caches make up almost two-thirds of the No Cache average response time, thus, reducing the response time significantly. This improvement for AC-extraQuery occurs despite generating 21% additional queries over those from the original application code. Also, although AC-extraQuery has dynamic query analysis overhead not present in the Hand-coded version, their response times are about equal. This shows the effectiveness of caching of the query analysis results.

Figure 4.6 shows the overhead of extra queries for the AC-extraQuery strategy, while running with 1000 clients. Response time for each request can be broken up into the cost of issuing of additional queries and the remaining time for the response. In this graph, the cost of issuing extra queries for each request is plotted above the remaining response time for that request. From the figure, we see that the cost of additional queries is a significant portion of the response time. However, this cost is compensated by the accompanying improvements in hits to the cache.

4.2.2 TPC-W

In Figure 4.7 and Figure 4.8 I show the results for TPC-W using the primary reporting mix of shopping. The cache hit rates are shown in Figure 4.7 and the response times in Figure 4.8 (please note the log scale of the y-axis).

From Figure 4.8, as expected, the No Cache version has a considerably higher response time than the caching versions. However, the three automatic caching versions perform nearly identically (the three lines are nearly on top of one another in the graph) despite the considerable differences in hit rates (24% up to 47%). In addition, the Hand-coded cache is considerably faster than the automatic cache versions, but its cache hit rate is only slightly higher than that of AC-extraQuery. Again, I can explain this behavior from the breakdown
Figure 4.7: Cache Hit Rates for TPC-W Shopping Mix

Figure 4.8: Response Times for TPC-W Shopping Mix
Figure 4.9: Breakdown of Hits by Request Type in TPC-W

of the request types and hit ratios and response times, shown in Figures 4.9 and 4.10.

The simplest automatic cache configuration AC-columnOnly effectively caches the execute search and new products web pages, with the bulk of the performance improvement coming from caching execute search web pages. The AC-valueBased strategy improves the hit rate of the cache with the product detail requests, but the response time is not noticeably affected because these requests, which are relatively fast, make up less than 2% of the overall average response time. The most aggressive automatic cache design, AC-extraQuery, helps improve the hit rate of the new products web pages somewhat; however, the response time is not noticeably affected because this request is also relatively cheap. Requests home interaction and search request are uncachable because they display promotional items based on random numbers and correspond to the hidden state problem. Requests order display and order inquiry have primarily cold misses.

Of potential concern is the fact that the additional queries generated by the AC-extraQuery cache to the heavily loaded database could actually reduce overall performance. I do not
Figure 4.10: Breakdown of Overall vs Miss Response Time in TPC-W

see this effect because less than 0.5% additional queries are generated for this benchmark for AC-extraQuery. I show the overhead of extra queries for this caching strategy while running with 400 clients in Figure 4.11, where the cost of extra queries for each request is plotted above the remaining response time for that request.

In the figure we see that the overhead of extra queries for admin response is small as compared to the remaining response time. For the request types buy request, buy confirm and cart interaction, the overhead becomes comparable to the response time.

My version of TPC-W implements the benchmark’s optional optimization that allows a temporary table needed for best seller processing to be cached without regard to consistency and generated only once per 30 seconds. The primary improvement in the Hand-coded cache comes from my determination via manual inspection that the best seller pages remain valid, and thus much more cacheable, for the full 30 second window. Since the best seller requests are expensive, caching these pages is a significant performance advantage. However, the temporary table is actually an instance of hidden
state and, thus, it is uncachable for any of the automatic caching versions.

4.2.3 Web Page Cache Size Requirement

In all the experiments, to effectively measure the cache hits for each invalidation strategy, I do not place any restriction on the size of cache. However, I present an estimate below for the size of cache required for each application.

In each case, let \( d \) be the total size of cache entries accessed between two successive fetches for the same entry (also known as the cache reuse distance). I plot a cumulative frequency distribution of \( d \) measured for all cache accesses. This plot gives us the ratio of cache hits obtained with a particular cache size to those obtained with an infinite sized cache, for different cache sizes.
RUBiS

Figure 4.12 shows that 90% of cache hits require a cache of size around 10MB. Most of the hits, however, can be achieved by having a cache size of 100MB.

TPCW

Figure 4.13 shows that a cache of size around 5MB is sufficient for 90% of the cache hits. A cache of size 20MB, on the other hand, can achieve most of the hits.

4.2.4 Query Analysis Caching

The query analysis cache stabilizes very quickly. I show the query analysis cache statistics in Table 4.1. For RUBiS, the total number of entries (unique SQL strings encountered) are 22 read query types and 10 write types. All query types were recorded in the first 4 minutes of simulation at 1000 clients. Similarly for TPCW, there are 10 read types and 14 write types and they all occur within the first minute of activity.

To measure the overhead of performing the query analysis, I turned off the query analysis cache in the AC-extraQuery configuration so that every web page cache miss would
Table 4.1: Query analysis cache statistics for RUBiS @ 1000 clients and TPC-W @ 400 clients

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reads</th>
<th>Writes</th>
<th>Time to stabilize</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUBiS</td>
<td>22 types</td>
<td>10 types</td>
<td>4 min</td>
<td>2.8x</td>
</tr>
<tr>
<td>TPC-W</td>
<td>10 types</td>
<td>14 types</td>
<td>1 min</td>
<td>4.0x</td>
</tr>
</tbody>
</table>

incur the analysis overhead. Therefore, for each cache miss, all the queries occurring in the request are analyzed for dependencies with other queries. In other words, the query analysis cache always returns a miss.

The average response time for TPC-W experienced a factor of 4.0 increase in response time. The overhead of RUBiS increased by a factor of 2.8. This clearly demonstrates the importance of caching the analysis results in a run-time system such as mine. Since all analysis is performed dynamically on the fly, it is important to reuse the computation already performed. Also, as we have seen, the analysis tables get stabilized quickly, making any overheads of dynamic analysis over a period of time insignificant.
Chapter 5

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 [17, 11, 16]. Alternatively, in dynamic web page caching whole pages or page fragments (either in HTML or in XML) are cached [6]. The two approaches are complementary and can be combined. This thesis focuses solely on dynamic web page caching. Our work distinguishes itself from other work on dynamic web page caching in that I provide both consistent and transparent caching. Previous work in this area has only provided one or the other of these two qualities. Among these works, I single out as representative examples CachePortal [2] (transparent but not consistent) and DynamicWeb [7] (consistent but not transparent).

The goal of CachePortal is to implement caching without modifying the underlying application code. This non-invasiveness is achieved by using the run-time logs to relate HTTP requests to database queries, grouped by (wall clock) timestamp [2] intervals. Further on-line processing narrows the association between cache pages and invalidates within a group. In addition, an off-line mode permits domain experts to manually specify relationships between request types to improve caching and invalidation. Thus, CachePortal is transparent to the application, but not the developer. In contrast, I achieve transparency to the developer via AspectJ to automatically make the necessary modifications to the application. For invalidations, CachePortal provides time-lagged consistency via polling, while
our invalidations are immediate. Like my design, extra queries can be generated to improve the selectiveness of invalidates.

DynamicWeb provides an API for specifying a dependency graph between certain events, in particular, between write queries and cached web pages [7]. The occurrence of said events triggers invalidations immediately. Thus, Dynamic WebCache provides consistency, but at the expense of considerable programmer effort. In my experience with the hand-coded caching system, manually locating all dependencies even in the relatively simple RUBiS benchmark (25 servlets and 4,600 lines of code) was challenging.

Other non-transparent approaches include Weave [18] that requires the use of a specialized language to describe dynamic web pages and event handlers for specifying invalidations, and various commercial solutions such as SpiderCache [15] and XCache [16], both of which provide an event API. In addition, these systems typically support periodic updates and therefore provide only time-lagged consistency.

The caching system presented in [4] for IBM's Websphere Accessible Business Rule framework relies on a general purpose cache library API to which developers manually insert calls. However, their DUP system performs static analysis to generate object dependence graphs to track interdependencies for invalidates. My work differs in three aspects. The first difference is that I improve the preciseness of invalidates by automatically generating additional database queries. This method can be instrumental in reducing the actual overall response times, as was seen with the RUBiS benchmark. Secondly, our use of AspectJ eliminated the need for static code analysis and let us focus on the query analysis directly. While the analysis has to be done at run-time, I demonstrated that the analysis results can be effectively cached. And thirdly, I focus on response time performance, not the hit rate of the cache only. In using an actual prototype instead of a synthetic workload generator, I discovered that not all request types are equal and performance improvements
come from aggressively reducing unnecessary invalidates of frequent, high cost requests.

My work is orthogonal to issues of caching granularity. Partial page caching can be implemented using ESI or similar techniques [6]. Our analysis method can be extended to insert, check, and invalidate a cache containing page fragments defined by ESI or similar annotations.
Chapter 6

Conclusions and Future Work

I demonstrated that it is possible to have consistent and transparent caching of dynamic web pages by automatic injection of caching logic on the server-side application code. To the best of my knowledge, this is the first system that provides both consistency and complete transparency. I explored a range of invalidation strategies which were strictly progressive in their precision of determining invalid pages. I discovered that these strategies are progressively effective in caching different request types. Finally, I showed how the runtime overheads of my system are made negligible through effective caching of the analysis results.

I noted that performance gains come from being able to cache select requests that dominate the overall average response time, although in my work I do not actually try to cache these over other requests. I also observed that hit rate alone is not an adequate predictor for performance and that the benefits of caching are different for different request types. In addition, being able to draw on additional information by querying the database during invalidation checks (i.e., extra queries) proved beneficial. The overhead of extra queries was not a factor in our benchmarks. However, in certain cases where the database is heavily loaded, issuing extra queries might prove detrimental to overall performance. To improve performance, the overhead of these extra queries must be compensated by gains obtained through more cache hits.

I use AspectJ to insert the cache interface calls. AspectJ is an implementation of aspect-oriented programming (AOP) for Java. It provides us with a neat interface to automatically
inject the caching logic into the application, without any help from the application developer. A disadvantage is that no static analysis is done to pre-process the queries in the application, all analysis must be performed at run-time. While the run-time analysis is expensive, in this environment the analysis results can be efficiently cached to make the overhead negligible.

This work presents itself several avenues for extension. A query cache storing results of queries made to the database would involve similar run-time analysis techniques as the web page cache and can be inserted between the application and the database server. A query cache to supplement the web page cache would offer an interesting trade-off in terms of memory requirements, granularity of invalidations and response time benefits from each.

We have observed that the benefits of caching do not apply equally to all request types. A natural extension of this work is to provide automatic run-time feedback to the caching system, to avoid caching of requests that do not benefit from caching.

In this system, I do not place any restriction on the cache size. Another practical dimension is to keep cache size fixed and investigate various cache replacement strategies. An intelligent scheme taking into account various factors like frequency of access, cost of response generation and size of cached responses would offer the highest benefits.

The present system endeavors to keep the cache consistent with the state of the database. Such strong consistency might not be desired in all scenarios. Exploring response time gains while relaxing the cache consistency model will provide an insight into the trade-offs involved.

Finally, my model assumes existence of one application server and one database server. When there is more than a single database server, the cache needs to be invalidated depending on the updates to all these databases. But as long as there is one application server, all requests go through it and a single cache works fine. When we have more than one appli-
cation server, however, the choice is between keeping a single cache managed together by all the servers and maintaining a cache on each of the servers. Having a single cache can result in the cache being the bottleneck itself, defeating the purpose of having a multitude of application servers in the first place. Providing a cache on each server poses the problem of maintaining coherency among these caches. This coherency problem is akin to the problem of cache coherency for a multi-processor system. It would be interesting to determine, however, whether keeping multiple caches in my system incurs additional consistency and performance issues.
Bibliography


