Consistent Freshness-Aware Caching for Multi-Object Requests

Meena Rajani[1], Uwe Röhm[2], and Akon Dey[2]

School of Information Technologies
The University of Sydney
Australia
meenakrajani@gmail.com,
{uwe.roehm,akon.dey}@sydney.edu.au
http://www.usyd.edu.au

Abstract. Dynamic websites rely on caching and clustering to achieve high performance and scalability. While queries benefit from middle-tier caching, updates introduce a distributed cache consistency problem. One promising approach to solving this problem is *Freshness-Aware Caching (FAC)*: FAC tracks the freshness of cached data and allows clients to *explicitly* trade freshness of data for response times. The original protocol was limited to single-object lookups and could only handle complex requests if all requested objects had been loaded into the cache at the same time. In this paper we describe the *Multi-Object Freshness-Aware Caching (MOFAC)* algorithm, an extension of FAC that provides a consistent snapshot of multiple cached objects even if they are loaded and updated at different points of time. This is done by keeping track of their *group valid interval*, as introduced and defined in this paper. We have implemented *MOFAC* in the JBoss Java EE container so that it can provide freshness and consistency guarantees for cached Java beans. Our evaluation shows that those consistency guarantees come with a reasonable overhead and that *MOFAC* can provide significantly better read performance than cache invalidation in the case of concurrent updates and reads for multi-object requests.

Keywords: Freshness, Distributed cache, Replication, Invalidation, Consistency.

1 Introduction

Large e-business systems are designed as n-tier architectures: Clients access a web-server tier, behind which an application server tier executes the business logic and interacts with a back-end database. Such n-tier architectures scale-out very well, as both the web and the application server tiers can be easily clustered by adding more servers. However, there is a certain limit to the performance and scalability of the entire system due to the single database server in the back-end. It is essential to minimize the number of database calls to alleviate this bottleneck.

A distributed cache layer at the application server level strives to minimize these access costs between the application server tier and the database tier. While this works very well for read-only access, updates induce a distributed cache consistency problem. Keeping the contents of the cache consistent with the backend database is necessary to ensure correctness of the overall system. The well-known techniques for handling this are cache invalidation and cache replication, both of which have certain disadvantages with regard to the best performance of either read- or update-intensive workload. In an n-tier architecture, business logic is processed in the application server making it more efficient by keeping the data closer to it.

The Java application server comes with Enterprise Java Bean (EJB) to manage business logic and persist application state. An entity bean usually represents a row from the database table and is hence costly to to create, presenting the need for a a second level cache. As a result, when an entity bean is updated in one node, either a replication or invalidation message is sent across all the nodes in the cluster. Both synchronous and asynchronous replication of entity beans are costly when data consistency issues are created due to invalidation. Invalidation of entity beans on the other hand causes cache misses and in-turn can cause database bottlenecks.

FAC showed that we can trade-off freshness with high availability but was limited to freshness management on the basis of a single Enterprise Java Bean (EJB). However, in real world applications most transactions access more than one object that exist in binary or ternary relationships with other objects. As a result, when an object is updated in a database, that object and all its associated objects are removed from the cache resulting in a high performance penalty. This must be avoided in order to maintain high performance. In this paper, we make the following contributions:

– We present a Multi-Object Freshness Aware Caching (MOFAC) algorithm which guarantees both the freshness and inter-object consistency of the cached data.
– We implemented this algorithm in the JBoss 6 middle-tier application server cache.
– We present results of a performance evaluation and quantify the impact of the different parameters of MOFAC on its performance.

2 Freshness-Aware Caching

In Freshness-Aware Caching (FAC) [11], each cache node keeps track of how stale its content is and only returns data that is fresher than the freshness limits set by the client. In this paper, we describe algorithms that can support this, and which also ensures that every request that touches multiple objects is given a consistent view of them; that is, there was a time, within the freshness limit, when all the information read was simultaneously up-to-date.

2.1 Freshness Concept

Freshness of data is a measure on how outdated (stale) a cached object is in comparison to the up-to-date master copy in the database [12]. There are several approaches to measuring this: *Time-based* metrics rely on the time duration since the last update of the master copy, while *value-based* metrics rely on the value differences between cached object and its master copy. A time-based staleness metric has the advantage that it is independent of data types and does not need to access the back-end database. On the other hand, a value-based metric needs the up-to-date value to determine the value differences allowing for local freshness decisions without making a trip to database to improve scalability. Hence, MOFAC uses a time-based metric.

Definition 1 (Staleness Metric). *The* staleness *of an object o is the time duration since the object's stale-point, or* 0 *for freshly cached objects. The* stale-point $t_{update}(o)$ *of object o is the point in time when the master copy of o was last updated while the cached object itself remained unchanged.*

$$stale(o) := \begin{cases} (t_{now} - t_{update}(o)) & \mid \textit{if } master(o) \textit{ updated at } t_{update}(o) \\ 0 & \mid \textit{otherwise} \end{cases}$$

Definition 2 (Freshness Intervals). *The cache lifetime of an object o consists of two disjoint intervals, $vi(o)$ and $di(o)$. Its* valid interval, *$vi(o)$, is the half open interval $[t_{load}, t_{update})$ when the object is loaded into the cache. The object is in the valid interval when it retains its state over a period of time until some event occur in the present or future that changes its state in database. The object's* deferred interval *$di(o)$ is defined as the half open interval $[t_{update}, t_{expire})$ such that the object enters the deferred interval as soon as it is updated in database.*

The timestamp of the object is recorded when it is loaded from the database to the cache node. When an object is updated via the cache node or any other cache nodes in the cluster, the update timestamp is recorded. The expiry time is calculated using the update timestamp and freshness constraint. In other words, the length of the deferred interval is adjusted by the freshness constraint. This is illustrated in Figure 1.

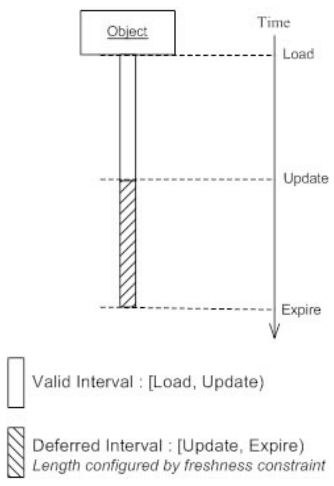

Object
Time
Load
Update
Expire
Valid Interval : [Load, Update)
Deferred Interval : [Update, Expire)
Length configured by freshness constraint

Fig. 1. Cache Objects Freshness Intervals

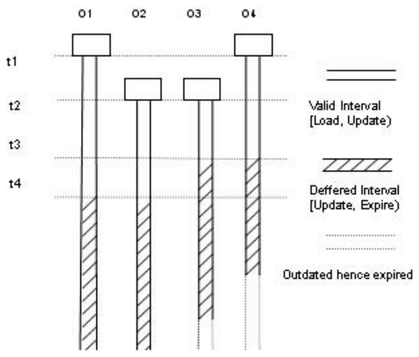
If an object is inside the valid interval, it is considered to be fresh and consistent with the backend database; if its timestamp falls inside the deferred interval, it means that the master copy on the database has been updated, but the staleness of the object still meets the freshness constraint. Only objects that have exceeded the expiry timestamp, are considered to be too old with respect to the freshness constraint and are evicted from the cache.

3 Multi-Object Freshness-Aware Caching (MOFAC)

The FAC algorithm, as described, cannot guarantee the consistency of several related objects that are accessed within the same transaction. We introduce a concept of object grouping to address this limitation.

An object group represents a set of logically related objects in which two entities are considered to be part of the same group of objects if they are associated with each other in an explicit or implicit relationship. For example, this can be done by leveraging the foreign-key-relationships in the schema of the underlying database which is often explicitly defined in the object-relational mapping at the middle-tier.

The objects in a group are mutually consistent when all of them have been persisted together in the database. Object grouping enables *snapshot consistency* to the objects in the group by ensuring that modifications to any object in the group results in notifications making all cached copies of the group stale. However, they may continue to reside in the cache as long as the freshness conditions are met using the formula described in Section 3.1. All members of the group are guaranteed to be loaded with the same snapshot of the database reflecting the freshness interval of the entire object group.



O1
O2
O3
O4
t1
t2
t3
t4
Valid Interval [Load, Update)
Deffered Interval [Update, Expire)
Outdated hence expired

Fig. 2. Multi Object Group Freshness

Definition 3 (Group Freshness Intervals). *Suppose we have a group of objects $G = \{o_1, o_2, ..., o_n\}$. The group's* valid interval*, denoted by* gvi*, is the intersection of all valid intervals of its group members:*

$$gvi(G) := vi(o_1) \cap vi(o_2) \cap ... \cap vi(o_n), o_i \in G$$

In other words, the gvi is the half-open time interval defined as

$$gvi(G) := [MAX(t_{load}(o_i)), MIN(t_{update}(o_i))),$$

$$1 \leq i \leq n, o_i \in G$$

Figure 2 illustrates an example in which o_1, o_2, o_3 and o_4 are a group of objects with the same meaning of freshness interval in Figure 1. The objects have been loaded into the cache at different points in time within the interval (t_1 and t_2).

When an object from this group is requested, the group's valid interval is calculated to verify that all the objects were in their valid interval together at some point of time; in this case, it is $[t_2..t_3)$. If this satisfies the user's freshness constraint, the user can access the cached group of objects. If an object in a group is updated by some other cache node, the stale point is registered (t_3 or t_4 in this example). Note that subsequent updates on already stale objects do not change the staleness point. We call a cache consistent group where *all* members are in their valid interval, *fresh*, and otherwise, *stale.*

Currently we have considered objects which are in one to many and many to one associations. This can be easily extended to many to many associations. If two different transactions require the same group of objects, only one instance of that group exists in the cache.

Definition 4 (Group Staleness Metric). *The* staleness *of a group of objects, G, is the time duration since the first update to any object in G until now, or is* 0 *for cached group G, if the master copy of all o in G is not updated since G is loaded into the cache.*

$$stale(G) := \begin{cases} (t_{now} - MIN(t_{update}(o_i))) & |\ \textit{if master}(o)\textit{updated at } t_{update}(o) \\ 0 & |\ \textit{otherwise} \end{cases}$$

In order to ensure that an application gets a group of objects which are mutually consistent and are fresh enough to serve a user request, each object in the group must be in the valid interval and the staleness of group G must satisfy the user-defined freshness limit.

Definition 5 (Group Consistency and Freshness). *A group of objects G is fresh enough and consistent if the group is present in cache, the group valid interval is not empty (i.e. at some point in time in the past, all group objects were valid at the same time), and the staleness of G is within the application's freshness limit.*

$$freshcon(G) : G \in Cache \ \wedge \ gvi(G) \neq \emptyset \ \wedge \ stale(G) \leq freshlimit$$

3.1 Example

Let us consider an online bookstore application where a book entity is brought into the application cache together with its authors and reviews as a group G. This could be because all the parts are needed to construct the content of a dynamic web page.

$$\begin{gathered} Cache := \{(b, \{a\}, \{r\}) | b \in Books, a \in Authors, r \in Reviews : \\ a.id \in b.authors \ \wedge \ r.isbn = b.isbn\} \end{gathered}$$

Here, each group consists of one Book object, and a set of corresponding Author and Review objects. In this example, we can guarantee a user is accessing a consistent data snapshot, which is within the required freshness constraint, by using Definition 5.

Figure 3 is an example of a book group. It has one book instance which is associated with two collections: one review collection and one author collection. The key of collection is generated from the root object's key, which is 1 here. When any object is updated in one of the cache nodes, the rest of the cache nodes receive the modified timestamp with the key of the corresponding object. They then register that timestamp and convert the group's status from fresh to stale.

4 MOFAC Algorithm

In this section we describe the multi-object freshness-aware caching (MOFAC) algorithm that handles freshness of multi-objects freshness in the application-tier cache. It consists of three sub-algorithms: the handling of MOFAC reads, the handling of user-level updates to a local cache node, and the processing of update-notifications on a remote cache node.

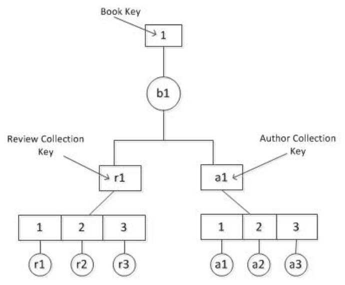


Fig. 3. Example of a book group with two child collections

4.1 Multi Object Freshness Read

A freshness-aware cache tracks the load and stale points of each cached object, group memberships and each groups valid intervals. This meta-data is used to decide whether an object can be returned from the cache or whether it has to be (re-)loaded from the backend database.

In Algorithm 1, it is worth to noting that on an initial cache miss for the whole group, all objects that form the group are loaded into the cache together. This is typically done by the `cacheable` and `association` annotations of an application. In addition, the usual cache granularity is at the level of individual objects and not cache groups. So in most applications, this algorithm is initiated multiple times in a row while the application is traversing the different object links within the same group. Most of these end us as fast in-memory operations because the whole group (complex object) is loaded into the cache due to the earlier cache misses. This cache miss behaviour is handled in detail in the MOFAC read-algorithm in Algorithm 1.

This algorithm assumes that the cache has mechanisms to determine dependencies between associated objects, such as to iterate over all direct child objects of a given object in the cache or to determine whether a complex object is completely cached with all its associated child objects (predicate *isComplete*() in

Algorithm 1. MOFAC Read Algorithm for an arbitrary complex object

```
input  : an object key $k$
input  : freshness limit $f_{limit}$
output: object reference $O$

$O \leftarrow$ lookup($k$)
if $O \notin Cache \vee \neg isComplete(O)$ then
    for all $o_i \in O, o_i \notin Cache$ : do
        retrieve $o_i$ (evtl. with child objects) from database
        $t_{load}(o_i) \leftarrow t_{now}$
        $t_{update}(o_i) \leftarrow t_{max}$
        $Cache \leftarrow Cache \cup o_i$
    end for
end if
if $gvi(O) = \emptyset$ then
    for all $o_i \in O, vi(o_i) \notin gvi(O)$ : do
        evict $o_i$ from Cache and reload (evtl. with child objects) from database
        $t_{load}(o_i) \leftarrow t_{now}$
        $t_{update}(o_i) \leftarrow t_{max}$
        $Cache \leftarrow Cache \cup o_i$
    end for
end if
if $stale(O) > f_{limit}$ then
    evict stale $O$ from Cache and reload (evtl. with child objects) from database
    $t_{load}(O) \leftarrow t_{now}$
    $t_{update}(O) \leftarrow t_{max}$
    $Cache \leftarrow Cache \cup O$
end if
return $O$
```

above's algorithm). This functionality is provided by the Java Persistence API (JPA) layer. The cache can determine, which objects should be cached together and whether a complex object (including (or references) sub-objects is completely cached or not, based on meta-data that is extracted from the annotations in the Java application code.

4.2 Update Handling on Local Cache Node

We have to distinguish between two cases when processing updates on a multi-object freshness aware cache: Firstly, how should updates be processed locally on the cache node that received the user transaction. Secondly, how should the update notifications be processed on the other nodes of the distributed cache.

Algorithm 2 listed in the Appendix describes how updates should be handled at a local cache node: The object to be updated is first persisted to the backend database and then corresponding update-notifications are sent to the other cache nodes in the cluster. These notifications differ slightly based on what kind of object was updated in the cache group (i.e. whether it is the root node, a child object or a child collection). All update notifications are sent within the original user transaction, so that we have a synchronous freshness update to all the nodes. It does not require any expensive 2-phase-commit protocol since the only issue that can arise on a remote node is that it may not have the object in the cache when the update message is received. In this case, the message can safely be ignored on the update-notification. All we need to have is a guarantee that the notifications are delivered so that nodes who indeed do cache the same object get notified.

4.3 Update Handling: Processing Update-Notifications

The second part of the MOFAC update algorithm is the reception of the update-notification on a remote node. Although we conceptually get three different kinds of update-notifications, for either a whole group or just a child object or child collection, the actual handling is the same just differing in the type of target object.

Algorithm 3 listed in the Appendix describes these steps. An update notification not only specifies which object has been modified (the request specifies the unique object identifier, but not the object itself) and the timestamp on when this happened at the original node. If the modified object is also present in the receiving local cache, and no message was received with respect to an earlier update, then the stale point of the cached copy of the updated object is set to the received timestamp.

The algorithm assumes that all nodes in the cluster are time synchronised so that these timestamps between nodes are comparable. This is not difficult assumption for a typical closely-coupled cluster of today's standard. However, when the caches are distributes over a wide-area, we recommend switching to local timestamps of receiving notifications. This might be later than when the original update happened on the remote machine, but would be more consistent with all other timestamps used for MOFAC comparison which also are all locally determined (such as load time or time of an application request).

5 Evaluation

We have implemented the MOFAC algorithm inside the in-memory cache of a Java EE platform version 5.0 server and evaluated the performance characteristics of our proposed method using an exemplified dynamic web application: a simplified online bookstore.

5.1 Benchmark Application

The bookstore benchmark application consists of three components: A client emulator, the clustered bookstore server application, and the backend database.

The client emulator is a multi-threaded Java application that simulates a configurable number of clients that access the bookstore with either browsing (read-only) or buying (read-write) request. For the browsing workload, a method is invoked to find a certain book with all its authors and reviews.

The bookstore server application consists of a session bean, that provides the corresponding browsing and buying calls, as well as the implementation of the three entity beans representing Book, Author and Review entities. It is deployed into a JBoss 6 application server container and configured to run on a variable number of cluster nodes. For the caching side, we configured JBoss to use Infinispan as a distributed caching tier. Note that although Infinispan is a separate product and comes with its own configuration files, it is indeed loaded as part of the JBoss installation into the same JVM when the application server starts. We configured Infinispan so that it tightly integrates with the JBoss container by installing a caching interceptor into the JBoss interceptor chain, so that it gets invoked with any EJB access.

Finally, the bookstore state is stored in a single backend PostgreSQL database that is shared among all JBoss/Infinispan instances. We have used entity bean POJO entity class to persist and load data to and from the database into three kinds of entity beans: `Book`, `Author` and `Review`. Book and Author are in a one-to-many relationship, while Book and Review are also in a one-to-many relationship. The details of these ORM definitions of the three entity beans are shown in the Appendix in Listing 1.1.

5.2 Evaluation Setup

All experiments were conducted on an evaluation system consisting of a small cluster of eight Dell Optiplex servers, each equipped with a quad-code Intel Core2 Q9400 CPU (2.66 GHz), 4 GB RAM, two 500 GB HDDs, and running RedHat Fedora Linux 10 (kernel version 2.6.27.30).

We used a Java-based test client simulator and JBoss version 6.0 application server. The client simulator was running on a dedicated separate computer, and another dedicated server was used as back-end database server, running PostgreSQL Server 9.1. All nodes were interconnected via Fast Ethernet. The communication between the JBoss Server instances in the cluster (partition) is handled by the JGroups group communication library via *channel* for node discovery and reliably exchanging messages among cluster nodes. We have configured the cluster to use a round robin load balancing policy.

Client simulator and application server were Java applications executed under Java version 1.6. The server was executed with a Java heap size of up-to 512 MB (option $-Xmx512m$).

5.3 Evaluation of MOFAC's Overhead

In the first evaluation series, we are interested in measuring the general overhead induced by MOFAC in comparison to the standard cache invalidation techniques of JBoss/Infinispan. We compare the following cache functionalities:

INV synchronous cache invalidation (cache invalidation with synchronous notifications to remote nodes)

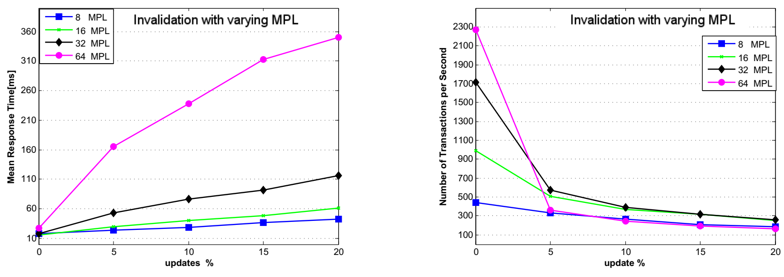


Fig. 4. Mean Response Time and Throughput of Cache Invalidation on Cluster Size 8 with varying Update Ratio

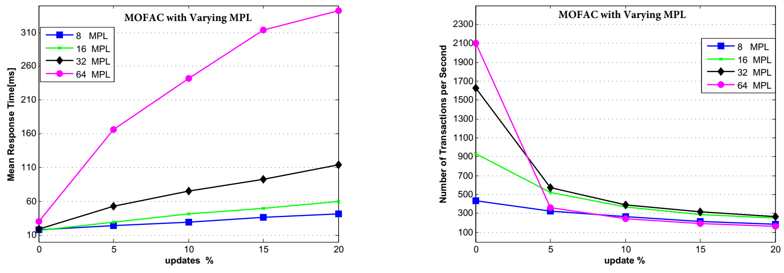


Fig. 5. Mean Response Time and Throughput of MOFAC on Cluster Size 8 with varying Update Ratio and Freshness Limit 0

MOFAC using the synchronous communication mechanism to send freshness notification to remote nodes.

In order to determine the costs of the caching control code, we fixed the freshness limit for all client requests to 0. This ensures that MOFAC caching produces about the same number of cache misses than cache invalidation - any update will trigger the eviction of any of its replica in other cache nodes. We evaluated both the MOFAC cache and the invalidation cache for varying update rates and varying multi programming level (MPL – number of concurrent clients) (Figures 4 and 5). The main difference is that with cache invalidation, this happens eagerly, directly at the end of the original update transaction. While on the other hand, with multi-object freshness-aware caching, it happens 'lazily', only when another transaction with freshness limit 0 tries to access a stale copy in a cache node.

Figures 4 and 5 shows that there is no measurable overhead for multi-object freshness aware caching as compared to the standard cache setting with cache invalidation. The two curves are always within a certain confidence interval of each other, in particular for the higher ratio of updates when a lot of invalidations are triggered within the system.

5.4 Evaluation of Invalidation vs. MOFAC with Varying Update Ratio

In this experiment we have multi-object read and update transaction, and we compare MOFAC with the Invalidation algorithm to measure the impact of the update ratio on the performance. The multi-programming level (MPL) is 64 is kept constant throughout this experiment. Figure 6 shows that increase in update ratio impacts performance of both the MOFAC and Invalidation algorithms.

However, if we increase the freshness level of MOFAC, we see a clear difference in throughput and response time between the two algorithms. As we relax the freshness limit, MOFAC shows reduced response time and increased throughput (even with higher update ratio). Where as in the case of invalidation, even a slight increase in the update ratio results in reduced throughput and increased mean response time.

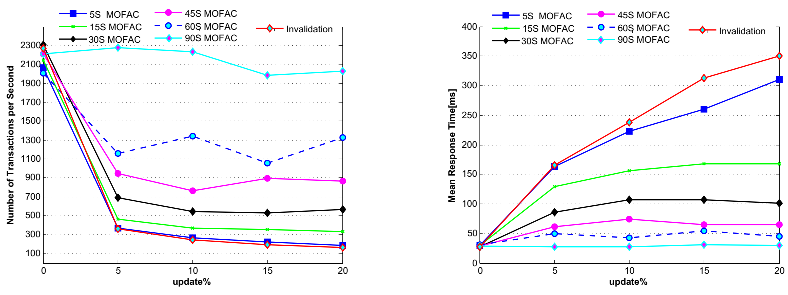


Fig. 6. Mean Response Time and Throughput of MOFAC vs Invalidation on Cluster Size 8 with varying Update Ratio

5.5 Evaluation of Varying Freshness Limit and Varying MPL

In the third experiment, we investigate the effect of varying the freshness limits and MPL. We fix the cluster size to 8 nodes and vary the MPL from 8, 16, 32 through to 64 and use a workload in which read-only transactions concurrently and randomly access a book object with it reviews and authors, while update transactions write to an existing book object. We vary both the amount of update transactions and the freshness limit from 0, 15, 30, 45, 60 to 90 seconds for books, authors and reviews.

As we can see in Figure 7, as the freshness limit increases, we get faster response times. This is exactly what we aim for with multi-object freshness-aware caching – to be able to trade freshness of data for query performance. When the freshness limit is set to 5 seconds, the response time is close to that of cache invalidation. With a freshness limit of 90 seconds, the response time is up-to just half of that of cache invalidation for an update ratio of 20%. With lower update ratios, the saving is proportionally less. We also observe that there is a

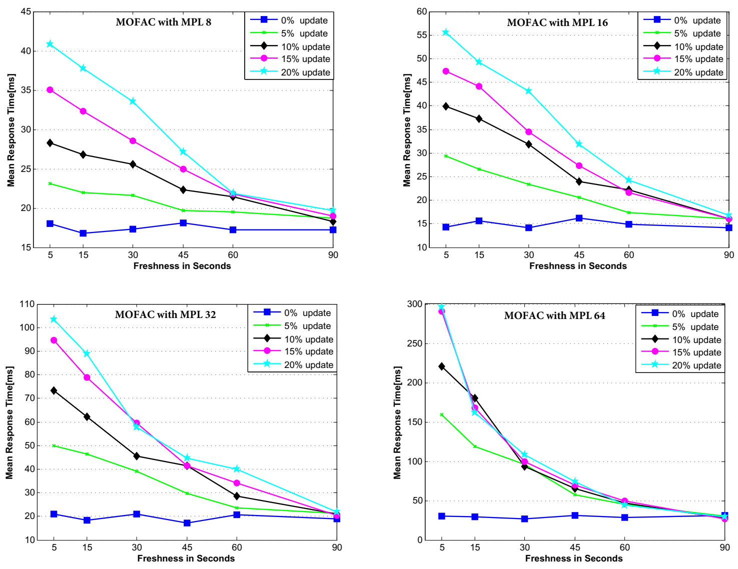


Fig. 7. Performance of MOFAC on Cluster Size 8 with MPL 8, 16, 32 and 64 varying Update Ratio and varying Freshness limit

throughput gain as well as reduced response time when we relax the freshness constraint.

From this experiment we can conclude that with multi-object freshness-aware caching, the more we relax the freshness constraint, the better the throughput and response time of the system becomes.

In the above experiments, we have seen that MOFAC can reduce mean response times and increase throughput for each update level with varying MPL. If we compare MOFAC for MPL 32 in Figure 7 with the invalidation cache algorithm results from Figure 4 we can clearly see that MOFAC performs better. At an update ratio of about 20%, the improvement in response time is about 50% with MOFAC and a (reasonable) freshness limit of 30 seconds. The higher the chosen freshness limit, the more this benefit increases since stale objects can be continue to used in the cache, thus improving throughput.

6 Related Work

Application-Tier Caching. Web caching is an attractive solution for reducing bandwidth demands, improving web server availability, and reducing network latencies. However web caching only supports static content [14]. But the dynamic nature of modern applications requires pages to be generated on the fly.

The state-of-the-art for clustered application servers is an asynchronous cache invalidation approach, as used in, e.g., the BEA WebLogic and JBoss application servers [3, 13]: When a cached object is updated in one application server node, that server multicasts a corresponding invalidation message throughout the cluster after the commit of the update transaction. Cache invalidation hence leads to more cache misses. After an update, all copies of the updated object get invalidated in the remaining cluster nodes. Hence, the next access to that object will result in a cache miss.

Earlier work [11] in this domain introduced the notion of *freshness-aware caching (FAC).* FAC tracks the freshness of cached data and allows clients to explicitly trade freshness-of-data for response times by specifying a freshness limit. However, the FAC algorithm presented in the paper [11] treats each object separately; thus, a client could place a freshness limit on the data seen, but if several objects were read then there is either a chance of high abort rates (Plain FAC) or they could be mutually inconsistent (δ-FAC). In this paper, we extend the theoretical foundations of FAC with ideas from [4] and [8], so that the new MOFAC can deal with *freshness intervals* and the *grouping* of related objects into consistency groups.

Middle-Tier Caching. Midle-tier caching approaches such as IBM's DB-Cache [1,5,10]or MTCache from Microsoft [9] are out-of-process caching research prototypes with an relatively heavy-weight SQL interface. Due to the lazy replication mechanisms, these approaches cannot guarantee distributed cache consistency, although some work around MTCache started at least specifying explicit currency and consistency constraints [8]. Our work differs in that we keep track of the freshness of data of each object separately and only notify about updates to objects that are actually modified; the remaining objects of a group still remain in their valid interval. Furthermore, MTCache works with the relational model while we are working on objects and object relational model.

Data Grid Caching and Replication. In recent years, service infrastructures for sharing large scientific datasets that are geographically distributed have been developed in the form of so-called data grids [6]. A core underlying concept of data grids is caching via adaptive data replication protocols [7]. There are three core differences to the work presented in this paper. Firstly, data grids deal with relatively static, read-mostly datasets, while we are focusing on dynamic web-based applications with frequent updates. Secondly, data grids are optimized for periodically exchanging large datasets, while our focus is on on-demand caching of individual interrelated objects. Thirdly, data grids target the data distribution problem over a wide-area network, while our proposed MOFAC algorithm assumes a closely-coupled caching system inside the same data center.

7 Conclusions and Future Work

This paper proposes a new and promising approach to distributed caching: *Multi-Object Freshness-Aware Caching* (MOFAC). MOFAC tracks the freshness of the cached data and provides clients a consistent snapshot of data with reduced response time if the client agrees to lower its data freshness expectation. MOFAC gives application developers an interesting tuning knob: The more parts of an application can tolerate (slightly) stale data, the better MOFAC can make use of the existing cache content and hence provide a better mean response times compared to cache invalidation. The choice is between performance versus data freshness.

The location within the application where this choice has to be made is application specific. In our evaluations, implemented using a MOFAC cache in the JBoss 6.0 application server, we measured savings of up-to 25% on response times, albeit in settings which may not be considered to be very realistic in the context of a real-life bookstore with lots of updates on books objects. However, even with more conservative freshness settings, such as freshness 0 for core book states and more relaxed freshness requirements for reviews, we have observed that a MOFAC cache can improve performance in the of range of 10% to 15%.

When in doubt, a developer has the choice of picking a freshness limit of 0 for requests, in which case, the proposed MOFAC algorithm behaves similar to normal cache invalidation. This makes the proposed multi-object freshness-aware caching a very attractive approach for distributed caching for dynamic web applications delivering significant performance improvements in comparison to cache invalidation, while at the same time providing actual data freshness guarantee within the constraints specified by the application.

We intend to further evaluate and study the characteristics of MOFAC and compare it with FAC and other traditional approached like cache invalidation in more complex and interesting application scenarios. We will use this to develop techniques and tools to enable application developers to choose appropriate settings that best suite the different aspects of the business object hierarchy of the application.

Acknowledgement. The authors would like to thank the members of the Database Research Group at the School of Information Technologies, University of Sydney for their support with this work. We would also like to thank the anonymous reviewers of the paper for their valuable suggestions and comments.

References

1. Altinel, M., Bornhövd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H.: Reinwald. Cache tables: Paving the way for an adaptive database cache. In: VLDB (2003)
2. Amza, C., Soundararajan, G., Cecchet, E.: Transparent caching with strong consistency in dynamic content web sites. In: Proceedings of ICS 2005 (2005)
3. BEA. BEA WebLogic Server 10.0 Documentation (2007), `edocs.bea.com`
4. Bernstein, P., Fekete, A., Guo, H., Ramakrishnan, R., Tamma, P.: Relaxed-currency serializability for middle-tier caching & replication. In: SIGMOD (2006)
5. Bornhövd, C., Altinel, M., Krishnamurthy, S., Mohan, C., Pirahesh, H., Reinwald, B.: DBCache: Middle-tier database caching for highly scalable e-business architectures. In: Proceedings of ACM SIGMOD 2003, p. 662 (2003)
6. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., Tuecke, S.: The data grid: Towards an architecture for distributed management and analysis of large scientific datasets. Journal of Network and Computer Applications 23, 187–200 (2001)
7. Chervenak, A., Schuler, R., Kesselman, C., Koranda, S., Moe, B.: Wide area data replication for scientific collaborations. In: Proceedings of 6th IEEE/ACM International Workshop on Grid Computing (Grid 2005) (November 2005)

8. Guo, H., Larson, P.-Å., Ramakrishnan, R., Goldstein, J.: Relaxed currency and consistency: How to say 'good enough' in SQL. In: SIGMOD 2004 (2004)
9. Larson, P.-Å., Goldstein, J., Zhou, J.: MTCache: Transparent mid-tier database caching in SQL Server. In: Proceedings of ICDE 2004, Boston, USA (2004)
10. Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B., Naughton, J.: Middle-tier database caching for e-business. In: SIGMOD (2002)
11. Röhm, U., Schmidt, S.: Freshness-aware caching in a cluster of J2EE application servers. In: Benatallah, B., Casati, F., Georgakopoulos, D., Bartolini, C., Sadiq, W., Godart, C. (eds.) WISE 2007. LNCS, vol. 4831, pp. 74–86. Springer, Heidelberg (2007)
12. Röhm, U., Böhm, K., Schek, H.-J., Schuldt, H.: FAS – a freshness-sensitive coordination middleware for a cluster of OLAP components. In: VLDB (2002)
13. Stark, S.: JBoss Administration and Development, 3rd edn. JBoss Group. JBoss Group (2003)
14. Wang, J.: A survey of web caching schemes for the internet. SIGCOMM Comput. Commun. Rev. 29(5), 36–46 (1999)

Appendix

Algorithm 2. Update Handling on Local Cache Node

```
input  : object identifier $o$
input  : current transaction context $tx$
output: object $o$ updated in cache and on backend database
output: update notifications broadcasted to other cache nodes

if object $o \in Cache$ then
   update $o$ in local cache node
   persist $o$ in database
   if $isGroup(o) = true$ then
      send update-notification to neighbour nodes for group key $k$
   else if $isCollection(o) = true$ then
      send update-notification to neighbour nodes for collection key $k$
   else
      send update-notification to neighbour nodes for object $o$
   end if
   commit
end if
```

Algorithm 3. Processing of Update-Notifications

```
input  : object identifier $o$
input  : update-notification timestamp $ts$
output: object's $o$ stale point is updated if present in cache

if object $o \in Cache$ then
   if $stale(o) = 0$ then
      $t_{update}(o) \leftarrow ts$ {update meta-data of cache entry $o$}
   end if
end if
```

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
@Table(name = "BOOKENTITY")
public class BookEntity implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    private int ISBN;
    private String title;
    private String description;

    @Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY,
               targetEntity = AuthorEntity.class, mappedBy = "bookEntity")
    private Collection<AuthorEntity> authors;


    @Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
               targetEntity = ReviewEntity.class, mappedBy = "bookEntity")
    public Collection<ReviewEntity> reviews;
}

@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
public class ReviewEntity implements Serializable {
    @Id
    private int id;
    private String bookReview;

    @Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
    @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "ISBN")
    private BookEntity bookEntity;
}

@Id
    private int author_id;
    String authorName;
    String authAddress;

    @Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
    @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "ISBN")
    private BookEntity bookEntity;
}
```

Listing 1.1. ORM definitions of the three entity beans of the Bookstore JEE application including the Java annotations for the cache configuration.