Freshness-Aware Caching in a Cluster of J2EE Application Servers

Uwe Röhm[1] and Sebastian Schmidt[2]

[1] The University of Sydney, Sydney NSW 2006, Australia
[2] University of Karlsruhe, Karlsruhe, Germany

Abstract. Application servers 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. The standard approaches to this problem, cache invalidation and cache replication, either do not guarantee full cache consistency or impose a performance penalty.

This paper proposes a novel approach: *Freshness-Aware Caching* (FAC). FAC tracks the freshness of cached data and allows clients to explicitly trade freshness-of-data for response times. We have implemented FAC in an open-source application server and compare its performance to cache invalidation and cache replication. The evaluation shows that both cache invalidation and FAC provide better update scalability than cache replication. We also show that FAC can provide a significant better read performance than cache invalidation in the case of frequent updates.

1 Introduction

Large e-business systems are designed as n-tier architectures: Clients access a webserver 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 tier can be easily clustered by adding more servers into the respective tier. However, there is a certain limitation to the performance and scalability of the entire system due to the single database server in the back-end. In order to alleviate this possible bottleneck, it is essential to keep the number of database calls to a minimum.

For this reason, J2EE application servers[1] have an internal EJB cache (EJB = enterprise java bean). One type of EJBs, entity beans, encapsulate the tuples stored in the back-end database. Their creation is especially costly, because their state has first to be loaded from the database. Hence, the typical caching strategy of an application server is to cache entity beans as long as possible and hence to avoid calling the database again for the same data.

In the presence of clustering, things become more complex. The separate EJB caches of the application servers in the cluster form one distributed EJB cache,

[1] This paper concentrates on the J2EE standard and entity beans; the principles of FAC are however applicable to any middle-tier application server technology.

where a copy of the same entity bean can be cached by several nodes. Distributed caching involves managing a set of independent caches so that it can be presented to the client as a single, unified cache. While queries clearly benefit from this caching at the middle-tier, updates introduce a distributed cache consistency problem. Two standard approaches to solve this problem are *cache invalidation* and *cache replication*. But both either do not guarantee full cache consistency or impose a performance penalty.

This paper presents a new approach to distributed cache management that guarantees clients the freshness and consistency of cached data: *Freshness-aware Caching (FAC)*. The core idea is that clients can explicitly ask for stale data up to a specific freshness limit. FAC keeps track of the freshness of data in all caches and provides clients with a consistent and staleness-bound view of data. Stale objects can be cached as long as they meet the clients' freshness requirements. This allows clients to trade freshness of data for improved cache utilisation, and hence shorter response times. The main contributions of this paper are as follows:

1. We present two novel freshness-aware caching algorithms, FAC and FAC-δ, that guarantee clients the freshness and consistency of cached data.
2. We give an overview of our implementation of FAC in the JBoss open-source J2EE application server.
3. We present results of a performance evaluation and quantify the impact of the different parameters of FAC on its performance and scalability.

We evaluated our approach versus JBoss' standard cache invalidation algorithm and a cache replication solution. In these experiments with varying cluster sizes up to 7 nodes, both cache invalidation and FAC provided similar update scalability much better than cache replication. It also showed that FAC can provide a significant better read performance than cache invalidation in the case of frequent updates.

The remainder of this paper is organised as follows: In the subsequent section, we give an overview of related work. We formally define freshness-aware caching in Section 3, and briefly discuss its implementation in Section 4. Section 5 presents the results of an experimental evaluation of our approach with other J2EE caching solutions. Section 6 concludes.

2 Distributed Cache Management for J2EE Clusters

State-of-the-art is an asynchronous cache invalidation approach, as used in, e.g., the Bea WebLogic and JBoss application servers [1,2]: When an entity bean is updated in one application server node, that server multicasts a corresponding invalidation message throughout the cluster after the commit of the update transaction. Due to this invalidation message, any old copy of the updated bean is removed from the other caches. JBoss sends its invalidation messages using a group communication protocol which provides guaranteed delivery [2]. Cache invalidation falls short with regard to two important aspects:

1. It cannot give any consistency guarantees for the whole distributed cluster cache, because the invalidation is done asynchronous and decoupled from the commit at the originating cache node.
2. Cache invalidation leads to more cache misses. After an update, all copies of the updated bean get invalidated in the remaining cluster nodes. Hence, the next access to that bean will result in a cache miss.

This has triggered interest on cache replication which gives strong consistency, but for the price of reduced update scalability (e.g. [3]). There are several third party in-process caching solutions for J2EE servers [4,5,6,7]. They typically provide no transparent caching for entity beans, but rather expose special caching data structures which have to be explicitly used in an J2EE application. BEA WebLogic provides in-memory replication for stateful session beans, but not for persistent entity beans which are the scope of this work [1]. Hence we did not compare to those solutions; rather, in our evaluation, we used our own cache replication framework for JBoss that is completely transparent to the application programmer and that supports entity beans [8].

In recent years, middle-tier database and query caching has gained some attention in database research [9,10,11,12]. In a nutshell, a middle-tier database cache is a local proxy DBMS that is deployed on the same machine as the application server and through which all data requests are routed. The whole database is still maintained in a central back-end DBMS server. From the client-view, the access is transparent because the middle-tier database cache has the same schema as the back-end database. But it does cache only a subset of the back-end data. Queries are answered from the local cache as often as possible, otherwise they are routed automatically to the back-end system. Updates are always executed on the back-end database, and later propagated with an asynchronous replication mechanism into the middle-tier database caches. All these approaches such as IBM's DBCache [9] or MTCache from Microsoft [11] are out-of-process caching research prototypes with an relatively heavy-weight SQL interface. Due to the lazy replication mechanisms, the solutions cannot guarantee distributed cache consistency, although some recent work around MTCache started at least specifying explicit currency and consistency constraints [13].

There are also some commercial products available: TimesTen offers a midtier caching solution built on their in-memory database manager [14]. Oracle was also offering a database cache as part of its Oracle9i Application Server [15], but has dropped support for this database cache in its latest version Oracle 10g.

3 Freshness-Aware Caching

The design goal of FAC is a fast and consistent caching solution with less cache misses than cache invalidation, and with better update scalability than cache replication. The core idea is that FAC neither immediately evicts nor replicates updated objects, but each cache keeps track of how stale its content is, and only returns data that is fresher than the freshness limits requested by clients.

3.1 Assumptions

Our approach is based on the following assumptions that represent the typical behaviour and setup of today's business applications:

Firstly, we assume a cluster of application servers with a single, shared database server in the back-end that is owned by the application servers. In other words, the database will be only accessed and modified via the application servers — no external transaction can affect the consistency of the database.

Secondly, our approach assumes an object cache inside the application server that caches all data accessed from the back-end database. Because of the exclusive database access, it is safe for application servers to cache persistent objects across several transactions. In the context of the J2EE standard, this is called *Commit Option A* when using container managed persistent (CMP). Our implementation specifically assumes CMP entity beans with Commit Option A.

Further, we assume that one client request spans exactly one transaction and no transaction spans more than one client request. A client request in our case is a method invocation issued by the client that requests or updates one or more entity bean values. We further assume that each such transaction remains local to one cluster node, i.e. we do not allow distributed transactions (transactions that are executed on more than one application server in their lifetime).

We finally assume that no bean will be updated and then read within the same transaction, i.e. we either have update or read-only transactions.

3.2 Freshness Concept

Freshness of data is a measure on how outdated (stale) a cached object is as compared to the up-to-date master copy in the database. There are several approaches to measure 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 advantages that it is independent of data types and that it does not need to access the back-end database (a value-based metric needs the up-to-date value to determine value differences). Hence, FAC uses a time-based metric, and the freshness limit is a requirement for staleness to be less than some amount.

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 $t_{stale}(o)$ *of object o is the point in time when the master copy of o got updated while the cached object itself remained unchanged.*

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

3.3 Plain FAC Algorithm

We first present a limited algorithm with exact consistency properties, called plain FAC algorithm. This guarantees clients that they always access a consistent (though potentially stale) snapshot of the cache. The algorithm keeps track of the freshness of all cached objects in the middle-tier to be able to always meet the clients' freshness requirements. Clients issue either read-only or update transactions; read-only transactions include an additional parameter, the *freshness limit*, as central QoS-parameter for FAC.

Definition 2 (Freshness-Aware Caching). Freshness-aware caching (FAC) *is an object cache that keeps metadata about the freshness of each cached object. Let* $t_{read} = (\{r(o_1), ..., r(o_n)\}, l)$ *be a read-only transaction that reads a set of objects* o_i *with freshness limit* l*. FAC gives the following guarantees to* t_{read}*:*

$$\begin{array}{rl} \text{(freshness)} & stale(o_i) \leq l \quad \mid \text{for} \quad i = 1, ..., n \\ \text{(consistency)} & stale(o_i) = stale(o_j) \mid \text{if } stale(o_i) \neq 0, \text{ for } i, j = 1, ..., n \end{array}$$

The core idea of FAC is to allow clients to trade freshness-of-data for caching performance (similar to the rationale of [16]). The staleness of data returned by FAC is bound by the client's freshness limit. The rationale is to achieve higher cache-hit ratios because stale cache entries can be used longer by FAC; if the cached objects are too stale to meet the client's freshness limit, the effect is the same as with a cache miss: the stale object is evicted from the cache and replaced by the latest state from the back-end database.

The plain FAC algorithm guarantees clients to always access a consistent (although potentially stale) snapshot of the cache. This means that for read-only transactions, all accessed objects are either freshly read from the back-end database or they have the same stale point. Update transactions only update fresh objects. If one of these consistency conditions is violated, FAC aborts the corresponding transaction.

FAC Algorithm. Our FAC algorithm extends the object caches of an application server with an additional staleness attribute: a freshness-aware cache is a set of tuples $Cache := \{(o, s)\}$, where o is a cached object and s is the *stale point* of object o, or 0 for freshly cached objects. Furthermore, to provide consistent

Algorithm 1. Freshness-aware caching algorithm.

```
function read($t_{read}$, $oid$, $l$)
if ( $\exists (o_{oid}, s) \in Cache \ : \ (now - s) > l$ ) then
    $Cache \leftarrow Cache \setminus (o_{oid}, s)$          // evict too stale $o_{oid}$
fi
if ( $\nexists (o_{oid}, s) \in Cache$ ) then
    $o_{oid} \leftarrow master(o_{oid})$                    // load from database
    $Cache \leftarrow (o_{oid}, 0)$
fi
$(o, s) \leftarrow (o_{oid}, s) \in Cache \ : \ (now - s) \leq l$
if ( $s = 0 \ \vee \ s = s_{t_{read}}$ ) then      // consistency check
    if ( $s \neq 0$ ) then $s_{t_{read}} \leftarrow s$ fi
    return $o$
else abort $t_{read}$ fi
```

access to several cached objects, FAC keeps track of the first staleness value > 0 seen be a read-only transaction ($s_{t_{read}}$). If FAC cannot guarantee that all accessed objects have the same staleness value, it aborts the request. Algorithm 1 shows the pseudo-code of our algorithm.

3.4 Delta-Consistent FAC

The plain FAC algorithm will give full consistency for cache accesses, but high numbers of updates can lead to high abort rates of readers. Following an idea from [17], we are introducing *delta-consistent FAC* to address this problem.

This approach, called FAC-δ, is to introduce an additional *drift* parameter to FAC that allows clients to read data from different snapshots within the same transaction – as long as those snapshots are within a certain range.

Definition 3 (FAC-δ). *Let $t = (\{r(o_1), ..., r(o_n)\}, l, \delta)$ be a read-only transaction that reads a set of objects o_i with a freshness-limit l and drift percentage δ ($0 \leq \delta \leq 1$); let C_{FAC} be a freshness-aware cache.* FAC-δ *is a freshness-aware caching algorithm that gives client transactions the following two guarantees:*

$$\begin{array}{rcl} \text{(freshness)} & stale(o_i) \leq l & \mid \text{for} \quad i = 1, ..., n \\ \text{(delta} - \text{consistency)} & |stale(o_i) - stale(o_j)| \leq \delta \times l & \mid \text{for } i, j = 1, ..., n \end{array}$$

Delta-consistent FAC will abort transactions if they are accessing objects that come from different snapshots which are too far apart as compared to the δ drift value (percentage of the maximum staleness). This requires only two changes to plain FAC: Firstly, our FAC-δ algorithm has to keep track of two stale-points per transaction (the oldest and the newest accessed). Secondly, line 10 of Algorithm 1 changes to check delta-consistency as defined in Definition 3.

4 Implementation

We prototypically implemented freshness-aware caching into the JBoss open source J2EE server [18] for container-managed persistent (CMP) entity beans. Our framework is totally transparent to the CMP entity beans that are deployed into the custom EJB container: there is no special caching API to be called and the whole caching behaviour can be changed just in the deployment descriptor. In the following, we give an overview of our implementation; for space reasons, we restrict the overview to the main concepts of the FAC framework.

FAC maintains *freshness* values for each cache object. Our implementation extends the J2EE container's *entity enterprise context* so that it additionally keeps a timestamp with each entity bean. This timestamp is the bean's stale point, i.e. the point in time when the cached object got updated somewhere in the cluster and hence its cached state became stale. Newly loaded objects have an empty timestamp as by definition fresh objects have no stale point. The timestamp is checked by the *freshness interceptor* to calculate the staleness of a cache object as the time difference between the current time and the stale-timestamp associated with each object.

The *freshness interceptor* is an additional trigger in the interceptor chain of the J2EE container that contains the FAC caching logic. It checks each objects freshness against the current freshness limit and evicts too stale objects before the request is executed; such objects will be automatically freshly loaded by the next interceptor in the chain (the `CachedConnectionInterceptor`). The freshness interceptor also detects state changes of an entity bean and sends those state changes to the *FAC collector*. The *FAC collector* groups all state changes of one or more entity beans in a given transaction.

When a transaction commits, this collection is forwarded to the *freshness cache manager* that informs all caches in the cluster about the stale point of the changed objects. Our implementation uses JBoss' group communication toolkit [2] for all communication between the independent caches. This toolkit guarantees atomicity and message ordering and hence offloads this functionality from our implementation. Each node checks which of the modified beans it also caches and whether they have no stale point associated with them so far, and if both is the case, sets their corresponding stale point.

5 Evaluation

In the following, we present the results of a performance evaluation of FAC versus JBoss' standard cache invalidation and our own cache replication solution [8]. We are in particular interested on the impact of the different configuration parameters on the performance and scalability of FAC.

5.1 Experimental Setup

We used a simple J2EE test application that allows us to concentrate on the application server tier to clearly identify any caching effects. The benchmark application consists of two components: a J2EE application and a test driver in the form of a standalone Java client. The application consists of two entity beans and one stateless session bean. Each entity bean has 10 integer attributes that can be both read and updated; there are corresponding getter and setter methods. The session bean provides two methods to update and to read a several beans. The back-end database consists of the two tables with 2000 tuples each.

All experiments have been conducted on an application server cluster consisting of eight nodes, each with a 3.2GHz Intel Pentium IV CPU and 2 GBytes memory under Redhat Enterprise Linux version 4. One node was used as dedicated database server running Oracle 10.1g. The application server was JBoss 4.0.2 with container managed persistence. The different tests for FAC, invalidation and replication were run using separate JBoss configurations.

The middle-tier database caching solution was TimesTen 5.1 with Oracle as back-end. In TimesTen, we configured the cache schema as one user-defined cache group with incremental auto-refresh every 5 seconds, and automatic propagation of updates from TimesTen to the back-end server at commit time. The distributed cache was managed using the JBoss invalidation method.

5.2 Evaluation of Cache-Miss Costs

First of all, we want to quantify the costs of a cache-miss at the J2EE server, and the improvements possible by using a freshness-aware cache. To do so, we measured inside JBoss the duration of accessing 1000 individual CMP-based entity beans in different system configurations:

1. cache-hit in the EJB cache of JBoss,
2. cache-miss with state loading from a local middle-tier database cache,
3. cache-miss with state loading from the remote back-end database (without any middle-tier database cache).

This gives us the costs at the middle-tier for accessing a cached entity bean or for creating a new CMP-based entity bean instance in case of a cache miss. In the case of a cache hit, the call corresponds to a lookup in the internal EJB cache and returning a pointer to an already existing bean instance. In the case of a cache miss, the requested entity bean must first be created and its state loaded from the underlying relational database.

The resulting access time distributions, as shown in Figure 1, clearly demonstrate the benefits of caching at the middle-tier. The access latency for the internal EJB cache hit is about 0.117 ms on average (median). This is factor 23 times faster than a cache-miss with loading of the state of the CMP entity bean from the remote back-end database (with a median of 2,723 ms and for a warm database cache). Moving the persistent data nearer to the application server on the same middle-tier machine reduces those cache-miss costs drastically. However, having an 'out-of-process cache-hit' in the middle-tier database cache is still about 5 times slower than a cache-hit inside the J2EE process. This is the motivation behind FAC: to improve cache usage by allowing clients to access stale cached data rather than to experience an up-to 23-times slower cache-miss.

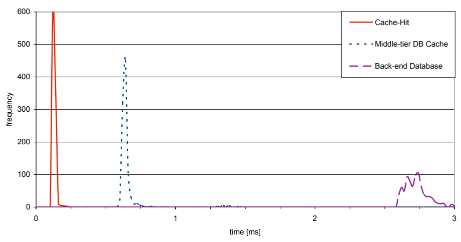
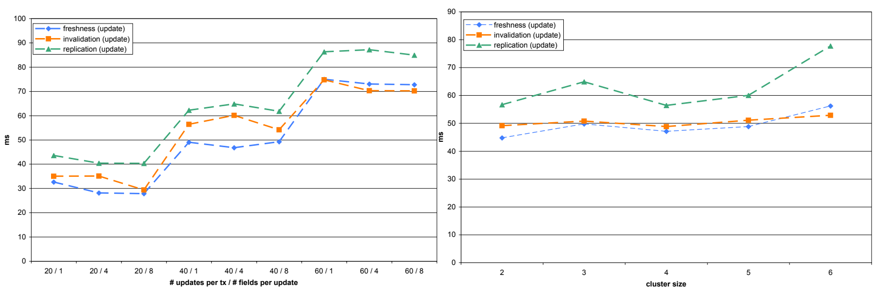


Fig. 1. Histogram of server-internal access times to access one EJB

5.3 Evaluation of Update Scalability

Next, we are interested in quantifying the scalability of FAC for updates with regard to varying update complexities and varying cluster sizes.

Influence of Update Complexity on Scalability. We start by evaluating the efficiency of FAC with regard to varying update complexity. We do so by



(a) Varying update complexity (b) Varying cluster size

Fig. 2. Influence of update complexity and cluster size on update response times

varying the number of updates per update transaction and the complexity of updates in terms of how many attributes are modified on a fixed size cluster with 3 nodes. Figure 2(a) shows the results.

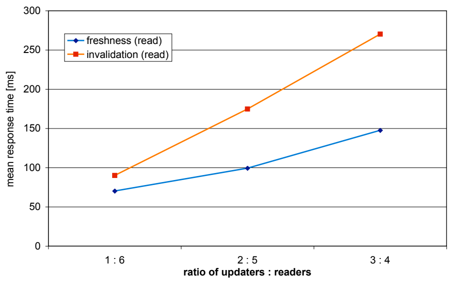
We see only a slight variance between the response times for the different number of fields for cache invalidation and FAC, as both send only one message per updated object. On the other side, replication has to send one message per field, which should show up with a higher response time for a growing number of fields. The only reason why this can not be seen in Figure 2(a) is that replication uses a hashmap to save the (field, value) pairs and the minimum size for a hashmap is 12 entries. The figure also shows higher response times for transactions with more updates. All three approaches show a linear update scalability, with cache invalidation and FAC around the same performance, while cache replication is about 15% slower.

Influence of Cluster Size on Scalability. Next, we are interested in the influence of the cluster size on the update performance of the different caching approaches. This will tell us the costs of the overhead of FAC. We ran a number of update transactions of fixed complexity consisting of 40 update operations each using a J2EE cluster of varying size between two and six nodes. The results are shown in Figure 2(b).

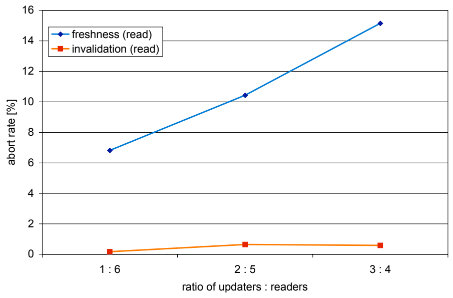
All three methods, replication, invalidation, and freshness-aware caching, show a linearly increasing update response time with increasing cluster size. Cache replication shows the slowest update performance and the worst scalability, with a 37% increase of its update time from two to six nodes. In contrast, FAC and cache invalidation show a similar update performance with a very good scalability. This is to be expected as our implementation of freshness-aware caching uses the same update-time propagation mechanism than cache invalidation.

5.4 Influence of Read/Write Ratio

In the following, we concentrate on a comparison between cache invalidation and FAC with regard to multiple concurrent read and update transactions. The next



(a) MRT with varying read/write ratios



(b) Aborts with varying r/w ratios

Fig. 3. MRT and abort rates with varying ratios of read- and update-transactions

test is designed to show the influence of higher cache hit rates in freshness-aware caching and in replication with increasing ratio of updaters to readers. This test runs on a cluster of 7 nodes and simulates 7 concurrent clients, some of which issue read-only transactions, some of which issue update transactions (from 6 readers and 1 updater to 4 readers and 3 updates). Each read-only transaction accessed 100 objects, while each update transaction modified 500 objects.

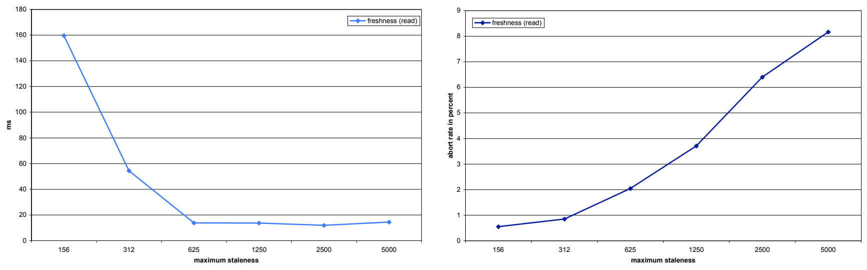
Figure 3(a) shows how the varying read-write ratios affect the cache performance. The more updaters, the slower the mean response times for readers because more cache misses occur. However, FAC is always faster than cache invalidation, in particular with higher number of updaters (up to 80% faster response time with 3 writers and 4 readers). This is due to the fact that with increasing number of updates cache invalidation evicts more and more objects from the cache. In contrast, FAC can still use many stale objects in the cache due to the freshness limits given by the readers (max staleness was 2.5s).

As Figure 3(b) shows, the performance of FAC has also a drawback: higher abort rates. With increasing number of updaters, the cache contains objects with more different staleness values. This increases the chance, that FAC cannot provide a client with a consistent set of stale objects from the cache and hence must abort the reader. Cache invalidation is not affected by this, and consequently shows much lower abort rates due to conflicting reads and writes.

5.5 Influence of Freshness Limit

Next, we want to explore the effect of trading freshness of data for read performance. The experiment was done on a cluster of six nodes and with 60 concurrent clients consisting of 85% readers and 15% updaters. Each read-only transaction was accessing 10 objects, while each update transaction was changing 50 objects. We varied the freshness limit of the readers and measured the effect of the different freshness limits on the mean response times for readers.

The results in Figure 4(a) show how applications which tolerate a higher staleness of the cached data can yield faster response times. There is however a



(a) MRT with varying freshness limit (b) Abort rates vs. freshness limit.

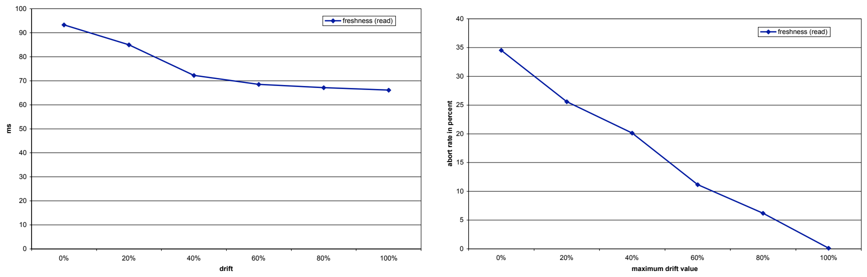
Fig. 4. Mean response times and abort rates with varying freshness limits

limitating effect on the performance by the number of aborts due to inconsistent objects within the same read transaction. This is illustrated in Figure 4(b).

The more readers accept to access outdated data, the higher is the probability that they access objects of different staleness within the same transaction. If those different staleness values exceed the drift value tolerated by the application, the corresponding transaction is aborted. In our case, we configured a maximum drift value of 80%. As shown in Figure 4(b), the abort rate increases from below 1% up to 8% with increasing staleness value. From a maximum staleness of about 625ms onwards, the number of aborted readers exceeds 2% and the throughput in the system actually decreases from now on.

5.6 Influence of Drift Constraint

The previous experiments showed that the costs of lower response times via wider freshness limits are more aborts. In Section 3.4, we had hence introduced FAC-δ that does only provide delta consistency to clients. In the last experiment, we are now investigating the influence of the drift constraint on the read performance with FAC-δ. The following test uses again a cluster of seven nodes and simulates 60 concurrent clients, of which 15% are updaters. Each read-only transaction was accessing 100 objects, while each update transaction was changing 500 objects.



(a) MRT with varying drift values. (b) Abort rate wth varying drift values.

Fig. 5. Mean response times and abort rates with varying drift values

The results in Figure 5 show that the drift parameter of FAC-δ has the desired performance effect: larger drift values result in lower abort rates and subsequently to faster mean response times. A drift value of 0% represents the basic FAC algorithm, from where the abort rate decreases linearly with increasing drift percentage (cf. Figure 5(b)). With a drift of 100, the abort rate is finally zero, as reads are allowed to read any stale data below the freshness limit.

6 Conclusions

This paper presented a novel approach to caching in a cluster of application servers: *freshness-aware caching* (FAC). FAC tracks the freshness of cached data and allows clients to explicitly trade freshness-of-data for response times. We have implemented FAC in the JBoss open-source application server. In an experimental evaluation, we studied the influence of the different parameters of FAC on its performance and scalability.

It showed that there are significant performance gains possible by allowing access to stale data and hence avoid cache misses. However, because FAC also guarantees clients a consistent cache snapshot, higher freshness limits increase the abort rates. We addressed by introducing *delta-consistent FAC* that allows clients to tolerate stale data from different snapshots within a maximum delta. When comparing our approach with JBoss' standard cache invalidation, it showed that both have a similar scalability and update performance. However, with higher read-write ratios, FAC clearly outperforms cache invalidation.

We are currently working on extending FAC to support consistency constraints to provide scalable full consistency for multi-object requests.

References

1. BEA: BEA WebLogic Server 10.0 Documentation (2007), `edocs.bea.com`
2. Stark, S.: JBoss Administration and Development. JBoss Group, 3rd edn. (2003)
3. Wu, H., Kemme, B., Maverick, V.: Eager replication for stateful J2EE servers. In: Proceedings of DOA2004, Cyprus, pp. 1376–1394 (October 25-29, 2004)
4. JBoss Cache: A replicated transactional cache, `http://labs.jboss.com/jbosscache/`
5. SwarmCache: Cluster-aware caching for java, `swarmcache.sourceforge.net`
6. ehcache: ehcache project (2007), `ehcache.sourceforge.net`
7. Progress: DataXtend CE (2007), `http://www.progress.com/dataxtend/`
8. Hsu, C.C.: Distributed cache replication framework for middle-tier data caching. Master's thesis, University of Sydney, School of IT, Australia (2004)
9. Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B., Naughton, J.: Middle-tier database caching for e-business. In: SIGMOD (2002)
10. Altinel, M., Bornhövd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., Reinwald,: Cache tables: Paving the way for an adaptive database cache. In: VLDB (2003)
11. Larson, P.Å., Goldstein, J., Zhou, J.: MTCache: Transparent mid-tier database caching in SQL Server. In: Proceedings of ICDE2004, pp. 177–189. Boston, USA (2004)

12. Amza, C., Soundararajan, G., Cecchet, E.: Transparent caching with strong consistency in dynamic content web sites. In: Proceedings of ICS 2005, pp. 264–273 (2005)
13. Guo, H., Larson, P.Å., Ramakrishnan, R., Goldstein, J.: Relaxed currency and consistency: How to say 'good enough' in SQL. In: SIGMOD 2004, pp. 815–826 (2004)
14. TimesTen Team: Mid-tier caching: The TimesTen approach. In: Proceedings of ACM SIGMOD 2002, pp. 588–593 (June 3-6, 2002)
15. Oracle: Oracle 9i application server: Database cache. White paper (2001)
16. Röhm, U., Böhm, K., Schek, H.J., Schuldt, H.: FAS – a freshness-sensitive coordination middleware for a cluster of OLAP components. In: VLDB, pp. 754–765 (2002)
17. Bernstein, P., Fekete, A., Guo, H., Ramakrishnan, R., Tamma, P.: Relaxed-currency serializability for middle-tier caching & replication. In: SIGMOD (2006)
18. JBoss Group: JBoss (2007), http://www.jboss.org