Multi-Cache Coherence Protocol for Distributed Internet Services

Damián Serrano[a], Sara Bouchenak[a], Ricardo Jiménez-Peris[b], Marta Patiño-Martínez[b]

[a]INRIA
Projet Sardes - Montbonnot St Martin, 38334 St Ismier Cedex, France
{Damian.Serrano, Sara.Bouchenak}@inria.fr

[b]Universidad Politécnica de Madrid
Distributed Systems Lab (LSD) - Campus de Montegancedo, 28660 Boadilla del Monte, Madrid, Spain
{rjimenez, mpatino}@fi.upm.es

Abstract
Multi-tier architectures provide a means for building scalable distributed services. Caching is a classical technique for enhancing the performance of systems (e.g. database servers, or web servers). Although caching solutions have been successfully studied for individual tiers of multi-tier systems, if collectively applied, these solutions may violate the coherence of cached data. This paper precisely addresses this issue. It presents e-Caching, an end-to-end caching system for multi-tier services aimed at guaranteeing data consistency across caches. The paper describes a novel caching protocol to automatically detect data dependencies and apply inter-cache invalidations. The implemented protocol is evaluated with the TPC-W online multi-tier bookstore service. The evaluation shows that e-Caching improves service performance compared to regular caching techniques while guaranteeing global data consistency in multi-tier systems.

1. Introduction

Today, many applications are implemented using multi-tier architectures. Internet services, which range from Web servers to streaming media services, are based on this kind of architectures. Classical three-tier Web applications start with requests from Web clients that flow through an HTTP front-end server and provider of static content, then to an enterprise server to execute the business logic of the application and generate web pages on-the-fly, and finally to a database that stores non-ephemeral data. To face high loads in multi-tier systems, a commonly used approach is caching [10]. Roughly speaking, caching consists in making a copy of request results for faster future reuse of these results.

Current caching solutions may successfully apply at individual tiers of multi-tier systems to cache database query results [7, 8, 12, 2], application business objects [6, 9, 11], or dynamic Web documents [1, 3, 5]. Intuitively, caching back-end tier results (i.e. database query results) provides higher cache hit ratio than caching front-end tier query results, since the same back-end query may be issued and shared by different front-end queries. Whereas, caching front-end tier results offers a greater benefit in terms of service response times in case of a cache hit, since front-end caching saves time on the front-end and on the back-end. Therefore, caching solutions at these different levels are complementary and would improve the overall performance of multi-tier systems. However, if put all together, existing individual caching solutions of multi-tier systems might result in globally inconsistent cached data across tiers, in case of dynamic and inter-dependent data.

The objective of this paper is to precisely bridge the gap between performance and consistency of caching in multi-tier systems as follows :

– *Automatic data analysis* to detect inter-dependent data across caches of all tiers.
– *Data consistency* across multi-tier cache systems by using the generated data dependencies.
– *Transparent caching* for an easier and automated integration of caching to multi-tier systems.

In this paper, we present e-Caching, an end-to-end caching system that provides consistency across cache levels. It guarantees consistency of cached data through an automatic multi-tier flow analysis. We describe a novel caching protocol that guarantees consistency of cached data through multi-tier flow

analysis and global cache consistency management. Furthermore, thanks to the automated query and flow analysis combined with a proxy-based approach, e-Caching can be easily applied to existent multi-tier systems. Other issues such as cache replication, location and replacement strategies fall out of the scope of this paper.
The rest of the paper is organized as follows. Section 2 describes the underlying system model, and Sections 3 and 4 present e-Caching design principles and protocol. The results of the evaluation of e-Caching in a real setting are described in Section 5. The related work is discussed in Section 6. Finally, Section 7 draws our conclusion.

2. System Model

Multi-tier system.

A multi-tier system is a distributed system that consists of multiple tiers $T_1, T_2,..,T_n$, as its name indicates. Each tier runs on a different node. Tier T_1 may receive requests from external clients and also act as a client of tier T_2, which itself may receive requests from external clients and act as a client of tier T_3, and so on. External clients of a tier are end-users, operators or administrators of the tier. Thus, all tiers of a multi-tier system act as clients and servers but tier T_n which is server-only. T_n is responsible of the storage of persistent data of the multi-tier application (e.g., in a database).

Nested requests.

A request sent to a server at tier T_i is denoted Q_i. Symmetrically, a result produced at T_i in response to Q_i is denoted R_i, and returned to the client that issued the request. A request handled at T_i may nest subsequent requests to T_{i+1}. We define $\mathsf{Sub}(Q_i)$ as the set of Q_i's subsequent requests. We define Q_i's preceding requests, $\mathsf{Pre}(Q_i)$, as the inverse function of Sub, that is the set of requests that induced Q_i from tier T_{i-1}, if any. The result R_i of request Q_i is built based on the results of Q_i's subsequent requests. Figure 1 presents an example of nested requests in a two-tier system. Request Q_{1_1}, handled by the front-end Web tier, calls subsequent requests Q_{2_1} and Q_{2_2} on the back-end tier, performs some local processing before generating Q_{1_1}'s result R_{1_1}. Requests Q_{2_1} and Q_{2_2}, handled by the back-end tier, perform some local processing before respectively generating results R_{2_1} and R_{2_2} returned to the front-end tier. These two latter results are themselves part of the global result R_{1_1}. Thus, $\mathsf{Sub}(Q_{1\_1}) =$

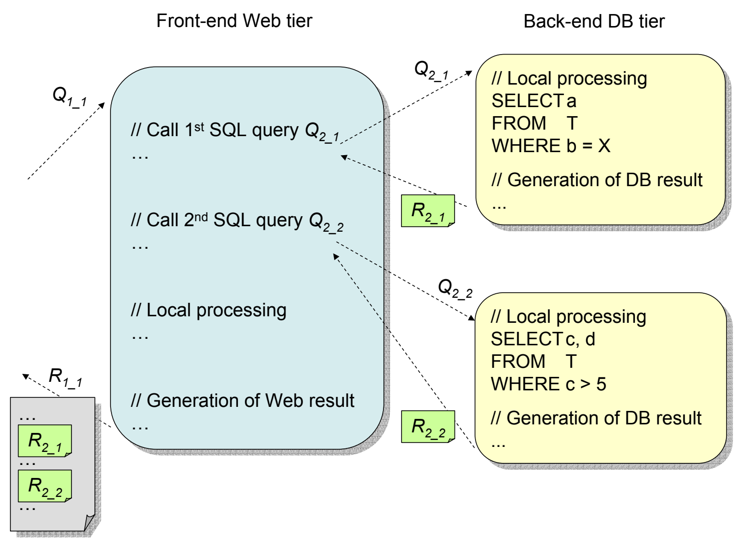


FIGURE 1 – Nested requests in a two-tier system

$\{Q_{2\_1}, Q_{2\_2}\}$ and $\mathrm{Pre}(Q_{1\_1}) = \emptyset$, $\mathrm{Pre}(Q_{2\_1}) = \mathrm{Pre}(Q_{2\_2}) = \{Q_{1\_1}\}$ and $\mathrm{Sub}(Q_{2\_1}) = \mathrm{Sub}(Q_{2\_2}) = \emptyset$. The back-end tier, that stores persistent data, may receive read-only queries or read-write queries. As indicated by its name, a read-only query $Q_{n/r}$ does not update the persistent data whereas a read-write query $Q_{n/w}$ does. We say that a read-write query *impacts* a read-only query if the former updates persistent data that the latter reads.

3. e-Caching Design Principles

Caching is an effective means for reducing load on servers [10]. It lies in making a copy of the results of a read-only request to directly serve future identical requests from that copy. This avoids redundant processing to regenerate request results. Moreover, a request Q_i that directly or indirectly induces subsequent requests to persistent data usually results in high response times because of database access. With e-Caching, if such a request is read-only, it becomes *cacheable* and benefits from better response times. Volatile, i.e. non-persistent, data is unlikely to be reused. Thus, a request on tier T_i is a read-only request $Q_{i/r}$ if it directly or indirectly (through subsequent request on T_{i+1}) induces access to persistent data in read-only mode. A request on tier T_i is cacheable if it is a read-only request. Symmetrically, a request on tier T_i is read-write, $Q_{i/w}$, if it induces directly or indirectly at least an access to persistent data in read-write mode.

3.1. Multi-tier caching

For higher performance, e-Caching enables caching at multiple levels of a multi-tier system. For instance, caching may apply to results produced by HTTP queries to the front-end web server, to results produced by requests to application components of the middle-tier application server, or to results produced by SQL queries to the back-end database server.

In the following, C_i denotes the cache that stores request results produced by tier T_i. Notice that this does not necessarily imply that C_i is located on a server at T_i but it may rather be located on a proxy server. For transparency purposes for the client, the cache C_i stands as a proxy in front of its associated server at tier T_i. A cached request has a unique identifier Q_i, such as a URL and its set of parameters for a web request, or an SQL query for a database request. Each cache C_i provides the interface with operations for the insertion of a new entry in the cache (i.e. a request and its result), for the lookup of the cached result of a request, or for the invalidation of an entry in the cache (Fig. 2).

```
// Insert an entry in cache $C_i$
$C_i.\mathrm{insert}(Q_i, R_i)$

// Fetch a cached result from cache $C_i$
$C_i.\mathrm{lookup}(Q_i)$

// Invalidate an entry from cache $C_i$
$C_i.\mathrm{invalidate}(Q_i)$
```

FIGURE 2 – Cache API for tier T_i

Note that if caches of request results of different tiers co-exist independently in the same multi-tier system, the coherence of cached data among the tiers is violated. This happens, for instance, when a dynamic web page requested by a web client is cached on a front-end cache, and later the persistent data embedded in that page are updated by an operator from the back-end tier. In that case, the back-end cache contains the updated data while the front-end cache keeps using invalid data. The main advantage of e-Caching is that it avoids this situation providing consistency across tiers. The following section describes how e-Caching works.

3.2. Multi-tier consistency

Local consistency.

Local consistency applies to the back-end tier cache C_n. Local consistency ensures that whenever a read-write request $Q_{n/w}$ is committed on T_n, all read-only queries whose results are cached in C_n and which are impacted by $Q_{n/w}$ are invalidated. Thus, cache C_n provides an additional function described in Figure 3(a) which returns read-only queries cached in C_n that are impacted (i.e. must be invalidated) by a given read-write request at tier T_n. For instance, the read-only SQL request `SELECT` $T.c_1$ `FROM` T `WHERE` $T.c_2 = X$ is impacted by the write SQL request `UPDATE` T `SET` $T.c_1 = new\_val$ `WHERE` $T.c_2 = X$. A more detailed example of such a function is described in Section 4.4.

Global consistency.

Besides local consistency that applies to the back-end tier cache C_n, global consistency between the different caches of a multi-tier system must be provided to guarantee the consistency of cached data in $C_1..C_n$. Indeed, a read-write request on tier T_i may impact the cached result of a read-only request in cache C_j if the former request updates persistent data that the latter request has read. Thus, e-Caching ensures that whenever a read-write request $Q_{i/w}$ is executed on T_i, all request results cached in caches $C_1..C_n$ and impacted by $Q_{i/w}$ are invalidated.
To provide global consistency, e-Caching is mainly based on two facts. On the one hand, local consistency of cached persistent data ensures that whenever a read-write request is executed on the back-end tier and impacts a cached read-only request of the back-end tier (i.e., the former updates data the latter reads), the latter request is invalidated from the cache. On the other hand, the precedence function, $\mathtt{Pre}$, allows tracing back the call path across tiers. Whenever a cached read-only request of the back-end tier is invalidated, the precedence function provides the read-only requests on the preceding tier that issued that invalidated request. Thus, if they are in the cache, these preceding requests are invalidated in the cache of the preceding tier through automatic cache invalidation propagation (see Figure 3(b)). This cache invalidation propagation is performed recursively from the back-end tier to the front-end tier until the first request in the call path using the $\mathtt{Pre}$ function. The proposed global cache consistency management protocol, e-Caching, is described in the following.

// Return entries in cache C_n that are
// impacted by write query $Q_{n/w}$
$C_n.Imp(Q_{n/w})$
	// Forward invalidations to queries
	// in the set QSet
	$C_i.forward\_invalidate(QSet)$

(a) API for cache local consistency	(b) API for cache global consistency

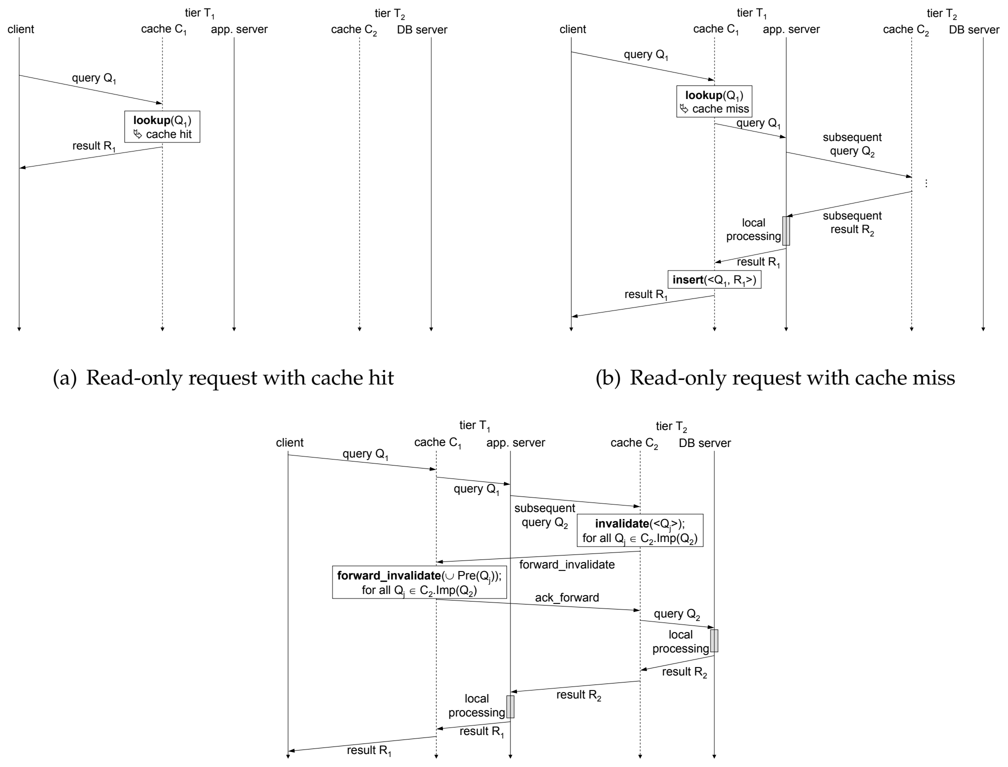
FIGURE 3 – Cache consistency API

Finally, whilst strong consistency is ensured for single-query transactions in the database, multi-query transaction isolation [9] must be tackled to fully provide strong consistency, but this is out of the scope of the paper. Furthermore, end-to-end transactional properties are assumed to be guaranteed, although this is also out of the scope of this paper [4].

4. e-Caching Protocol

4.1. Overview of caching protocol

In e-Caching, a proxy cache C_i intercepts requests to tier T_i to transparently integrate multi-caching to any multi-tier service and guarantee global consistency across the tiers as follows.
Figure 4(a) and 4(b) illustrate the case of a read-only request execution in case of a cache hit and a cache miss, respectively. For simplicity purposes, the figure illustrates the case of a two-tier system where a front-end request has one subsequent request, but this can be easily generalized to any multi-tier system with multiple subsequent requests. When a read-only request (Q_1) arrives to a tier, it first looks up the cache of that tier, fetches the result stored in the cache (C_1), and simply returns the result (R_1) to the client (Figure 4(a)). In Figure 4(b), a read-only request (Q_1) arrives to tier T_1, it looks up the cache of that tier (C_1) and does not find the corresponding result. In this case, the request is forwarded to the server of that tier (app. server) for its actual execution, which as part of the execution of Q_1 invokes Q_2 on tier T_2, and the result (R_1) is inserted in the cache (C_1) before it is returned to the client.
Figure 4(c) describes a read-write request arriving to a front-end or middle-tier cache. Here, the request is forwarded to be handled by the underlying server which itself sends a subsequent read-write request



(a) Read-only request with cache hit

(b) Read-only request with cache miss

(c) Read-write request

FIGURE 4 – e-Caching protocol execution scenari

to the following tier. When a read-write request reaches the back-end tier, end-to-end cache consistency management applies as follows. First, all entries in the back-end tier cache that are impacted by the current read-write request are invalidated from that cache (local consistency). This information is obtained thanks to the Imp function (Section 3.2). Then, for each invalidated entry in the back-end cache, the Pre function provides the set of preceding read-only requests that issued that invalidated read-only request, and that become themselves inconsistent. Thus, the back-end cache forwards invalidation to its preceding cache tier to guarantee global consistency. Cache invalidation forwarding applies recursively from the back-end to the front-end. Besides cache invalidation propagation, the read-write request and its subsequent read-write requests are executed by their respective servers at each tier.

We do not detail in this paper cache replacement strategies for e-Caching since they are considered orthogonal to this work. In case of eviction, the evicted entry is considered as invalidated and triggers the invalidation of the impacted entries in the preceding cache tier.

Finally, as described by the general behavior of e-Caching in Figure 4, if a request is cacheable (read-only), it is handled as described in cases 4(a) and 4(b). If the request is read-write, it corresponds to case 4(c). If the type of the request is not yet determined (i.e. request executed for the first time), that means that the request result is not cached (or uncacheable). In this case, the request is forwarded to the underlying server and its type will be dynamically determined as described in the following.

4.2. Request type analysis

A request type is either *uncacheable*, *cacheable* or *read-write*. A request is *cacheable* if it (directly or indirectly) accesses persistent data on the back-end tier and if all these accesses are in read-only mode[1]. A request is *read-write* if it (directly or indirectly) updates persistent data on the back-end tier. Otherwise, the request is *uncacheable*. The type of a request is automatically determined at runtime the first time the

1. e-Caching considers caching the result containing persistent data (volatile data is unlikely to be reused).

request is executed. Once determined, a request type is stored in a data structure that acts as a request type cache, and reused for faster future determination of request types. If not present in the request type cache, a request type is *undefined* and is dynamically determined as part of the request execution as follows. Algorithm 1 describes how request type analysis is integrated to the general behavior of a server at tier T_i. The integration is done in three well defined points : request prolog, request epilog and subsequent request epilog (lines 1, 8, 13).
Initially in the request prolog (Algorithm 2), the type of the request Q_i is undefined and the request is set as not accessing the back-end tier. Then, after the execution of Q_i's subsequent requests, the types of these subsequent requests are determined (see Algorithm 3). Thus, the types of the subsequent requests allow to determine if Q_i accesses persistent data on the back-end tier. Furthermore, if one of the subsequent requests is read-write, the type of Q_i is defined as read-write as well. Finally, if not yet defined, the type of Q_i is determined in the request epilog as described in Algorithm 7. In short, only requests generating SELECT queries in the database are cacheable. The three algorithms use the interface $Types_i$ (see Fig. 5(a)), an additional interface exposed by the cache at tier T_i. $Types_i$ provides operations that allow to set or get the type of a request Q_i, and to state that/check if a request Q_i (directly or indirectly) accesses persistent data on the back-end tier.

4.3. Request flow analysis

The e-Caching protocol builds upon a request flow analysis process for determining the Pre and Sub functions introduced in Section 2. In practice, e-Caching makes use of these functions for read-only requests. Request flow analysis is performed the first time a request is executed. For efficiency, its result is stored in a data structure to be reused when encountering the same request again. The proxy cache at tier T_i exposes an additional interface, the $Flow_i$ interface described in Figure 5(b). This interface defines operations that allow to get or set preceding requests of a request Q_i executed at tier T_i. Thus, the $Pre(Q_i)$ function introduced in Section 2 is equivalent to $Flow_i.getPreQueries(Q_i)$. Both can be used indistinctively in the rest of the paper. Similarly for $Sub(Q_i)$ and $Flow_i.getSubQueries(Q_i)$.

```
// Set/get the type of a query
$Types_i.setType(Q_i, type)$
$boolean\ Types_i.isReadwrite(Q_i)$
$boolean\ Types_i.isCacheableReadOnly(Q_i)$
$boolean\ Types_i.isUncacheable(Q_i)$
$boolean\ Types_i.isUndefined(Q_i)$

// Specify that/check if $Q_i$ accesses or not
// the back-end
$Types_i.setAccessesBackend(Q_i, boolean)$
$boolean\ Types_i.accessesBackend(Q_i)$
```

(a) Interface $Types_i$

```
// Get/set/add/remove preceding queries of $Q_i$
$querySet\ Flow_i.getPreQueries(Q_i)$
$Flow_i.setPreQueries(Q_i, querySet)$
$Flow_i.unsetPreQueries(Q_i)$
$Flow_i.addPreQueries(Q_i, querySet)$
$Flow_i.removePreQueries(Q_i, querySet)$

// Get/set/add/remove subsequent queries to $Q_i$
$querySet\ Flow_i.getSubQueries(Q_i)$
$Flow_i.setSubQueries(Q_i)$
$Flow_i.unsetSubQueries(Q_i)$
$Flow_i.addSubQueries(Q_i, querySet)$
$Flow_i.removeSubQueries(Q_i, querySet)$
```

(b) Interface $Flow_i$

FIGURE 5 – Interfaces at tier T_i

Request flow analysis is integrated into the general behavior of a server at three well identified points as it happens with the query type determination : the request prolog, the request epilog and the subsequent request epilog (see Algorithm 1). First, as the request prolog (see Algorithm 4), if the request Q_i is analyzed for the first time, its sets of preceding and subsequent requests are initialized as empty. Then, after the execution of each subsequent request Q_j that is *cacheable*, the relationship between Q_i and Q_j is stored (see Algorithm 5). Finally, if Q_i is *read-write* or *uncacheable*, the stored request flow is updated accordingly (see Algorithm 6). Moreover, whenever an entry is invalidated in cache C_i, the associated preceding and subsequent requests in the stored request flow are updated as described in Algorithm 8.

4.4. Request dependency analysis

The e-Caching protocol builds upon the Imp function. This function determines the read-only requests cached in the back-end tier cache that are impacted (invalated) by a read-write request executed on the back-end tier. This is based on the dependency (or not) between an SQL write query and an SQL read query. Roughly speaking, a dependency between a read request and a write request exists if the write modifies one or more columns in the row(s) being read, and/or results in changing the set of rows satisfying the selection clause of the read request [1].

Dependency analysis first checks if the database tables and columns used in the read request are also updated in the write request. To reduce false positive indications that may be induced by this column-only check, finer-grain checks are applied. Selection criteria in the read request WHERE-clause are matched to values from the write request to see if the same rows are being updated. For instance, if a read request and a write request are respectively as follows :

`SELECT` $T.c_1$ `FROM` T `WHERE` $T.c_2 = X$

`UPDATE` T `SET` $T.c_3 = new\_val$ `WHERE` $T.c_2 = Y$

and $X \neq Y$, then the requests are not interdependent.

Request dependency analysis can be made even more precise by executing extra requests to retrieve missing data needed to test for request intersection. This option generates additional requests to the back-end but also reduces unnecessary cache invalidations which improves cache hit ratio.

It is interesting to note that while the column intersection analysis is based on the static portion of the SQL query string (i.e. the request pattern), the other components of request dependency analysis come into play at run-time, once the actual values used in the selection criteria are known. For efficiency, e-Caching caches in a dedicated data structure the request patterns and the results of column intersection analysis between request patterns. These can be reused on receiving requests with the same pattern.

5. Evaluation

In this section, we evaluate the performance of e-Caching and show its consistency compared to other caching solutions.

5.1. Implementation and Evaluation Setup

We have evaluated our caching protocol using the TPC-W benchmark [14]. TPC-W implements an online bookstore and defines a web benchmark for evaluating e-commerce systems. TPC-W establishes three different workloads : browsing (5% writes), shopping (20%), and ordering (50%). The benchmark establishes a database with 10 tables. We implemented the proposed e-Caching protocol in the context of TPC-W where a front-end cache stores dynamically generated web pages and the back-end cache stores SQL query results. In the experiments, the cache completely fits in memory and we did not apply cache replacement strategies (e.g. LRU) that it is an orthogonal issue.

Using e-Caching only required to include approximately 150 lines of code to the TPC-W application. Since TPC-W is a two-tier Web application, entry points for Web requests or SQL requests are very well defined because they use defacto standard APIs (e.g. HTTP Servlet API, JDBC/SQL API). Following a proxy-based approach and intercepting entry points of each tier allowed us to capture requests/responses of these tiers, and thus include calls to e-Caching as request pre-processing and post-processing (cf. Algorithm 1). However, the integration of e-Caching to a multi-tier application could be automated using instrumentation techniques, aspect-oriented techniques, Java filtering techniques, or using proxies applied on standard APIs (e.g., J2EE, Servlets, JDBC).

The experiments were run in a cluster of 3 homogeneous nodes running Ubuntu 8.04. Each site is equipped with two processors Pentium 2.80GHz and 4 GB of RAM. One node ran Postgres 8.4, another one Tomcat 6.0 and the other the TPC-W clients. The two caches are collocated with Tomcat. Sites in the cluster are interconnected through a 1-GBit Ethernet switch.

We have run shopping and browsing workloads in the benchmark with 50 clients (EBs). All update transactions in the browsing workload were removed in order to have a read-only workload. The database population parameters were 256 emulated browsers (EBs) and 10,000 items which generated a database of nearly 1.5 GB. The measurements have been taken in between the ramp-up (150 seconds) and cool-down phases (120 seconds). The measurement phase lasts 300 seconds. The ramp-up phase initializes the system until it reaches a steady state. The cool-down phase maintains the load used during the measurement phase to allow requests of the measurement phase to complete. Each experiment was executed five times, the results show the average.

5.2. e-Caching performance

The goal of this experiment is to show the benefit in terms of response time when using e-Caching. We compare the results of e-Caching with the multi-tier system with no caching.

Figure 6 shows the response time for browsing (RO) and shopping (RW) mixes. Each workload has been tested with three different e-Caching configurations varying the duration of customer sessions : short sessions (RO-Low and RW-Low), default sessions (RO-Med and RW-Med) and long sessions (RO-High and RW-High). The longer the session, the higher the probability of hits in the cache and vice versa because each customer might request the same data during its session. The experiments with no caching produced similar results independently of the session duration. As expected, if there is no cache, the response time is very high. The results for the browsing mix show that when using e-Caching, the response time decreases from almost 50 ms. to 13, 10 and 1.5 ms. respectively. That indicates that the more the probability of cache hits, the better the response time is. The same behavior can be found for the shopping mix. Using e-Caching, response time is reduced from almost 60 ms. to 20, 18.5 and 7.75 ms. Note that during the shopping mix e-Caching is also maintaining data consistency across tiers when there are writes on the database.

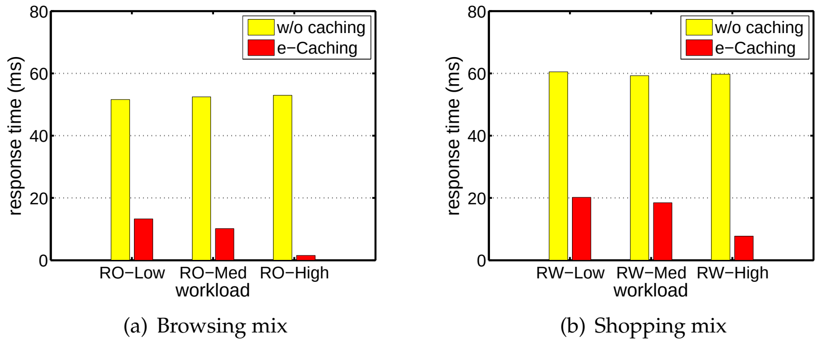


FIGURE 6 – No cache vs. e-Caching

Then, we compare e-Caching with two other caching solutions (Web caching and SQL caching) in order to exhibit the benefits of combining caches at multiple tiers instead of having a cache at a single tier. Web caching stores only front-end request results (web pages) and SQL caching stores only database results. Both caching solutions provide local consistency. In SQL caching, consistency is ensured as in e-Caching (see Section 3.2). Conversely, local consistency in Web caching is ensured with a pessimistic ad-hoc analysis, i.e., when detecting a write requests, all the cache entries that potentially could be impacted by that write request are invalidated.

Figures 7(a) and 7(b) show the response time obtained with e-Caching and the two other solutions. They also have been tested for short, default and long customer sessions with the browsing (RO-Low, RO-Med and RO-High) and shopping (RW-Low, RW-Med and RW-High) workloads.

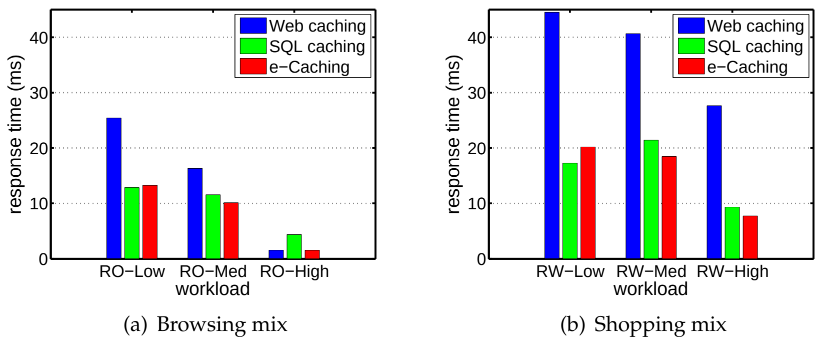
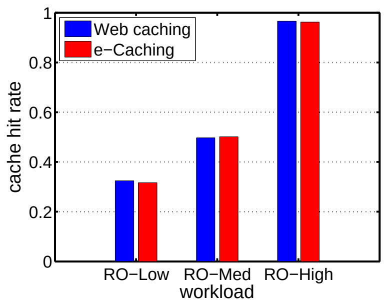


FIGURE 7 – e-Caching vs. single tier caching solutions

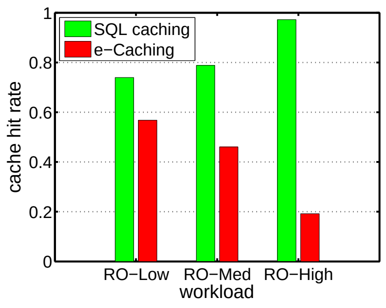
SQL caching provides better response time than Web caching with RO-Low and RO-Med. This happens because in order to generate a result (web page), the front-tier submits several requests to the database and different requests share some database queries. So, if a database request is already cached by SQL caching, the access to the database is avoided saving processing time. With RO-High due to the fact that the probability of hits is higher, Web caching performs better than SQL caching because a high rate of request does not need to go to the back-end. Finally, the lowest response time is obtained when e-Caching is used. This is due to the fact that some of the requests that are not cached in the front-end are cached in the back-end. Therefore, the combination of several caches improves the response time while provides data consistency across caches.

Figures 8(a) and 8(c) show the cache hit ratio obtained in the previous experiments in Web caching and in the front-end tier of e-Caching . As expected, the longer the customer session, the higher the hit ratio in both configurations. We obtained 30%, 50% and 95% of hits for RO-Low, RO-Med and RO-High, respectively, and 30%, 45% and 80% for RW-Low, RW-Med and RW-High. Note that for the shopping mix, the Web caching hit ratio is smaller because of the pessimistic consistency protocol (we will analyze this in detail in the next subsection).

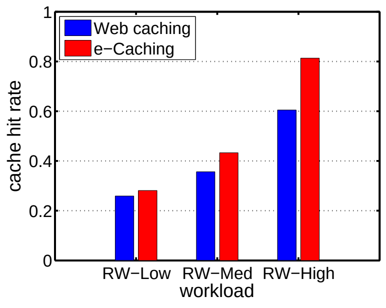
The hit ratio in SQL caching and in the back-end tier of e-Caching is depicted in Figures 8(b) and 8(d). Figure 8(b) shows that for the browsing mix and e-Caching the number of hits decreases at the back-end when customers have longer sessions. This happens because there are more hits in the cache of the front-end tier. That implies that, fewer requests are submitted to the back-end tier and therefore, there are fewer hits at the back-end. This does not hold for the shopping mix (Fig.8(d)). Since write requests are not cached at the front-tier, their subsequent requests are submitted to the back-end. The subsequent requests of a write request in TPC-W include several read requests to the back-end, and some of these requests are cached at the back-end. For instance, when a customer enters an order (a write request), its personal data and the content of his/her shopping cart may be already in the back-end cache.



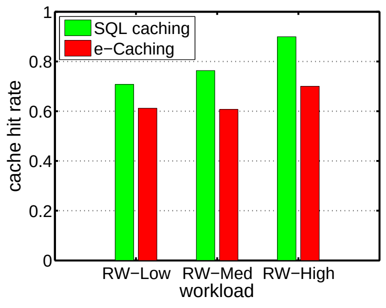
(a) Front-end - browsing mix



(b) Back-end - browsing mix



(c) Front-end - shopping mix



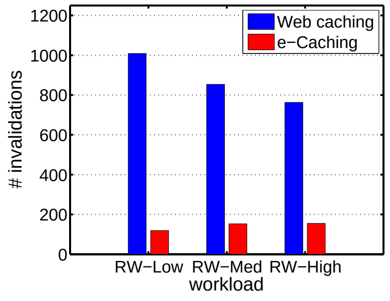
(d) Back-end - shopping mix

FIGURE 8 – Cache hits

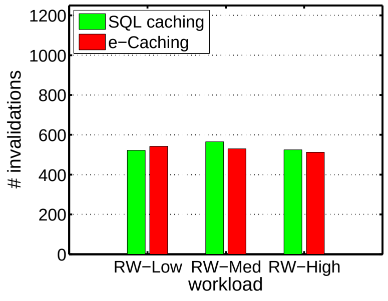
5.3. Caching invalidations

Inconsistencies might appear in caches in the presence of writes in the database. We analyze in this experiment how each caching solution avoids inconsistencies through invalidations. Web caching uses a pessimistic consistency protocol though an ad-hoc analysis of application requests that manually identifies write request and invalidates the set of possible cache entries that might be impacted by a write requests. On the other hand, both e-Cachingans SQL caching use the automatic consistency protocol for

the back-end cache described in Section 4. For the front-end cache, e-Cachinghas a more precise invalidation process since the invalidations in the back-end cache determine what cache entries are invalidated in the front-end cache. We first executed the shopping mix (it contains write requests) with both e-Caching and Web caching and counted the number of invalidated entries in their front-end cache. Figure 9(a) shows the results. Web caching invalidates on average almost 1000, 850 and 775 entries in the cache due to the pessimistic approach. In contrast, e-Caching invalidates only up to 160 entries because of the invalidations in the back-end tier. This measures the number of inconsistencies that e-Caching detects and avoids and because the smaller number of invalidations, e-Caching can benefit of better hit rates (cf. Figure 8(c)).
Finally, a similar number of invalidations happens at the back-end cache for both e-Caching and SQL caching (9(b)) because in both cases back-end caches contain similar results.



(a) Front-end tier cache



(b) Back-end tier cache

FIGURE 9 – Cache invalidations (shopping mix)

6. Related Work

Caching has been extensively studied in the context of Web and multi-tier Internet services [10, 15, 13]. Systems such as Wikimedia and Facebook propose multi-tier caching [2], [3]. However, these multi-tier caching systems are restricted to non-dynamic data or inter-dependencies among data in the caches are not well defined. Furthermore, in such systems there is unfortunately no attempt to automatize cache invalidations. In the following, we focus on caching dynamic content. Existing solutions mainly differ with regard to the type of cached data, cache consistency levels and cache transparency.
Type of cached data. Existing caching solutions in multi-tier systems handle one of the following types of cached data : dynamic Web content [1, 3, 5], application business objects [6, 9, 11], or database query results [7, 8, 12, 2]. Intuitively, caching back-end tier results (i.e. database query results) provides higher cache hit ratio than caching front-end tier query results, since the same back-end query may be issued and shared by different front-end queries. On the other hand, caching front-end tier results offers a greater benefit in terms of service response times in case of a cache hit, since front-end caching saves time on the front-end and on the back-end. e-Caching allows to jointly cache different types of data in a multi-tier Internet service and thus, combines benefits in terms of cache hit ratio and response times.
Cache consistency. Time-based weak consistency of data is proposed by some solutions caching query results from the database tier [3, 7, 8]. Other approaches provide strong consistency when caching results from the front-end tier [1], the middle-tier [6, 9], or the back-end tier [2]. However, this is limited to the local tier of the cached data, and hence inconsistencies may appear on other tiers of the multi-tier system. e-Caching proposes a multi-cache protocol providing global data consistency across the tiers.
Cache transparency. To be successfully applied, some caching approaches require considerable effort from the application developer to provide input about the application structure, the flow of its queries and their dependencies [5, 7, 8, 16]. In contrast, e-Caching automates the process of application analysis to dynamically and transparently build information about application query flow and dependencies.

2. http ://wikitech.wikimedia.org/images/f/ff/Ryan_Lane_-_Wikimedia_Architecture.pdf
3. http ://www.scribd.com/doc/4069180/Caching-Performance-Lessons-from-Facebook

7. Conclusions

We have presented e-Caching a consistent multi-tier caching system. e-Caching avoids the inconsistencies that might appear when combining independent caching systems at different tiers. The evaluation shows that althought e-Caching mantains data consistency, it provides a noticeable performance improvement by combining caching at multiple tiers compared to other single-tier caching solutions that also provides data consistency. e-Caching provides a set of well-defined interfaces that ease the integration of caching in any multi-tier system. This integration can be easily automated, providing transparent caching.

8. Acknowledgements

We would like to thank our anonymous reviewers for their insightful comments and specially our shepherd, Guillaume Pierre, who provided us valuable help to improve the final version of this paper.

References

1. S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, and W. Zwaenepoel. Caching Dynamic Web Content : Designing and Analysing an Aspect-Oriented Solution. In *Middleware*, 2006.
2. C. Amza and G. Soundararajan and E. Cecchet. Transparent Caching with Strong Consistency in Dynamic Content Web Sites. In *ICS*, 2005.
3. K. S. Candan, W.S. Li, Q. Luo, W.P. Hsiung, and D. Agrawal. Enabling Dynamic Content Caching for Database-driven Web Sites. In *ACM SIGMOD*, 2001.
4. S. Frølund and R. Guerraoui. e-transactions : End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering*, 28(4), 2002.
5. A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *USITS*, 1997.
6. A. Leff and J.T. Rayfield. Improving Application Throughput With Enterprise JavaBeans Caching. In *ICDCS*, 2003.
7. Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B.G. Lindsay, and J.F. Naughton. Middle-Tier Database Caching for e-Business. In *ACM SIGMOD*, 2002.
8. Oracle. Oracle9iAS Caching Solutions, 2009. http ://www.oracle.com/.
9. F. Pérez, M. Pati no Martínez, R. Jiménez-Peris, and B. Kemme. Consistent and Scalable Cache Replication for Multi-Tier J2EE Applications. In *Middleware*, 2007.
10. M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Pearson Education, 2001.
11. RedHat. JBoss Cache, 2009. http ://www.jboss.org/jbosscache/.
12. L. Rilling, S. Sivasubramanian, and G. Pierre. High Availability and Scalability Support for Web Applications. In *SAINT*, 2007.
13. S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. Analysis of Caching and Replication Strategies for Web Applications. *IEEE Internet Computing*, 11(1), 2007.
14. Transaction Processing Performance Council (TPC). TPC-W transactional web e-Commerce benchmark. http ://www.tpc.org/tpcw/.
15. J. Wang. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Computer Communication Review*, 29(5), 1999.
16. K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching Strategies for Data-Intensive Web Sites. In *VLDB*, 2000.

Algorithm 1: Request handling on server at T_i

```
Input:
$Q_i$ : request received by server at $T_i$
Output:
$R_i$ : result of request $Q_i$
1  Request prolog : type determination, flow analysis
2  /* Initial empty result */
3  $R_i$ := " "
4  foreach $Q_j$ induced by $Q_i$ do
5      /* Subsequent request execution */
6      send request $Q_j$ to $T_{i+1}$
7      wait until receive $R_j$ from $T_{i+1}$
8      Request epilog : type determination, flow analysis
9      /* Result composition */
10     $R_i := R_i \oplus R_j$
11 /* Local processing and $R_i$ finalization */
12 ...
13 Request epilog : type determination, flow analysis
14 return $R_i$
```

Algorithm 2: Request type determination - Q_i request prolog

```
if $Types_i.isUndefined(Q_i)$ then
    $Types_i.setAccessesBackend(Q_i, false)$
```

Algorithm 3: Request type determination - Subrequest epilog

```
// After $Q_j$ execution, i.e. $Q_i$'s subsequent request
if $Types_i.isUndefined(Q_i)$ then
    if $Types_{i+1}.accessesBackend(Q_j)$ then
        $Types_i.setAccessesBackend(Q_i, true)$
    if $Types_{i+1}.isReadWrite(Q_j)$ then
        $Types_i.setType(Q_i, read - write)$
```

Algorithm 4: Request flow analysis - Q_i prolog

```
/* First time request is analyzed, initialize its flow */
if $Types_i.isUndefined(Q_i)$ then
    $Flow_i.setPreQueries(Q_i, \emptyset)$
    $Flow_i.setSubQueries(Q_i, \emptyset)$
```

Algorithm 5: Request flow analysis - Q_i's subrequest epilog

```
/* If subsequent request $Q_j$ is cacheable_read_only, add $Q_i$ as its preceding request */
if $Types_i.isUndefined(Q_i)$
 and $Types_{i+1}.isCacheableReadOnly(Q_j)$ then
    $Flow_i.addSubQueries(Q_i, \{Q_j\}))$
    $Flow_{i+1}.addPreQueries(Q_j, \{Q_i\})$
```

Algorithm 6: Request flow analysis - Q_i epilog

```
/* Request type is defined : if uncacheable or read - write request, not used by e-Caching, remove it from the stored flow */
if $Types_i.isUncacheable(Q_i)$
 or $Types_i.isReadWrite(Q_i)$ then
    foreach $Q_j$ in $Flow_i.getSubQueries(Q_i)$ do
        $Flow_{i+1}.removePreQueries(Q_j, \{Q_i\})$
    $Flow_i.unsetPreQueries(Q_i)$
    $Flow_i.unsetSubQueries(Q_i)$
```

Algorithm 7: Request type determination - Q_i request epilog

```
if $Types_i.isUndefined(Q_i)$ then
    if $i = n$ then
        /* Determine SQL query type */
        if $Q_i$ is an SQL SELECT then
            $Types_i.setType(Q_i, cacheable\ read-only)$
        else
            $Types_i.setType(Q_i, read - write)$
        $Types_i.setAccessesBackend(Q_i, true)$
    else if not $Types_i.accessesBackend(Q_i)$ then
        $Types_i.setType(Q_i, uncacheable)$
    else
        $Types_i.setType(Q_i, cacheable\ read-only)$
```

Algorithm 8: Request flow update in $C_i.invalidate(Q_i)$

```
foreach $Q_j$ in $Flow_i.getSubQueries(Q_i)$ do
    $Flow_{i+1}.removePreQueries(Q_j, \{Q_i\})$
$Flow_i.unsetPreQueries(Q_i)$
$Flow_i.unsetSubQueries(Q_i)$
```