AOP-Based Caching of Dynamic Web Content: Experience with J2EE Applications

Sara Bouchenak, Alan Cox, Steven Dropsho, Sumit Mittal, Willy Zwaenepoel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

AOP-Based Caching of Dynamic Web Content: Experience with J2EE Applications

Sara Bouchenak, Alan Cox, Steven Dropsho, Sumit Mittal, Willy Zwaenepoel

N° 5483

February 2005

Thème COM

ISSN 0249-6399

AOP-Based Caching of Dynamic Web Content: Experience with J2EE Applications

Sara Bouchenak[*], Alan Cox[†], Steven Dropsho[‡], Sumit Mittal†, Willy Zwaenepoel‡

Thème COM – Systèmes communicants
Projet Sardes

Rapport de recherche n° 5483 – February 2005 - 35 pages

Abstract: Caching dynamic web content is an appealing approach to reduce Internet latency and server load. In aspect-oriented programming, caching is usually presented as an orthogonal aspect that could be automatically integrated to an application. A classical AOP motivating example is adding caching of static data with no underlying consistency. But what about caching dynamic data? In this paper, we explore the feasibility of *aspectizing consistent* caching of dynamically generated web documents. We use two J2EE web applications to validate our experiments: the TPC-W on-line bookstore and the RUBiS auction site. To the question "Can we consider consistent caching of dynamic web content as a separate aspect that could be transparently and efficiently integrated to a dynamic web application?", our conclusions are the following: (a) Just as in the classic AOP caching example having no consistency management, AOP provides a modular way to add caching having a *strong* consistency policy. (b) However, maintaining strong consistency on web pages results in prohibitively expensive run-time processing and, thus, any straightforward implementation in AOP is too slow. We propose an optimization that essentially eliminates *all* the run-time overhead in practice. (c) Furthermore, we identify instances where consistent web caching may not be orthogonal to J2EE applications, especially for those applications that rely on sophisticated web techniques (e.g., cookies). In summary, adding caching supporting strong consistency using AOP turned out to be an unexpected challenge.

Keywords: Caching, aspect-oriented programming, J2EE applications, dynamic content.

[*] Grenoble I University, Department of Computer Science, Grenoble, France, *Sara.Bouchenak@inria.fr*
[†] Rice University, Department of Computer Science, Houston, TX, *{mittal, alc}@cs.rice.edu*
[‡] EPFL, Department of Computer Science, Lausanne, Switzerland, *{Steven.Dropsho, Willy.Zwaenepoel}@epfl.ch*

Caches Web Dynamiques Basés sur la Programmation par Aspects : Expérimentation avec les Applications J2EE

Résumé: L'utilisation de techniques de cache de documents web dynamiques est une approche intéressante pour réduire la latence du réseau et la charge des serveurs. Dans le domaine de la programmation par aspects, le cache est traditionnellement présenté comme un aspect orthogonal qui pourrait être automatiquement intégré à une application. Un des exemples classiques de la programmation par aspects est l'ajout d'un cache de documents statiques sans gestion de la cohérence du cache. Dans cet article, nous étudions la possibilité d'*aspectiser* un cache *cohérent* de documents web générés dynamiquement. Pour valider nos expérimentations, nous avons utilisé deux applications web J2EE : l'application TPC-W de librairie en ligne et l'application RUBiS de vente aux enchères. A la question « Peut-on considérer la gestion cohérente d'un cache de documents web dynamiques comme un aspect orthogonal qui peut être intégré à une application web dynamique de manière transparente et efficace ? », notre réponse est la suivante : (a) Comme dans le cas classique de cache de documents statiques sans gestion de la cohérence, la programmation par aspect fournit un moyen modulaire pour la prise en compte d'un cache cohérent de documents dynamiques. (b) Cependant, une mise en œuvre naïve d'un cache cohérent basée sur la programmation par aspects peut résulter en une solution inefficace ; nous avons ainsi proposé une mise en œuvre optimisée qui élimine tout éventuel surcoût à l'exécution. (c) Par ailleurs, nous avons identifié des cas d'applications web J2EE où la gestion cohérente d'un cache de documents dynamiques n'est pas orthogonale à l'application, telles que les applications utilisant des techniques web sophistiquées (ex. cookies). Finalement, la prise en compte d'un cache cohérent de documents web dynamique en utilisant la programmation par aspect s'est avérée ne pas être triviale.

Mots clés: Cache, programmation par aspects, applications J2EE, documents dynamiques.

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 magnitudes slower than static documents [10, 12]. Therefore, caching dynamic web documents is an appealing approach to reduce Internet latency and server load. Web sites for dynamic content are usually based on a multi-tier J2EE architecture implemented using several middleware systems [31]: 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 generate the dynamic web content, and a database to store the non-ephemeral data required by the application. 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.
Caching is usually presented as a feature that is orthogonal to applications functionality, added to improve applications performance. Moreover, the process of adding caching to an application can be more or less *transparent*, in the sense that it does not require a manual change to the application nor a help from the application programmer. In aspect-oriented programming (AOP), caching is a classical example of an orthogonal aspect that can be automatically woven to an application. Some simple (academic) examples clearly explain how to use AOP to automatically add caching of static, i.e., unchanging, data to an application, a caching solution that ignores consistency [18, 6].

In this paper, we relate our experience in using AOP to add *consistent caching* to more complex and realistic applications: dynamic J2EE web applications (i.e., applications that dynamically generate web content). Indeed, we explore the feasibility of *aspectizing consistent* caching of dynamically generated web pages. To this end, we use AspectJ [2] to implement an AOP-based web caching system for dynamic J2EE applications, and we use two J2EE web applications to validate our experiments, the TPC-W on-line bookstore [33] and the RUBiS auction site [1].

The motivation of our work is to address the following question: can we consider consistent caching of dynamic web content as a separate aspect that could be automatically integrated to dynamic web applications? Our conclusions are the following:

a) AOP provides a modular way to add consistent caching to J2EE web applications, through code factorization and clean separation of concerns.
b) However, maintaining strong consistency on web pages results in prohibitively expensive and repeated run-time analysis and, thus, any straightforward implementation of strongly consistent web caching in AOP is too slow. We propose an optimization that essentially eliminates *all* the run-time overhead in practice.
c) Unlike caching static data (with no consistency management), consistent caching of dynamic web content may be a *non-orthogonal* aspect for some J2EE applications, especially for those applications that rely on sophisticated web techniques (e.g., cookies [23], and randomly generated advertisement banners [27]).

In summary, caching is usually presented as being orthogonal to applications (i.e., could be *aspectized*); this may be true for simple cached data and/or with no cache consistency management. In this paper we show that in the case of consistent caching of dynamic web content, *complete aspectization* of caching is simply not possible.
The remainder of the paper is organized as follows. Section 2 presents background information on J2EE web applications. Section 3 and Section 4 describe the design and implementation details of respectively consistent web caching and transparent (i.e., AOP-based) web caching. Section 5 and Section 6 respectively present the results of the experimental evaluation of the

AOP-based consistent web caching, and the lessons learned from our experience. Section 7 describes the related work and finally, Section 8 draws our conclusions.

2 Background on J2EE web applications

Java 2 Platform, Enterprise Edition (J2EE) defines a model for developing distributed applications, e.g., web applications, in a multi-tiered architecture [29]. Such applications usually start with requests from web clients that flow through an HTTP server front-end and provider of static content, then to an application server to execute the business logic of the application and generate web pages on-the-fly, and finally to a database that stores resources and data (see Figure 1).

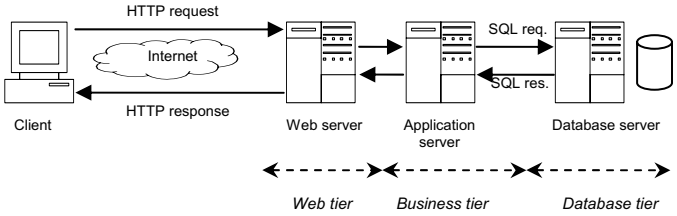


Figure 1. Architecture of dynamic web applications

Upon an HTTP client request, either the request targets a static web document, in which case the web server directly returns that document to the client; or the request refers to a dynamic document, in which case the web server forwards that request to the application server. When the application server receives a request, it runs one or more software components (e.g., Servlets, EJB) that query a database through a JDBC driver (Java DataBase Connection driver) [30]. Finally, the resulting information is used to generate a web document on-the-fly that is returned to the web client. Figure 2 gives an example of the class of a Servlet software component with its main program.

```
public class MyServletClass extends HttpServlet {
  public void doGet(HttpRequest  request, HttpResponse response)
    throws IOException, ServletException {

    // Execute SQL queries on the back-end database
    …

    // Generate a web document
     …

    // Return the web document to the client
    …
  }
  …
}
```

Figure 2. Example of a Servlet software component

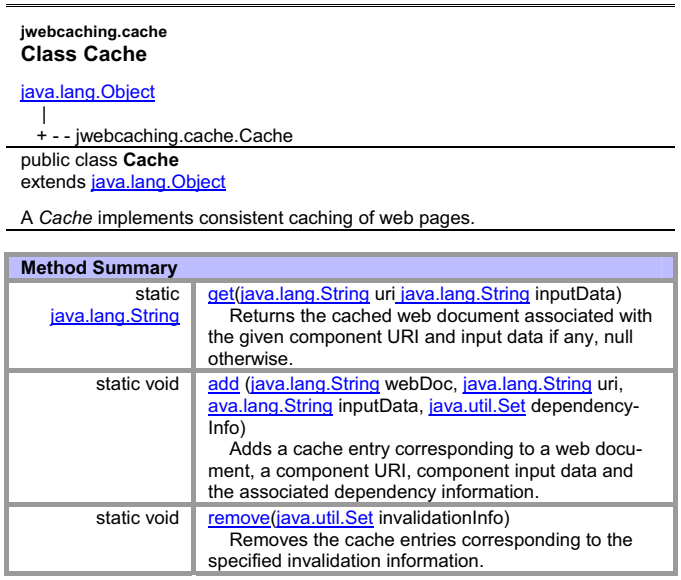
3 Consistent web caching

Caching is a classical technique that has been studied for server-side acceleration of web sites. Nowadays, dynamically generated web content represents a large portion of web requests, and the rate at which dynamic documents are delivered is often one or two orders of magnitudes slower than static documents [10, 12]. In this context, caching dynamic web documents is an appealing approach to reduce Internet latency and server load. Indeed, if caching static web content prevents the web client from remotely re-accessing the web server to fetch static web pages, caching dynamic web content prevents the client from remotely re-accessing the database server to re-execute SQL queries, and from regenerating dynamic web pages on the application server.

But the use of caching is rendered more complicated by the dynamic nature of web applications, requiring mechanisms to maintain consistency between the data and its cached copy. In brief, a caching system of dynamic web content involves the following mechanisms: cache checks and inserts, cache invalidations and collecting information to manage cache consistency. Upon a client request, the cache is first checked to look up the requested page. In case of a cache hit, the cached page is simply returned to the client. Otherwise in case of a cache miss, the application server dynamically generates a web page that is returned to the client, and a copy of that page is stored in the cache. In order to maintain cache consistency, a web page that has been cached must be invalidated if a client request performs a write to the database that results in modifying the data set used to generate the content of the now cached page. Therefore, appropriate consistency information must be collected to perform cache invalidations.

3.1 Overview of the JWebCaching system

Our experiments make use of *JWebCaching*, a Java library that we provide for caching web pages and managing their consistency [3]. In JWebCaching, the cache is located on (in front of) the application server, and it consists of a set of web pages that result from client read-only requests (i.e., requests that do not generate write SQL queries to the database). The goal is to avoid re-execution of read-only requests with the same set of arguments, and return the results from the cache instead. In other words, two client read-only requests to the same URI + set of arguments result in the same web document, until a client write request that modifies the content of that document is performed (i.e., a client request that involves write SQL queries to the database that result in modifying the data set used to generate the content of the cached page). JWebCaching uses SQL query analysis techniques to determine if a write query *intersects* with a read query, and thus to deduce if a client write request invalidates the result of a client read-only request. Moreover, JWebCaching provides various invalidation strategies depending on the level of precision for detecting write and read query intersection. In our experiments, we use the most efficient cache invalidation strategy provided by JWebCaching, namely the *AC-extraQuery* strategy (see [3] for further details). On the other hand, the web pages resulting from client write requests if any are not cached, since they need to be re-computed on each subsequent request (the underlying write SQL queries need to be re-executed).



jwebcaching.cache
Class Cache

java.lang.Object
|
+ - - jwebcaching.cache.Cache

public class **Cache**
extends java.lang.Object

A *Cache* implements consistent caching of web pages.

Method Summary	
static java.lang.String	get(java.lang.String uri java.lang.String inputData) Returns the cached web document associated with the given component URI and input data if any, null otherwise.
static void	add (java.lang.String webDoc, java.lang.String uri, ava.lang.String inputData, java.util.Set dependency-Info) Adds a cache entry corresponding to a web document, a component URI, component input data and the associated dependency information.
static void	remove(java.util.Set invalidationInfo) Removes the cache entries corresponding to the specified invalidation information.

Figure 3. Cache API

The main package of the *JWebCaching* library is the *jwebcaching.cache* package which provides several classes, among which the *Cache* class that provides the necessary features for cache management. Figure 3 illustrates a part of the API of the *Cache* class. The *get* method looks up an entry in the cache associated with a given URI and input data (i.e., arguments). In case of a cache hit, the corresponding cached web page is returned as a result of the method. In case of a cache miss, the *add* method can be called to add a new web page to the cache, associated with a given URI, input data and a set of dependency information used to determine when that page needs to be invalidated (see section 4 for further detail). Finally, upon a client write request and in order to ensure cache consistency, some cache entries may be removed, (i.e., invalidated) using the *remove* method.

3.2 Cache consistency

The basic cache operations to look-up pages, add pages, and remove pages from the cache can be efficiently implemented with obvious data structures and their details will not be discussed here. Complications arise when maintaining cache consistency. We provide a sufficient discussion here for the reader to appreciate the computational effort involved. More details can be found in [3].

Determining if a client write request invalidates the cached page resulting from a previous client read-only request is equivalent to determining if the set of SQL queries associated with the former request invalidates one of the SQL queries underlying the latter request. For this purpose, the JWebCaching system includes a query analysis engine has the task of determining the dependencies between SQL queries for efficient discovery of web pages affected by an update. This analysis is performed on queries automatically during run-time. Query analysis has, more precisely, three primary components: query string parsing, determining possible dependencies between queries, and actual intersection testing to reveal true dependencies.

Query string parsing. SQL queries are given as a string with a vector of dynamic values to be inserted for the particular instance. Parsing separates a query into the tables and columns being referenced or used in selection criteria of the WHERE-clause. The names of the database tables and columns are saved in hash tables for each category of data: SELECT-, FROM-, and WHERE-clauses. Dynamic values in the WHERE-clause are associated in the hash table to the column names being conditioned, e.g., customer_id=7097. This processed format is stored and

cross referenced with its associated web page for fast look-up. Parsing is done once per request so it adds O(1) overhead per request.

Dependence testing. Consistency conflicts occur between update queries and the read-only queries used to create the cached web pages. Whenever an update occurs, the SQL queries for the update must be compared to all cached page queries to test table, column, and ultimately row dependencies indicating a true intersection. A straightforward implementation iterates across an enumeration of the cached web pages comparing the names looking for possible shared tables and columns, in that order. Comparing tables first eliminates quickly from consideration pages that have no possibility of intersecting. For an intersection to be possible, the columns being updated must match some columns in the web page's SELECT- or WHERE-clauses. What remains after the first scan is the set of possible web page candidates that may be affected by the update because of table and column dependencies. This method has complexity O(*N*), where *N* is the number of cached web pages and may number in the thousands.

Intersection testing. To this remaining set of pages, final testing is applied to each to determine if the actual row(s) being modified by the update are included in the web page's. If queries share common tables, columns, and rows with the entries being updated then they intersect. For true intersection, one of three possible scenarios must exist:

1. A web page element changes if the column being updated is in the web page's SELECT-clause(s) AND the modified row satisfies the web page's WHERE-clause(s).
2. A web page loses an element if the row prior to being updated satisfied the web page's WHERE-clause(s), but the row no longer does with the new(i.e., modified) value.
3. Conversely, the web page gains an element if the row prior to being updated did not satisfy the web page's WHERE-clause(s), but with the new value the row now does.

The brute force method of plugging and chugging with updated row's values into the WHERE-clauses of a cached page on an update is fairly quick to test for any of the above three conditions. The primary impact on performance is the number of possible pages that need to be considered, which is still O(*N*), where *N* is the number of cached web pages.

3.3 Cache optimizations

Dependence analysis. The simple consistency scheme is O(*N*) and does not scale, but we can do better. The dependence analysis is based on the static portion of the query string; the dynamic values are only needed for intersection testing. Since only the static portion of the query string is needed, repeated queries that differ in dynamic values (*e.g.*, the customer id) will have the same query analysis result (i.e., query template). Thus, the analysis results can be re-used with a new analysis occurring only when a never-before-seen query arrives. In practice, there are usually a small fixed number of different query templates. In our testbed applications, TPC-W has a total of 24 unique queries (different templates) and RUBiS has a total of 32 (c.f., section 5.1).

Let us compare with a compiler-based approach as in [8]. The compiler does a similar analysis at compile time and embeds the results for simple look-up at run-time. The proposed JWeb-Caching solution also achieves zero run-time analysis overhead through result caching, but is much easier to develop than compiler tools since it uses AOP tools (c.f., section 4). Another subtle advantage of the proposed approach is that it is robust even if the SQL queries are dynamically formed, since it uses the run-time value of the string at the point of the SQL call. For a compiler, query strings must be statically available.

Intersection testing. Intersection testing requires explicit testing of the WHERE-clauses of each web page having a dependence with an update query. For general evaluation functions that can be used in the WHERE-clauses, an O(*N*) linear scan to evaluate the conditions of each page is necessary. However, for simple, but common functions a technique with O(1) complexity can be used. The technique tackles web applications where almost all of the conditionals are tests for equality of one or two fields, *e.g.*,

```
WHERE item.author_id = author.id AND item.subject = ?
```

The dynamic values (here, the value for '?') are used as a hash table index. If a hash table look up is null then the particular values do not satisfy the WHERE-clause of any page in the cache and no invalidates are needed. If the look-up is non-null, what is returned is the web page whose values match. This page is invalidated. Other simple conditionals can also be handled in O(1) access time with variations on the above scheme. Note that this testing overhead occurs at run-time in both the AOP and compiler solutions.

4 Aspectizing web caching

In order to explore the feasibility of *aspectizing* consistent caching in the context of dynamic web applications, we designed and implemented AutoWebCache. AutoWebCache was built to answer the following question: can we consider consistent caching of dynamic web content as a separate aspect that could be automatically integrated to (almost) any dynamic web application? In the following, we first recall the main concepts of AOP before describing the design and implementation details of an AOP-based web caching system.

4.1 AOP and AspectJ

Aspect-Oriented Programming (AOP) is a methodology with concepts and constructs to modularize crosscutting concerns (i.e., *aspects*) [16]. With AOP, the different aspects involved in a system are separately implemented in different modules. To form the final system, AOP provides a way to specify the relationship between the aspects, i.e., the way the different modules need to be woven.

Aspects are woven together via the join point model, a fundamental concept in AOP. It specifies identifiable execution points in a system, e.g., method call, object construction. The join points are the places where enhancements may be added by injecting crosscutting actions.

A realization of the AOP methodology consists of a programming language and a set of tools to work with that language. AspectJ is an aspect-oriented environment that provides the AOP language and set of tools for aspects written in the Java programming language [2].

The AspectJ language exposes a set of join points that are well-defined places in the execution of a Java program flow. Such join points include method calls and executions, constructor calls and executions, read and write access to fields, exception handler executions. *Pointcuts* allow a programmer to capture certain join points while an *advice* provides a way to express crosscutting actions to be performed at a certain pointcut. At a pointcut, an advice specifies the weaving rules involving that point, such as performing some actions before or after the execution of the pointcut.

```
(a)  aspect ServletExecution {
(b)    // Pointcut definition
(c)    pointcut doGetExecution()  :
(d)        execution(
(e)              void HttpServlet+.doGet(
(f)                    HttpServletRequest, HttpServletResponse)) ;
(g)    // Advice definition
(h)    before() : doGetExecution() { ... crosscutting actions   ...}
(i)  }
```

Figure 4. Pointcut and advice examples

Figure 4 gives an example of a pointcut and advice declaration in the AspectJ language. This example defines a pointcut called *doGetExecution* that designates the execution of the *doGet* method in the *HttpServlet* class or its subclasses[4] that takes a first argument of type *HttpServletRequest* and a second argument of type *HttpServletResponse* (lines (c) to (f) in Figure 4). This example also defines an advice that executes prior to the specified pointcut, i.e., prior to the execution of any *doGet* method in the *HttpServlet* class or its subclasses that takes a first argument of type *HttpServletRequest* and a second argument of type *HttpServletResponse* (line (h) in Figure 4).

It is important to notice that the pointcuts and advices that define the weaving rules to be applied are specified as separate entities from the individual aspect modules. Aspect weaving,

[4] The + sign following the *HttpServlet* class name in Figure 4 designates subclasses of that class.

i.e., the process of composing the final system from individual aspects by following the weaving rules, is performed by the *ajc* tool, the AspectJ compiler. This is usually performed through a Java binary-to-binary translation.

4.2 Design principles of AOP-based caching

When designing AutoWebCache, our initial objectives were to provide a consistent web caching solution for dynamic J2EE applications with the following transparency properties:

- ¿ No modification to the client or to the middleware platforms of the multi-tier J2EE architecture (e.g., web server, application server, database server).
- ¿ No modification to the J2EE application code.
- ¿ No use of application semantics information, no help from the application programmer.

From a conceptual point of view, introducing caching to a dynamic web application involves the following changes:

Cache checks. Upon a client read request, the cache is first checked to look up the requested page. In case of a cache hit, the cached page is simply returned to the client and the execution of the request is bypassed.

Cache inserts. Upon a client read request and in case of a cache miss, the read request is executed by the application server (and SQL queries are possibly executed on the database server) to dynamically generate a web page that is returned to the client; and a copy of that page is stored in the cache.

Whereas cache check and insert operations are classically used in any caching system (for caching static or dynamic data), consistent caching of dynamically generated web content involves the following additional mechanisms:

Cache invalidations. Upon a client write request, the cache entries that may be affected by that write request must be invalidated.

Collecting consistency information. In order to ensure cache consistency, a method is needed to determine which cached web pages are affected by database updates generated by client write requests. This method should maintain, on the one hand, dependency information between cached web pages and the underlying data in the database, and on the other hand, invalidation information that determines which data are actually affected by a database update. Upon a change to underlying data, the invalidation information and dependency information are examined to determine which cached pages are affected and thus need to be invalidated. Dependency information is associated with cached pages and is collected from read-only SQL queries, and invalidation information is collected from write SQL queries[5].

Figure 5 and Figure 6 show how collecting dependency and invalidation information, and how cache check, insert and invalidation operations take place within web application request handlers.

Finally, in order to implement an AOP-based solution for consistently caching dynamic web pages in J2EE applications, the following properties are needed:

- ¿ The entry and exit points of request handlers in web applications must be well-known points. This is necessary to automatically inject cache check, insert and invalidation operations to those handlers.
- ¿ The call to SQL queries that underlie the request handlers in web applications must be well-known points. This is necessary to collect dependency and invalidation information.

[5] See [3] for further detail about collecting consistency, i.e., dependency and invalidation, information, and see [8,13] for similar techniques.

One might think that those assumptions violate the initial design objective regarding genericity: "*No use of application semantics information, no help from the application programmer*". But if the well-known points specified in those assumptions are defined in standard APIs that are of general use in J2EE web applications, and are not specific to one particular application, we may consider that the genericity objective remains valid.

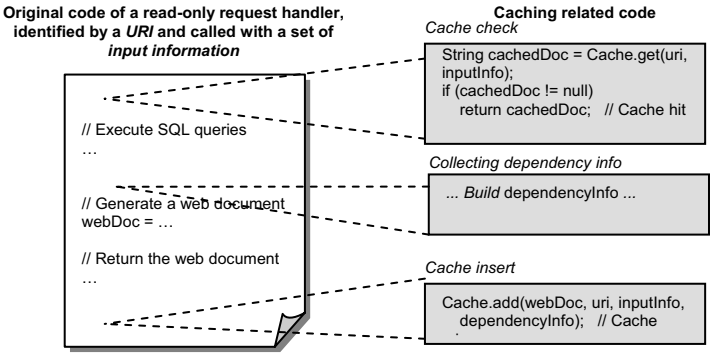


Figure 5. Caching and read requests

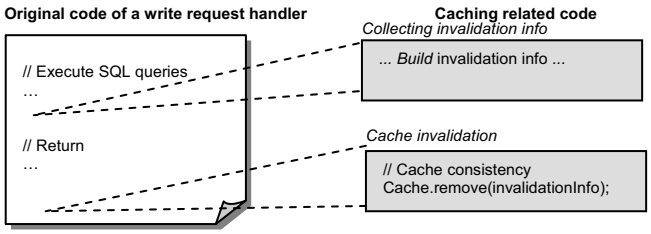


Figure 6. Caching and write requests

4.3 Implementation details

We implemented AutoWebCache as an AOP-based solution that provides a way to automatically inject consistent caching mechanisms to dynamic J2EE web applications. This involved the following steps:

- *Weaving rules specification*, which defines how to integrate the caching aspect into the web application core aspect in order to form the final system. The weaving rules specify the points in the web application where caching mechanisms need to be injected, e.g., points where cache check, insert or invalidation operations are called (c.f., Figure 5 and Figure 6).
- *Aspect weaving*, which is the process of composing the final cache-enabled system from individual J2EE web application and JWebCaching aspects by following the weaving rules. This is performed by the AOP compiler (see Figure 7).

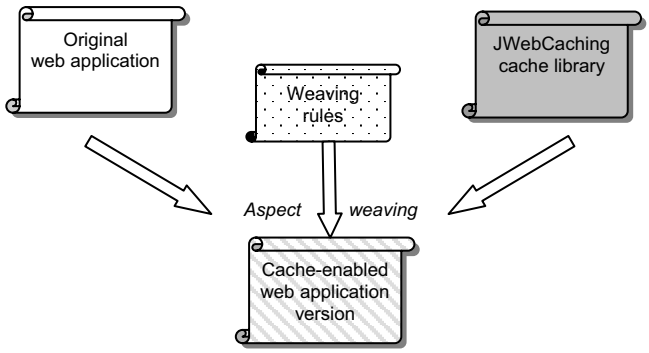


Figure 7. Aspectizing caching

The current prototype of AutoWebCache is implemented as a generic solution for any J2EE web-centric application, i.e., a web application that uses Servlets embedding SQL queries based on JDBC [31]; since this pattern is widely used in many J2EE applications [5]. In the following, we briefly describe the main implementation details of AutoWebCache using AspectJ.
Figure 8 shows how to capture the execution of a Servlet's main method in AspectJ; this is necessary to inject cache-related pre- and post-processing operations: cache checks, inserts and invalidations. Since Java Servlets are defined with a standard API, their main methods are known as being either *doGet* or *doPost* that respectively implement HTTP GET and POST; and the AspectJ's *execution* keyword used in the pointcut captures the execution of those methods..

```
// Pointcut for Servlets' main method
pointcut servletMainMethodExecution(...) :
    execution(
        void HttpServlet+.doGet(
            HttpServletRequest, HttpServletResponse))
    || execution(
        void HttpServlet+.doPost(
            HttpServletRequest, HttpServletResponse));
```

Figure 8. Capturing Servlets' main method[6]

Cache checks and inserts. In Figure 9, the *around* advice surrounds the normal execution of the main method of a Servlet with pre- and post-processing phases (the *proceed* keyword calls the normal execution of the method). The pre-processing phase performs a cache check operation and returns in case of a cache hit (i.e., the normal execution of the Servlet is bypassed). The post-processing phase adds an entry in the cache in case of a cache miss. Here, the pre- and post-processing phases make the necessary calls to JWebCaching. This advice is aimed at tackling read-only Servlets.

[6] Since a Servlet's *doGet* method may call the *doPost* method (and vice versa), it is necessary not to capture the execution of both methods but only the top-level one when they are interleaved. This can be achieved in AspectJ using a *cflowbelow* pointcut (see [18], Chapter 3). For simplicity purposes, we do not use it here.

```
// Advice for read-only requests
around(...) : servletMainMethodExecution (...) {
        // Pre-processing: Cache check
        String cachedDoc;
        cachedDoc = ... call Cache.get of JWebCaching
        if (cachedDoc != null)  {
          ... return cachedDoc
        }

        // Normal execution of the request
        proceed(...);

        // Post-processing: Cache insert
        ... call Cache.add of JWebCaching
}
```

Figure 9. Weaving rules for cache checks and inserts

Cache invalidations. Figure 10 describes an advice that is aimed at tackling write Servlets; it defines the *after* advice that executes following a Servlet main method. This specifies the post-processing that performs the necessary cache invalidations, based on calls to the JWebCaching library.

```
// Advice for write requests
after(...) : servletMainMethodExecution (...) {
        // Cache invalidation
        ... call Cache.remove of JWebCaching
}
```

Figure 10. Weaving rules for cache invalidations

Collecting consistency information. Figure 11 first declares a pointcut that captures calls to read-only and write SQL queries (through standard JDBC API calls, e.g., *executeQuery*, *executeUpdate*). The *after* advice executes following an SQL query and performs the necessary processing to collect the consistency, i.e., dependency or invalidation, information derived from that query. Since query analysis is performed at run-time, a non-negligible performance overhead may be induced, especially for aggressive invalidation protocols that require greater analysis for more precise information. This considerable analysis overhead was made negligible by caching query analysis results. Furthermore, since the cache consistency mechanism necessitates collecting the consistency information associated with all SQL queries executed within a Servlet context, AutoWebCache introduces a new data structure that gathers all that information. The introduction of this data structure, which is associated with a Servlet, is achieved in AspectJ using member introduction [18].

```
// Pointcut for SQL query calls
pointcut sqlQueryCall()  :
    call(ResultSet PreparedStatement.executeQuery())
    || call(int PreparedStatement.executeUpdate());
// Advice for SQL query calls
after() : sqlQueryCall () { ... collect consistency info   ...}
```

Figure 11. Collecting consistency information

Miscellaneous. We have tried to describe the main design choices and implementation details that underlie an AOP-based solution for consistently caching dynamic web content in J2EE applications. We did not focus on technical details that we briefly describe below[7]:

- ¿ In some J2EE web applications (e.g., TPC-W and RUBiS), the request handlers (e.g., Servlets) generate and return parts of the resulting web document throughout the execution of those handlers and not only at the end of their execution. That means that, in order for the caching system to capture the whole generated wed document and to cache it at the end of the execution of a request handler, the separately generated parts of the document should be buffered. Such a buffering mechanism is part of AutoWebCache.
- ¿ In order to manage cache consistency, the proposed caching system automatically associates with each cached web document information about the SQL queries executed to generate that document. This, of course, implies that only committed queries are taken into account and not the aborted ones.
- ¿ Since the standard Servlet API does not allow, at aspect weaving time, to differentiate between read-only and write Servlets, it is not possible to know to which Servlet classes advice of Figure 5 will apply and to which Servlet classes advice of Figure 6 will apply. That is why we built an advice that is a union of those to advices and that can check, at run-time, if the Servlet is read-only or write, and to apply the corresponding processing.
- ¿ The proposed caching solution is completely transparent when all database updates go through the server-side application. If this is not the case and if some updates are directly performed on the database, then transparency is difficult to achieve. A possible solution is to simply extend the caching system with an API similar to the ones provided by the DynamicWeb and Weave systems to allow an external entity to invalidate cache entries 11, 35]. This external entity could, for instance, work through database triggers.

[7] For further detail on those technical aspects, the reader is invited to refer to [3].

5 Performance evaluation

To validate our experiments, we used two J2EE benchmark applications to evaluate the performance improvement obtained when caching dynamic web content.

5.1 Testbed applications

We used two J2EE applications to test the work described here: the TPC-W on-line bookstore and the RUBiS auction site. 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.

TPC-W. TPC-W implements an on-line bookstore [33]. It defines 14 different interactions among which accessing a user home page, listing new products and best sellers, registering a new user, updating the shopping cart, ordering. TPC-W can be used from a web browser (see Figure 12) or with a benchmarking tool that emulates web client behavior and provides statistics (e.g., client response time). TPC-W specifies 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%. TPC-W contains 46 Java files and about 12K lines of code. It consists of 8 database tables that store items, authors, countries, customers, address, orders, order_line and credit_info. The database size is 350MB. In our experiments, we used an implementation of TPC-W originally proposed by the University of Wisconsin [19].

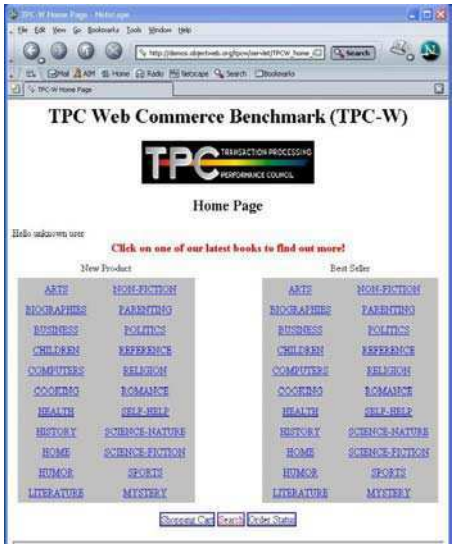


Figure 12. TPC-W application home page

RUBiS. RUBiS implements the core functionality of an auction site modeled over eBay [1]. It defines 26 interactions including registering new users, browsing items by category or region, bidding, buying or selling items, and leaving comments. RUBiS can be used from a web browser (see Figure 13) or with a benchmarking tool that emulates web client behavior and provides statistics. There are two mixes for RUBiS, a browsing mix that is 100% read-only and a bidding mix with 85% read-only interactions. The Servlet-based implementation of RUBiS that we used for our experiments contains 25 Java files and about 5.8K lines of code. It consists of 7 database tables that store users, items, bids, buy_now, comments, categories and regions. The total size of the database is 1.4GB.

Figure 13. RUBiS application home page

5.2 Experimental results

The AutoWebCache AOP-based caching system was finally applied to the TPC-W and RUBiS applications, automatically building a cache-enabled version of those applications (c.f., Figure 7). The corresponding experimental results are given below, and they are consistent in the fact that caching dynamic web pages clearly improves applications performance[8]. Figure 14 shows the response time in RUBiS, comparing the results of the evaluation of the cache-enabled version (AutoWebCache) with the original version of the application (No cache). The application was run with 50, 100, 300, 500, 600, 700, 800, 900, and 1000 clients. Here, RUBiS is running the bidding mix which has updates and, thus, generates cache invalidates to ensure cache consistency. For that mix, the cache hit rate is 54%. Here, caching provides a clear performance advantage, improving (i.e., reducing) response time by up to 64%.

Figure 15 shows the results for TPC-W using the primary reporting mix of shopping which has updates. Please note the log scale of the y-axis. From the graph, caching has significantly faster response times than the *No cache* version, as expected. Indeed, with caching the response time is reduced by up to 98%, and the cache hit rate is 43%.

[8] In these experiments, we use Apache v.1.3.22 as the Web server. The Servlet engine is Jakarta Tomcat v3.2.4, with the MM-MySQL v2.04 type 4 JDBC driver [21], running on Sun JDK 1.4.2. We use MySQL v.3.23.43-max [22] as our database server with the MyISAM tables. All machines run the 2.4.12 Linux kernel. Each machine has an Intel Xeon 2.4GHz CPU with 1GB ECC SDRAM, and a 120GB 7200pm 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. All machines are connected through a switched 1Gbps Ethernet LAN.

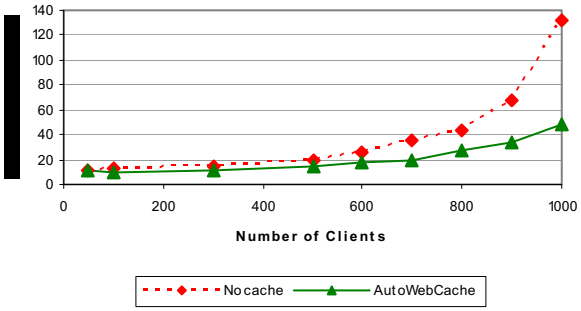


Figure 14. Response time for RUBiS – Bidding mix

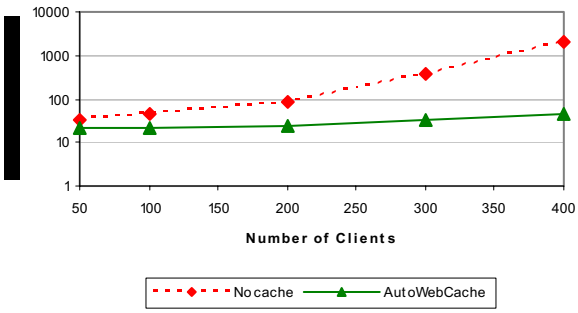


Figure 15. Response time for TPC-W – Shopping mix

6 Lessons learned

In this section, we present the lessons learned from our experience in using AOP for providing consistent caching of dynamic web content in J2EE applications.

Modularity. Thanks to the separation of concerns that is inherent to aspect-oriented programming, AOP-based systems benefit from greater modularity. Indeed, the implementation of each individual aspect (e.g., a J2EE web application and the caching of dynamic web content) may evolve separately without inducing a change in the implementation of the other aspects. Moreover, even if the design of the crosscutting caching aspect (i.e., the specification of its interfaces) changes, this modification usually implies a change in the weaving rules specification and not in the code of the other aspects (e.g., the web application). While in a non-AOP-based implementation of a cache-enabled version of a dynamic web application, a change in the interfaces of the caching library will imply changes in several places in the web application code where calls to those interfaces are made. Table 1 compares the code size of the individual aspects, the TPC-W and RUBiS testbed applications and the JWebCaching library, with the code size of the AOP-based AutoWebCache system which is much smaller and thus easier to maintain.

	Web application		Caching library		AOP-based caching	
Application	# Java classes	Java code size	# Java classes	Java code size	# AspectJ files (weaving rules)	Size of AspectJ code
TPC-W	46	12K lines	13	4.6K lines	1	150 lines
RUBiS	25	5.8K lines				

Table 1. Web application and caching library code size vs. AOP-based caching code size

Almost a generic solution. The AutoWebCache prototype shows that AOP provides a generic way to add consistent caching of dynamic web pages to a Servlet-based web application that interfaces a database with JDBC. To evaluate the genericity of AutoWebCache, two web applications were used, the TPC-W on-line bookstore and the RUBiS auction site, where consistent caching was introduced to each application with no necessary change to the applications code. More precisely, the cache-enabled version of those applications was obtained using the AspectJ compiler tool that automatically applies the caching weaving rules to the original Java binary code of the applications. Subsection titled "*Caching aspect may not be orthogonal to applications*" discusses the points that make an AOP-based caching solution not completely generic.

Complexity of consistently caching "derived data". The complexity of maintaining consistency is due to caching "derived data", e.g., dynamic web pages. We call such data derived because it is not identical to some piece of data in the persistent store of the application (e.g., rows in the database tables), but rather derived from it. When caching non-derived data, such as persistent data objects, those objects directly map to unique items in the persistent store, thus, checking for inclusion in the cache is a simple matter of checking for the existence of a unique global identifier (*e.g.*, an address). When caching data derived from base persistent data, as with web pages, the mapping relationship is obscured. In our implementation, more than one web page can depend on the same field in the database and multiple elements from the database are used to create the page, thus, detecting if a change to the database affects a web page involves complex testing for inclusion of the change in each page's input set. This is much more difficult than simple id look-up and is the source of inter-query dependence analysis and run-time query intersection testing.

Avoiding AOP's run-time overhead. The ease of using AspectJ comes at the cost of doing all SQL query dependence analysis at run-time, even though a static analysis could be performed

to avoid repeated re-computation at run-time [8]. However, in practice this overhead is eliminated by caching the analysis results and re-using them (c.f., 3.3). This technique is very effective and it is not surprising that it can be employed. Table 2 gives statistics on the time for the query analysis cache to stabilize with the RUBiS and TPC-W applications. All templates of SQL queries of RUBiS are analyzed within the first four minutes when there are 1000 clients. For TPC-W, the queries are analyzed within the first minute with 400 clients. In both applications, the analysis overhead occurs early during the warm-up period and so the analysis overhead is completely absent from the experimental run-time measurements.
The fact that static analysis can be applied means that the set of queries in the application is static (except for variable values inserted at run-time). Static analysis would fail if the SQL queries were formed dynamically; while an AOP-based solution would work.

Benchmark	Read queries	Write queries	Number Clients	Time to stabilize
RUBiS	22 types	10 types	1000	< 4 min
TPC-W	10 types	14 types	400	< 1 min

Table 2. Query analysis cache statistics for RUBiS and TPC-W

Caching aspect may not be orthogonal to applications. General caching has been incorrectly considered an orthogonal aspect. When time-lagged weak consistency is employed, then caching is orthogonal because once cached, entries are valid until some timeout constraint occurs. When using strong consistency of derived data then changes must be tracked on the data used to generate the cache items. This may result in a non-orthogonal caching aspect as we describe in the following. However, by specifying clear guidelines for developers to follow, an AOP-based solution dramatically reduces the manual effort. Violations of orthogonality between caching and the application may occur in the following cases:

¿ **Cookies.** A consistent cache of dynamic web content associates with each cached document the URI + set of parameters used upon the client request that generated that document. This allows the cache to uniquely identify cached documents. Therefore in order to automatically add caching to a dynamic web application, it is necessary to be able to automatically capture the URI and set of parameters upon a client request. The Java Servlet standard API provides a way to capture the URI and Servlet parameters that are specified in the HTTP client request; that is how AutoWebCache automatically captures that information from TPC-W and RUBiS applications without requiring any change to those applications. However, some web applications store part of their request parameters in cookies instead of implementing them as explicit parameters of the HTTP requests (e.g., the user name and password). In that case, in all requests to the server, the client includes its cookie [23]. A cookie is a small amount of state that can be viewed as a concatenation of strings (i.e., data) with no defined structure; thus, each web application defines its own ad-hoc cookie structure. Consequently, there is no standard way to capture parameters embedded in cookies upon client requests: the cache management can not be automated in that case.

  A possible solution is to make applications help the pseudo-automation of caching by providing a way to capture, in a generic way, request parameters embedded in cookies. For example, application Servlets might implement a specific pre-defined interface that allows the externalization, from the cookie, of the parameters used by the Servlet. The cache could therefore call that cookie externalization interface to capture the necessary parameters and associate them to the cached document. Of course, this solution does not provide full cache automation/transparency for the application. But it is an intermediate solution between a full automatic caching system and ad-hoc caching.

¿ **Randomly generated content.** Web documents are considered as dynamic because their content is based on data generated on-the-fly. An example of such data is the result of SQL queries performed on a backend database, as used in TPC-W and RUBiS applications. Another example of such data is randomly generated information, e.g., advertisement banners [27].
In web applications that generate dynamic web pages using both SQL query results and randomly generated data, the web pages can not be *automatically* cached in their whole form. Therefore, either a web application is well-structured, separating web pages that only depend on SQL query results from pages depending on randomly generated data (using for example some annotation techniques [9]), in which case the caching system could use that structure information to apply automatic caching when required. Otherwise, automatic caching can simply not be applied.

¿ **Complex application semantics.** For AOP to be used, the key semantic concepts must be conveyed via the syntax of the code and, therefore, must be rather straightforward. For instance, in the TPC-W application, the expensive Best Seller web interaction uses a 30 second window allowing dirty reads[9]. We implemented a hand-coded cache version that makes use of such application semantics information. The primary improvement in that hand-coded cache is that manual inspection of the application code allowed us to determine that the *best seller* pages remain valid, and thus cacheable, for the full 30 second window. This is an explicit deviation from the requirement of full consistency that cannot be extracted from the code. Note, without this performance optimization the behavior is still functionally correct, just slower, as we show in Figure 16 when compared to a hand-coded version of a cache having this optimization.

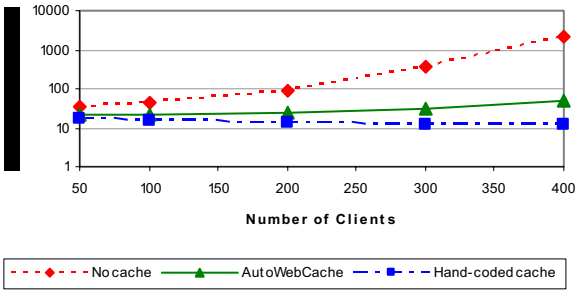


Figure 16. Cache improvement in TPC-W based on application semantics

[9] The effects of a change committed to the database by any web interaction which completed less than 30 seconds before the Best Seller web interaction was requested may not be reflected in the response page. This conforms to clauses 3.1.4.1 and 6.3.3.1 of the TPC-W v1.8 specification [33].

7 Related work

An ideal web caching solution would be both *completely transparent* and support *strong consistency*. Complete transparency means that no effort is required from the application programmer to achieve caching, without regard to application specific details; thus, such a cache is very easy to add. Support for strong consistency means the cache can ensure it is always synchronized with the state of the persistent backing store; thus, such a cache would have the widest audience. The related work discussed in this section attempts to achieve one or both of these goals.

Caching of static content achieves both goals because strong consistency is trivially achieved by the fact that the content does not change. Complete transparency is easily achieved because the final content is captured at well-known points, *i.e.*, when sent back to the client, and this does not vary between applications.

Caching of dynamic content with weak consistency achieves both goals as well because, as for static content, no information is needed for maintaining consistency. Typically, pages can be set to timeout so the cache content is periodically refreshed. CachePortal [4] has a unique form of weak consistency, time-lagged consistency, in that it relies on timestamps and HTTP logs to conservatively determine which pages to invalidate. Inconsistencies can exist for a time between the cache and the backing store.

Whereas web pages are derived content, caching the contents of the persistent store directly, *i.e.*, non-derived data, a high degree of transparency with strong consistency can be achieved. Examples include caching direct copies of raw DBMS tables [32] or caching copies of persistent Java objects [14]. The key distinction about caching non-derived objects is that the complexity of maintaining strong consistency is greatly simplified by the object's unique global identifiers and the one-to-one mapping with the backing store data.

Examples of caching more complex derived data with strong consistency suffer from a low level of transparency that requires considerable developer input about request structure or dependencies. DynamicWeb [11] provides a strongly consistent web page cache, but not transparently as developers must define the dependencies between events, e.g., read and write queries. Similarly, form-based proxy caching [20] of web pages requires developers to pre-define configurations of web page formats. Weave [35] requires the programmer to use a specialized language to describe dynamic web pages and event handlers to specify invalidations. Various commercial solutions such as SpiderCache [28], Xcache [34], and Oracle9iAS [24] provide an event API to the developers to add consistency management.

Closest to our work is [8, 13] in which Websphere business rule SQL query result sets are cached. As with our work, strong consistency is maintained through complex analysis of the SQL queries. While in their implementation, the analysis aims at extracting dependencies between SQL queries and elements in the backing store, our caching system aims at determining the dependencies between web pages and data in the backing store (i.e., the cached data is doubly derived in the latter case – from data store to SQL and from SQL to web pages –, its consistency is therefore harder to maintain). Moreover, a high degree of transparency is achieved through the compiler-based solution to insert the cache API calls tuned to the Websphere environment. In contrast, our work uses much simpler AOP tools.

A primary contribution of this work is to demonstrate that achieving the simultaneous goals of complete transparency and strong consistency in web caching is likely not possible for the general case. The key problem is in automatically verifying that no essential data in an application needed for caching flows through unexpected interfaces and, thus, elude the consistency logic: cookies and internally generated data are two examples of this phenomenon from our benchmarks.

However, we also demonstrate through our AutoWebCache design that a *highly* transparent and strongly consistent solution is possible, so developer intervention is minimized. Furthermore, we show that using an AOP-framework combines simplicity with sufficient flexibility to

achieve our level of transparency; more specifically, there was no need to develop a complex compiler framework for this task as in [8]. Such a framework does not provide any additional benefits in this scenario, especially since run-time analysis in an AOP solution can be effectively eliminated through caching.

To the best of our knowledge, there has been no previously published work on providing support for strongly consistent caching of dynamic web content (derived content) using AOP. AOP techniques were experimented for profiling [7], persistence [25], distribution [15], web cache pre-fetching [26], caching static content [18], caching (non-derived) Java objects [14], and finally transactions [17] where the authors conclude that, as for consistent caching of dynamic web content, transactions can not be aspectized.

8 Summary and conclusions

Caching is usually presented as a motivating example for aspect-oriented programming, an example of an orthogonal aspect that could be somehow automatically added to an application. Some simple (academic) examples illustrate how to *aspectize* caching. Those examples usually tackle caching of static data with no need to manage cache consistency, or tackle consistent caching of non-derived data which is usually implemented in a straightforward fashion. We experimented the use of aspect-oriented programming to provide consistent caching of more complex data: we explore the feasibility of automatically adding *consistent caching* to dynamic web applications. In such applications, servers are generally heavily-loaded because of the dynamic generation of web documents, and caching is thus an appealing approach to reduce server load. We used two J2EE dynamic web applications to validate our experiments: the TPC-W on-line bookstore and the RUBiS auction site.

The motivation of our work was to address the following question: can we consider consistent caching of dynamic web content as orthogonal to applications? In other words, could caching in this context be *aspectized*? Our conclusions are that just as in the classic AOP caching example having no consistency management, AOP provides a modular way to add caching having a *strong* consistency policy; however, we were surprised to discover that:

a) Consistent caching of dynamic web pages may be a *non-orthogonal* aspect in some J2EE applications that rely on sophisticated web techniques (e.g., cookies).
b) Maintaining strong consistency of web pages in the cache results in prohibitively expensive and repeated run-time analysis and, thus, any straightforward implementation of strongly consistent web caching using AOP is too slow. We proposed an optimization that essentially eliminates *all* the run-time overhead in practice.

In summary, caching, which is usually presented as an orthogonal aspect, can not be completely aspectized in the particular case of consistent caching of dynamic web content.

References

1. C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, USA, Nov. 2002. http://rubis.objectweb.org
2. AspectJ 1.1, 2004. http://www.eclipse.org/aspectj/
3. S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, W. Zwaenepoel. *Caching Dynamic Web Content in J2EE Applications*. EPFL Technical Report ID: IC/2004/82, Oct. 2004.
4. K. S. Candan, W. S. Li, Q. Luo, W. P. Hsiung, D. Agrawal. Enabling Dynamic Content Caching for Database-driven Web Sites. *ACM SIGMOD'2001*, Santa Barbara, CA, USA, 2001.
5. R. Cattell, J. Inscore. *J2EE Technology in Practice: Building Business Applications with the Java 2 Platform, Enterprise Edition*. Pearson Education, 2001.
6. A. Colyer. *Implementing Caching with AOP*. TheServerSide.COM, June 2004. http://www.theserverside.com/blogs/showblog.tss?id=AspectJCaching
7. J. Davies, N. Huismans, R. Slaney, S. Whiting, M. Webster, R. Berry. Aspect Oriented Profiler. *2nd International Conference on Aspect-Oriented Software Development (AOSD'2003)*, Boston, Massachusetts, USA, Mar. 2003.
8. L. Denagro, A. Iyengar, I. Lipkind, I. Rouvellou. A Middleware System Which Intelligently Caches Query Results. *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, USA, Apr. 2000.
9. Edge Side Includes. http://www.esi.org/
10. A. Feldmann, R. Cáceres, F. Douglis, G. Glass, M. Rabinovich. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. *IEEE Conference on Computer Communications (INFOCOM'99)*, New York, NY, USA, Mar. 1999.
11. 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.
12. A. Iyengar, E. MarcNair, T. Nguyen. An Analysis of Web Server Performance. *IEEE Global Telecommunications Conference (GLOBECOM'97)*, Phoenix, Arizona, USA, Nov. 1997.
13. A. Iyengar, J. Challenger. *Data Update Propagation: A Method for Determining How Changes to Underlying Data Affect Cached Objects on the Web*. IBM Technical Report RC 21093(94368), IBM Research Division, Feb. 1998.
14. JBoss Inc. *JBossCache*. http://www.jboss.org/products/jbosscache
15. M. A. Kersten, G. C. Murphy. Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming. *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, Denver, Colorado, USA; Nov. 1999.
16. G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, J. Irwin. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finland, June 1997.
17. J. Kienzle, R. Guerraoui. AOP: Does it Make Sense? The Case of Concurrency and Failures. *16th European Conference on Object-Oriented Programming (ECOOP'2002)*, Málaga, Spain, June 2002.
18. R. Laddad. AspectJ in Action – Practical Aspect-Oriented Programming. Manning Publications, 2003.

19. M. H.. Lipasti. *Java TPC-W Implementation Distribution*. htt://www.ece.wisc.edu/~pharm/tpcw.shtml
20. 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.
21. MM-Mysql. *MM MySQL JDBC Drivers*. http://mmmysql.sourceforge.net/
22. MySQL. *MySQL Open Source Database*. http://www.mysql.com/
23. Netscape. *Persistent Client State – HTTP Cookies*. http://wp.netscape.com/newsref/std/cookie_spec.html
24. Oracle. *Oracle9iAS Caching Solutions*. Oracle Technical White Paper, Dec. 2001. http://otn.oracle.com/products/ias/web_cache/pdf/9ias_caching_twp.pdf
25. A. Rashid, R. Chitchyan. Persistence as an Aspect. *2nd International Conference on Aspect-Oriented Software Development (AOSD'2003)*, Boston, Massachusetts, USA, Mar. 2003.
26. M. Ségura-Devillechaise, J. M. Menaud, G. Muller, J. Lawall. Web Cache Prefetching as an Aspect : Towards a Dynamic-Weaving Based Solution. *2nd International Conference on Aspect-Oriented Software Development (AOSD'2003)*, Boston, Massachusetts, USA, Mar. 2003.
27. S. Sol, G. Berznieks. *Instant Web Scripts with Cgi Perl*. M & T Books, 1996.
28. Spider Software. *SpiderCache Enterprise 2.0:Dynamic Content Delivered Faster*. Spider Software Technical White Paper, Sep. 2001. http://www.spidercache.com/
29. Sun Microsystems. *Java 2 Platform Enterprise Edition (J2EE)*. http://java.sun.com/j2ee/
30. Sun Microsystems. *Java DataBase Connection (JDBC)*. http://java.sun.com/jdbc/
31. Sun Microsystems. *Designing Enterprise Applications with the J2EE Platform, Second Edition*. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/
32. TimesTen. *TimesTen Real-Time Event Processing System*. TimesTen White Paper, 2003. http://www.timesten.com
33. Transation Processing Performance Council. *TPC-W: a transactional web e-Commerce benchmark*. http://www.tpc.org/tpcw/
34. XCahe Technologies. XCache Overview. 2004. http://www.xcache.com
35. 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.

Contents

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois, Campus scientifique
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu – 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 – 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles – BP 93 – 06902 Sophia Antipolis Cedex (France)

Editeur
INRIA - Domaine de Voluceau - Rocquencourt - BP 105 – 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399