Method-Based Caching in Multi-Tiered Server Applications

[Technical Report]

Daniel Pfeifer and Hannes Jakschitsch
Institute for Program Structures and Data Organisation (IPD)
Universität Karlsruhe, Germany
{pfeifer, s_jaksch}@ira.uka.de

ABSTRACT

In recent years, application server technology has become very popular for building complex but mission-critical systems such as Web-based E-Commerce applications. However, the resulting solutions tend to suffer from serious performance and scalability bottlenecks, because of their distributed nature and their various software layers. This paper deals with the problem by presenting an approach about transparently caching results of a service interface's read-only methods on the client side. Cache consistency is provided by a descriptive cache invalidation model which may be specified by an application programmer. As the cache layer is transparent to the server as well as to the client code, it can be integrated with relatively low effort even in systems that have already been implemented.

Experimental results show that the approach is very effective in improving a server's response times and its transactional throughput. Roughly speaking, the overhead for cache maintenance is small when compared to the cost for method invocations on the server side. The cache's performance improvements are dominated by the fraction of read method invocations and the cache hit rate. Our experiments are based on a realistic E-commerce Web site scenario and site user behaviour is emulated in an authentic way. By inserting our cache, the maximum user request throughput of the web application could be more than doubled while its response time (such as perceived by a web client) was kept at a very low level.

Moreover, the cache can be smoothly integrated with traditional caching strategies acting on other system tiers (e.g. caching of dynamic Web pages on a Web server). The presented approach as well as the related implementation are not restricted to application server scenarios but may be applied to any kind of interface-based software layers.

Categories and Subject Descriptors

H.3 [**Information Systems**]: Information Storage and Retrieval; D.1.5 [**Software**]: Object-oriented Programming; C.4 [**Performance of Systems**]: Optimization

General Terms

Information Systems, Architecture, Design, Optimization, Experimentation

Keywords

Caching, Application Server, EJB, Object-Oriented, Web-Application, Performance, Scalability

1. INTRODUCTION

In recent years, application server technology has become very popular for building complex but mission-critical systems. One the of most important standards in this context is the Java 2 Enterprise Edition platform by Sun (J2EE, [32]), including Enterprise Java Beans (EJB, [31]) and Java Server Pages (JSP, [33]) as major building blocks. A common use case for this technology is the development of database-driven e-business Web sites and Web applications in general. Although the resulting systems tend to have a clean architecture of well separated components, they often suffer from serious performance and scalability problems. A major reason for this is that total system functionality is scattered amongst various software layers (or tiers) on potentially different machines (e.g. a Web server, an EJB-based application server and a database server).

Industry and research have developed mechanisms to overcome these problems by stating design patterns and implementation tricks as well as providing sophisticated data caching techniques. On the one hand, design patterns have mostly focused on the design and the interfaces of components residing at an (EJB-based) application server. Caching techniques, on the other hand, have only dealt with the top and the bottom tiers of Web applications: *Web caches* usually cache entire Web pages (or at least fragments of pages) such as served by a related Web application. *Application data caches* store data that is sent to an application server as a result of database queries.

In contrast, the approach of this paper increases system performance by *caching results of method calls* as they occur for example when invoking EJB-methods from servlets running on a Web server. The related *method cache* caches the results on the application server's client side (which is a Web server for the case of Web applications). Thus, it differs from conventional caches who deal with caching attribute values of data objects.

In order to maintain cache consistency, we expect an application developer to create a so called *cache model*. The cache model states read-write dependencies between the methods exposed by the application server. Results of methods that only read data at the application server side (no state changes) may be cached and will be available for potential cache hits. Write-methods (which may change application server states) are always delegated to the application server.

As opposed to conventional software components, a part of the classes representing a method cache are automatically *generated* from the cache model. This way, the cache implements the interface classes exposed by the application server. The generated classes may be used on the system's client side. Still, it remains transparent to the client code that it actually invokes methods from the method cache rather than from the application server itself.

The rest of this paper is organized as follows: Section 2 highlights the overall architecture of a method cache differentiating runtime and generation time aspects. Section 3 formally introduces cache models which help to provide cache consistency. It also gives a sample cache model such as processed by our implementation and it describes the data structures and interfaces that are used for runtime cache maintenance. Section 4 discusses further issues of method caching, e.g. cache size and cache bypassing. Experiments concerning the overall cache performance are presented in Section 5. We then compare our contribution with related work and explain how a method cache can be integrated in a modern Web application architecture (Section 6). In this context, we propose a combination of dynamic Web page caching and method-based caching where invalidation events for cached Web pages are triggered by the method cache. The paper closes with a conclusion and an outlook on future work in Section 7.

2. GENERAL CACHE ARCHITECTURE

This section highlights the general architecture of the method cache system and how it integrates into application server systems. We distinguish between runtime aspects, that relate to the time when the cache is actually used, and generation time aspects.

2.1 Runtime Aspects

Figure 1 gives an illustration of an application server system's client and server part: The application server component exposes an object-oriented interface consisting of a set of abstract classes which hold a set of abstract methods. The client knows about these classes, their methods, and the related invocation protocols. It invokes the methods exposed by the server and receives the corresponding results. The server internally keeps implementation code for executing those methods.

Depending on a system's overall architecture, the calls may be remote or in process. Furthermore, one can distinguish between method implementations that *never* alter the internal state of the application server (or its dependent sub-

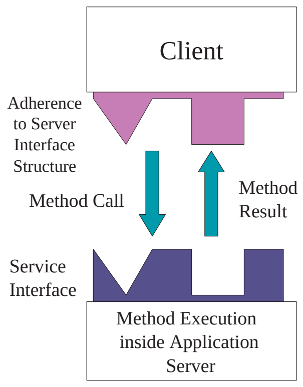
Client
Adherence to Server Interface Structure
Method Call
Method Result
Service Interface
Method Execution inside Application Server

Figure 1: Abstract interaction scheme for application server and client.

systems) and methods that potentially do. In this paper, we will refer to them as *read methods* and *write methods* respectively.

Figure 2 shows how the system's abstract structure is changed when introducing a *method cache*. The latter is located in between the client and server and exactly repeats the server's service interface. Method calls from the client that formerly addressed the application server are now received by the cache component. The cache component performs different actions depending on whether the called method is a read or a write method.[1]

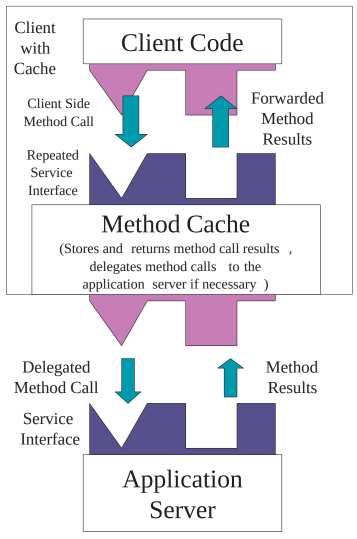

Figure 2: Interaction of application server and client when using a method cache.

If it is a read method, the component tries to look up a cached method result that has been stored in the cache during a former method call. In order to guarantee the result's correctness the original call must have been performed on an

[1]The method's type (read or write) is determined by means of a related cache model whose structure is described in Section 3.

identical application server object and a list of equal method arguments.[2] Hence, a method's signature, its arguments and the invocation-object act as a cache key.

If the key's lookup succeeds, there is a cache hit and the stored result is immediately returned to the client. Otherwise the call is delegated to the application server. The result from the application server is then cached and associated with the cache key. Eventually the result is returned to the client.

In case of a write method's invocation, the cache must invalidate all cached read method results that potentially depend on the state-altering effects of the write method call at the application server. The related cache entries are determined by the argument values of the method call and the related cache model. After the entries are deleted from the cache, the write method call is delegated to the application server and the result is propagated to the client.

2.2 Structure of a Method Cache

Figure 3 illustrates a simplified version of the cache's internal class structure using the UML notation: the generated cache classes implement the service interface such as exposed by the application server. They contain code for looking up cached method results and delegating calls to the application server as well as code for result invalidation.

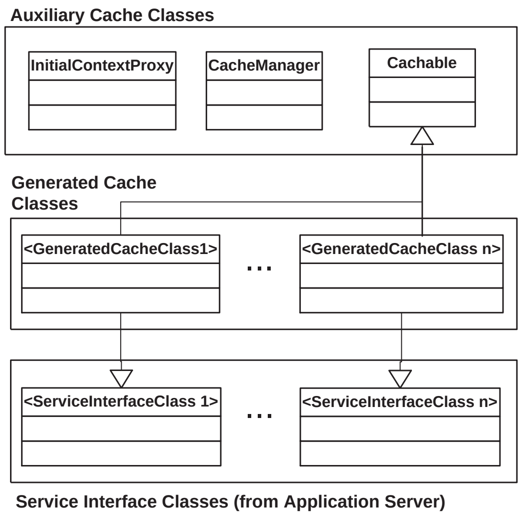

Figure 3: Simplified class structure of a method cache.

Instances of generated cache classes act as proxy objects for application server objects (e.g. EJBs) on the application server side. If the cache is activated, the client code keeps references (or handles) to proxy objects from the cache instead of handles to application server objects. However, there is a unique mapping from proxy objects to application server objects and internally every proxy object carries a handle to its related application server object. The handle is required when delegating corresponding method calls to the application server (e.g. because of a cache miss). The cache's auxiliary classes provide functionality that is common to all generated classes. This includes mapping of application objects to proxy objects, storing method results that need to be cached as well as keeping validity and performance information for the cache.

[2]Equality tests for method arguments can be customized. By default, they are based on standard comparison methods for the respective programming language (e.g. `java.lang.Object.equals()` for Java).

The class `InitialContextProxy` is specific to the cache's usage in the EJB context: it mimics the functionality of the J2EE class `javax.naming.InitialContext` and therefore is in charge of looking up the initial handles of application server objects on the client side. However, in contrast to the original class `javax.naming.InitialContext`, the substitute does not return handles to application server objects but to their related proxy objects from the cache. Thus, if one makes the client code access `InitialContextProxy` instead of `javax.naming.InitialContext`, all client-server interaction will be seamlessly redirected to go through the method cache.

2.3 Generation Time Aspects

At generation time, a generator tool takes the service interface structure and a related cache model as its input and writes out source files for generated cache classes. After the latter ones have been compiled, they can be used in the cache's runtime environment.

The related data flow of our implementation is shown in Figure 4: The service interface is provided as a set of compiled Java interfaces whose method structure is accessed through the Java Reflection API. The cache model is represented by an XML file whose formal semantics will be discussed in Section 3.

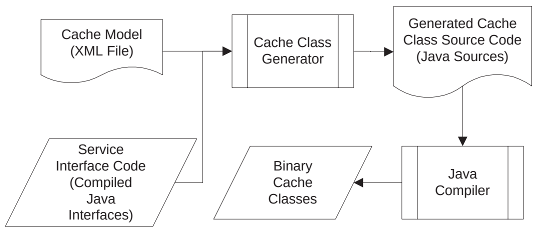
Cache Model (XML File)
Cache Class Generator
Generated Cache Class Source Code (Java Sources)
Service Interface Code (Compiled Java Interfaces)
Binary Cache Classes
Java Compiler

Figure 4: Data flow for the generation of cache classes.

3. CACHE INVALIDATION

In order to guarantee the validity of cached method results, we have designed a structure for specifying cache models. In general, a cache model expresses under what circumstances a cache value must be invalidated because it might not be consistent with the state on the application server side anymore.

As our cache strategy is based on methods whose implementations and data dependencies are unknown and potentially difficult (or impossible) to automatically analyze, we expect an application programmer to provide abstract information about data dependencies of methods as part of a cache model.

This section formally introduces the kind of method dependencies which must be taken into account. Then the semantics and the qualities of the corresponding cache models are discussed. We give a short example on how those models are realized in our system. Lastly, we discuss the data structures and interfaces that are required to efficiently realize our invalidation approach in practice.

3.1 Method Dependencies

In this section, we give a simple but abstract definition of service interfaces and read-write dependencies. In order to focus on aspects that are relevant for cache models, we have avoided many details such as method signatures, method implementations or a related type system.

Definition 1: Let $S = \{s_1, s_2, \ldots\}$ be a countable set of server states and $P = \{p_1, p_2, \ldots\}$ be a countable set of potential method arguments or results. A *service interface* is a finite set of methods $M = \{m_1, \ldots, m_n\}$, where an m_i is a computable function $m_i : S \times P \to S \times P, (s_{bef}, p_{arg}) \mapsto (s_{aft}, p_{res})$.

In other words, a method maps an initial server state (s_{bef}) to a resulting server state (s_{aft}), while accepting a set of arguments (p_{arg}) and returning a result (p_{res}).

Definition 2: A *read method* is a method $m_i \in M$ such that $\forall s \in S, \forall p \in P : \exists p' \in P : m_i(s,p) = (s,p')$. A *write method* $m_j \in M$ is a method which is not a read method.

Definition 3: A *dependency* $dep(m_i, p_k,\ m_j, p_l)$ exists iff m_i is a read method and $\exists s, s' \in S : \exists p'_l, p'_k, p''_k \in P : m_j(s, p_l) = (s', p'_l) \wedge m_i(s, p_k) = (s, p'_k) \wedge\ m_i(s', p_k) = (s', p''_k) \wedge p'_k \neq p''_k$.

So, a read method invocation (m_i, p_k) depends on a write method invocation (m_j, p_l), if and only if the write method invocation alters the read method invocation's result for some server state. As read methods don't change server states, there are no dependencies between read methods.

3.2 Cache Models

Cache models must enable dependencies of method calls to be determined on the client side. Unfortunately, computing those dependencies on the client side, would involve a simulation or an execution of the related service method calls in the same place. This cannot be done because the related computation would be too expensive and, also, it would corrupt the entire approach of a client server system. Instead, the formalism proposed below allows for *estimating* a method call's read-write dependencies on the client side. This is done by abstracting from the actual read-write dependencies and replacing them by so-called *model dependencies*.

The central structure for defining model dependencies is a collection of *abstract indexes*. Abstract indexes are sets which help to model invalidation dependencies between read and write methods. If a read and a write method call access the same elements of the same index, it is taken as an indication of a potential dependency between the two calls. At cache runtime, the read method call's cached result will be invalidated, if the write method call writes the corresponding index element. The indexes are abstract in a sense that they may but do not have to represent real data dependencies such as caused by a service method's implementations. Thus, their major purpose is to provide for a consistent but light-weight invalidation of cached read method results on the client side.

The number of indexes and the way their elements are accessed must be defined by the cache model designer: For every service method the designer has to specify a set of indexes and a corresponding number of so called *index functions*. An index function's argument is based on the arguments (and the result) of a service method call and an index function's result represents an index element. The cache model designer also declares whether a service method is a read or a write method. When a service method is invoked on the client side, its arguments are well known and the respective index functions can be evaluated. The evaluations just return the index elements which are needed for tracking invalidation dependencies between read and write method calls.

The following definitions formalize the related concepts:

Definition 4: Let $IFun$ be a finite set of computable index functions, where each element $f \in IFun$ has the following signature: $f : P \to \mathbb{N}$. Then, a *cache model mod* is a function $mod : M \to \{\mathtt{r}, \mathtt{w}\} \times \wp(\{1, \ldots, k\} \times (IFun \cup \{\mathtt{all}\}))$ (M as in Definition 1).[3]

Here, the number of indexes is represented by k. A function $f \in IFun$ states what index elements are accessed by a method call (m, p) (with $m \in M$ and $p \in P$). In order to keep the model simple, a method call may either access a single index element or all elements of an index. The latter case is indicated by the special function `all`. The values `r` and `w` express whether a method performs read or write access.

Definition 5: A *model read method* is a method $m \in M$ where: $mod(m)(p) = (\mathtt{r}, ifs)$ with some index function set $ifs \in \wp(\{1, \ldots, k\} \times (IFun \cup \{\mathtt{all}\}))$. Otherwise it is a *model write method*. Furthermore, $mrm(mod)$ represents the set of model read methods for the cache model *mod*. More formally: $mrm(mod) = \{m \in M \mid mod(m).1 = \mathtt{r}\}$[4]

Definition 6: A *model dependency* $moddep(m, p, m', p')$ exists iff m is a model read method such that $mod(m) = (\mathtt{r}, ifs)$, m' is a model write method such that $mod(m') = (\mathtt{w}, ifs)$ and $\exists i \in \{1, \ldots, k\} : \exists f, f' \in IFun \cup \mathtt{all} : (i, f) \in ifs \wedge (i, f') \in ifs' \wedge (f = \mathtt{all} \vee f' = \mathtt{all} \vee f(p) = f'(p'))$.

The following model is called the *trivial model* mod_{triv} (for a given M) as it assumes that every method is a model write method: $k := 0, \forall m \in M : m \mapsto (\mathtt{w}, \emptyset)$.

All model dependencies for a model *mod* form a relation $moddep \subseteq M \times P \times M \times P$. For the trivial model we have $moddep_{triv} = \emptyset$ and also $mrm(mod_{triv}) = \emptyset$.

[3] $\wp(x)$ specifies the powerset of a set x.

[4] $.1$ denotes the selection of the first element of an n-tuple.

Definition 7: A cache model is *correct* iff every model read method is a read method and $\forall m, m' \in M, \forall p, p' \in P : m \in mrm(mod) \Rightarrow (dep(m, p, m', p') \Rightarrow moddep(m, p, m', p'))$.

Correctness only requires that dependencies are indicated by model dependencies for those methods who are model read methods. However, a cache model may cause a "false alarm" and invalidate a cached method result, although the corresponding method invocation on the server would still return the same value. For example, one may declare a read method as a model write method and state model dependencies between this read method and other read methods.

Obviously, mod_{triv} is a correct cache model because it contains no model read methods at all.

Lemma (Correctness of Cache Results): Let $m \in M$ be a model read method and *seq* be a sequence of (consecutive) method invocations of the following form: $seq = (m(s,p) \mapsto (s, res) = (s_{j_1}, res), m_{l_1}(s_{j_1}, p_{m_1}) \mapsto (s_{j_2}, p'_{m_1}), m_{l_2}(s_{j_2}, p_{m_2}) \mapsto (s_{j_3}, p'_{m_2}), \ldots, m_{l_n}(s_{j_n}, p_{m_n}) \mapsto (s', p'_{m_n}), m(s', p) \mapsto (s', res'))$. Then, if *mod* is a correct model for M and $\forall k \in \{1, \ldots, n\} : \neg moddep(m, p, m_{l_k}, p_{m_k})$ then $res = res'$. (The related proof is straight forward by induction on n.)

Definition 8: Let mod_1 and mod_2 be two correct cache models for the same set of methods M and $moddep_1$, $moddep_2$ the respective model dependencies. Then, mod_1 is *more precise* than mod_2 iff $mrm(mod_2)$ is a proper subset of $mrm(mod_1)$ or, both sets of model read methods are equal and $moddep_1$ is a proper subset of $moddep_2$. More formally: $mrm(mod_2) \subsetneq mrm(mod_1) \vee (mrm(mod_1) = mrm(mod_2) \wedge moddep_1 \subsetneq moddep_2)$.

According to this definition, if the two cache models mod_1 and mod_2 hold the same set of model read methods, mod_1 is more precise because it better models when method calls are independent of each other. On the other hand, if mod_1 contains more model read methods than mod_2, it is more precise than mod_2, since it allows for a larger set of methods to be cached. Note that for correct cache models, the number of model read methods ranges from zero (for the trivial model) to the total number read methods of M.

Precision is a semi-ordered relation. Obviously, the trivial model is a lower bound for precision. *dep* from Definition 3 is the unique upper bound for precision and, in general, it is different from any correct cache model for a given service interface. A useful cache model should be more precise than the trivial model. However, if a cache model is too precise, it might become expensive to compute the related dependencies.

3.3 Example

This section demonstrates how the proposed cache models are realized in practice. In essence, the presented implementation corresponds to the formalism from above. One important difference is that abstract indexes can be named and are not just represented by numbers (such as in Definition 4). Moreover, elements of abstract indexes are now represented by Java objects instead of natural numbers.

The Java pseudo code in Figure 5 represents an extract of a service interface for subscribers holding a list of subscriptions. Here, a subscriber is identified by an ID while the subscription is identified by the corresponding document title. For a subscriber new subscriptions may be added, or existing ones removed. The list of subscribers may be retrieved from a subscription. Also, one may read the list of subscriptions for a certain subscriber.

```
public interface Subscription {
  String getTitle();
  Vector getSubscriberList();
}

public interface Subscriber {
  String getID();
  Vector getSubscriptionList();
  void addSubscription(String title);
  void removeSubscription(String title);
}
```

Figure 5: Java interface pseudo code for a subscription service.

An XML-based cache model that deals with the given service interface is presented in Figure 6. It defines two indexes named `Subscription` and `Subscriber` (Lines 2 and 3), which correspond to index numbers $k = 1$ and $k = 2$.

Every interface method is annotated with mappings that correspond to elements of function *mod* from Definition 4. E.g. every interface method (identified by its name) is declared as either a read or a write method (attribute `access`).[5] The `model`-tag states on what indexes the corresponding method acts (attribute `index`). The attribute values for `ifun` specify index functions as Java code fragments. A method may read or write all elements of an index (specified by `ifun="all"`) or just a single element. In the latter case the corresponding element is computed at runtime by executing a Java expression which must be stated as `ifun`'s attribute value. The expression must be functional and may refer to the annotated method's `this`-object (by using the keyword `$this$`), its result (by using the keyword `$result$`) or its parameter variables. (The latter two options are not demonstrated in the example.) As mentioned before, index elements may be arbitrary Java objects that result from the expression's evaluation. In the example below `$this$.getID()` is used to specify elements of the `Subscriber`-index as ID-Strings. The use of objects instead of natural numbers is sufficient because index elements only need to be checked for equality (see Definition 6). At cache generation time, the code fragments will be embedded in the generated cache classes.

In the following paragraph, the model's correctness is briefly discussed on an intuitive level: basically, the two given indexes represent the list of `Subscriber` and `Subscription` objects that exist on the server side (Lines 2 and 3). As those objects are uniquely identified by their ID or their re-

[5]In the sample XML file, methods are identified by their unique names. However, there are additional mechanisms for identifying methods, whenever their names are not unique because of overloading.

```
1     <cachemodel>
2      <index name="Subscription"/>
3      <index name="Subscriber"/>
4
5      <interface name="Subscription">
6       <method name="getTitle" access="r" >
7        <model index="Subscription"
8               ifun="$this$.getTitle()"/>
9       </method>
10      <method name="getSubscriberList" access="r"
11              mapstrategy="collection"
12              containedclass="Subscriber">
13       <model index="Subscriber" ifun="all"/>
14      </method>
15     </interface>
16
17     <interface name="Subscriber">
18      <method name="getID" access="r">
19       <model index="Subscriber"
20              ifun="$this$.getID()"/>
21      </method>
22      <method name="getSubscriptionList" access="r"
23              mapstrategy="collection"
24              containedclass="Subscription">
25       <model index="Subscriber"
26              ifun="$this$.getID()"/>
27      </method>
28      <method name="addSubscription" access="w">
29       <model index="Subscriber"
30              ifun="$this$.getID()"/>
31      </method>
32      <method name="removeSubscription" access="w">
33       <model index="Subscriber"
34              ifun="$this$.getID()"/>
35      </method>
36     </interface>
37    </cachemodel>
```

Figure 6: Cache model for the subscription service in XML.

spective title, the applied index functions `$this$.getID()` and `$this$.getTitle()` simply return those values. The model tries to reflect changes that occur on the extent of the `Subscriber` and the `Subscription` class: e.g., the annotation for method `addSubscription()` states that adding a subscription writes the subscriber object from which `addSubscription()` was invoked (Lines 26, 27). The result of method `getSubscriberList()` changes, if the corresponding `Subscription`-object is added to or removed from a `Subscriber` using `addSubscription()` or `removeSubscription()`. As this could happen to any subscriber, `getSubscriberList()` is invalidated whenever a subscriber is written (Lines 22, 26 and 30). Note that every subscription list object is considered as a component of a subscriber object and the list object's state forms a part of a corresponding subscriber object's state.

The use of the attributes `mapstrategy` and `containedclass` will be explained in Section 4.1.

3.4 Interfaces and Data Structures for Cache Invalidation

This section discusses data structures that are used in our implementation for efficient cache management and invalidation. Also, we highlight the interface methods that are used for cache management and how they are accessed from generated cache classes.

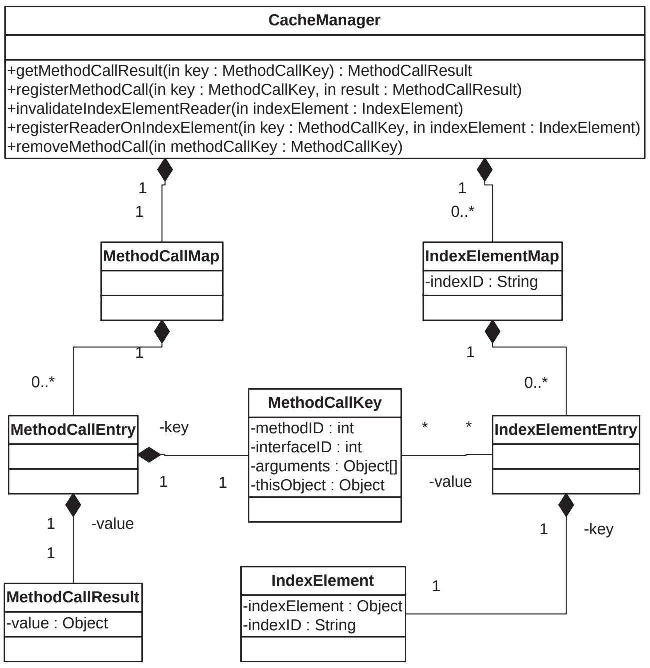
CacheManager
+getMethodCallResult(in key : MethodCallKey) : MethodCallResult
+registerMethodCall(in key : MethodCallKey, in result : MethodCallResult)
+invalidateIndexElementReader(in indexElement : IndexElement)
+registerReaderOnIndexElement(in key : MethodCallKey, in indexElement : IndexElement)
+removeMethodCall(in methodCallKey : MethodCallKey)
1
1
1
0..*
MethodCallMap
IndexElementMap
-indexID : String
1
1
0..*
0..*
MethodCallKey
-methodID : int
-interfaceID : int
-arguments : Object[]
-thisObject : Object
MethodCallEntry
-key
1
1
*
*
-value
IndexElementEntry
1
-value
1
1
-key
MethodCallResult
-value : Object
IndexElement
-indexElement : Object
-indexID : String
1

Figure 7: Data structures for efficient cache invalidation.

In Figure 7 the UML notation is used to depict a simplified version of the cache manager's interface and its data structures for storing cache content. The related `CacheManager` has already been mentioned in Section 2. It offers five methods that are crucial for runtime cache management:

`getMethodCallResult()` tries to lookup the result of cached read method call from the cache's store. In order to do so, it accepts an instance of the class `MethodCallKey` which represents a method call. For this reason, a related `MethodCallKey` object holds IDs to identify the method that has been invoked. The `interfaceID` attribute refers to the method's interface and `methodID` identifies the method itself (within the respective interface). Moreover, the attributes `thisObject` and `arguments` store the `this`-object on which the method was invoked and respectively the method call's argument list.

`getMethodCallResult()` uses an instance of the class `MethodCallMap` to lookup a cached method result. The corresponding `MethodCallMap` instance belongs to the cache manager and is implemented as a hash table whose keys are `MethodCallKey` objects and whose values are `MethodCallResult` objects. `getMethodCallResult()` returns the `MethodCallResult` object that is bound to a method call key. If none is found, it returns `null`.

`registerMethodCall()` stores pairs of `MethodCallKey` and `MethodCallResult` objects in the cache manager's `MethodCallMap` hash table.

In order to state that a stored method call depends on a certain index element, one has to call `registerReaderOnIndexElement()`. The method accepts a method call which is represented by a `MethodCallKey` object and associates it with an index element of class `IndexElement`. An `IndexElement` object simply holds the index to which it refers (attribute `indexID`) and the index element itself (attribute `indexElement`). The cache manager has to store this information to be able to invalidate the respective method call at a later point of time. For an efficient lookup at invalidation time, an index element and the method call keys that depend on the index element are stored in a hash table of class `IndexElementMap`. Thereby, the index element is used as the key and the set of `MethodCallKey` objects represents the respective value. Note that a set is required since many read method calls might depend on a single index element.

The cache manager keeps several index elements maps – one for each cache index. The reason for this that an invalidation that is based on the keyword `all` from Section 6 invalidates all read method calls that are associated with a cache index. In order to delete the corresponding read method calls, the system simply operates on all entries of the `IndexElementMap` object for the respective cache index.

`invalidateIndexElementReader()` provides the task of invalidating method call results who are associated with a certain cache index (because of a previous call of `registerReaderOnIndexElement()`). `invalidateIndexElementReader()` takes an `IndexElement` object as an argument and looks up the respective entry in the corresponding index element map. It iterates over the list of method call keys that might be bound to the index element and removes the corresponding `MethodCallKey` objects and its values from the `MethodCallMap` hash table. It also removes the method call key from the index element map itself.

`removeMethodCall()` is required to implement replacement strategies for the cache. After passing in a `MethodCallKey` object, the method deletes all entries referring to the method call key. `removeMethodCall()` first looks up the respective method call entry in the `MethodCallMap` table and deletes it from there. With the entry it also finds the stored `MethodCallKey` object which holds references to all related `IndexElement` objects (via the `value` association from Figure 7). Based on these references, the corresponding index element entries can be updated or deleted from the `IndexElementMap` tables.

Figure 8 shows the Java pseudo code of a generated cache class for the `Subscriber` interface from Section 3.3. The code clarifies the use of cache manager's methods such as discussed above. Taking the inserted comments into account it should be self-explaining.

```
public class CachableSubscriber implements Subscriber ... {
  private Subscriber delegate;
  private CacheManager manager;
  ...
  // Example of a write method.
  public void addSubscription(String title) {
    // Delegate call to server.
    delegate.addSubscription(title);
    // Invalidate dependent read method calls.
    manager.invalidateIndexElementReaders("Subscriber",
    // The respective index element is computed here.
                                          this.getID());
  }

  // Example of a read method.
  public Vector getSubscriptionList() {
    // Compute the method call key for this method call.
    // (The argument list is empty and the respective
    //  interface and method ID are assumed to be 1.)
    MethodCallKey key =
      new MethodCallKey(1, 1, delegate, new Object[]{});
    // Try to get the result from cache and return it.
    MethodCallResult result =
      manager.getMethodCallResult(key);
    if (result != null) return (Vector) result.value;
    // Delegate call in case of a cache miss.
    Vector value = (Vector) delegate.getSubscriptionList();
    ...
    // Register the result.
    manager.
      registerMethodCallResult(key,
                               new MethodCallResult(value));
    // Computer and register the index element for
    // invalidation.
    manager.
      registerReaderOnIndex(key,
                            new IndexElement("Subscriber",
    // The respective index element is computed here.
                                             this.getID()));
    return value;
  }
}
```

Figure 8: Java pseudocode for a generated cache class.

4. FURTHER ISSUES OF METHOD-CACHING AND THEIR SOLUTION

This section discusses a list of important challenges and problems that occur when implementing a method-based cache.

4.1 Mapping of Result Objects

Once a cache is working for a service interface class, the client using the cache should not get direct access to the corresponding application server objects but act on related cache proxy objects instead. Direct access of application server objects by the client might harm the cache's consistency and leads to bad system performance due to the cache being bypassed.

A problem in this context arises when service methods invoked by the client return (handles to) application server objects as results. In this case, the cache has to detect the returned value and map it to a proxy object, which is eventually returned to the client. However, this strategy fails if the returned application server objects are wrapped in runtime objects that should not be cached, e.g. arrays or lists. In this case, the cache needs to create copies of runtime objects in which all references to application server objects are replaced by references to proxy objects.

E.g. the method `Subscriber.getSubscriptionList()` from Section 3.3 returns a `Vector` of application server objects with type `Subscription`. The cache maps the method's result to a `Vector` of corresponding proxy objects and passes it on to the client code. It recognizes the required mapping strategy by means of the attributes `mapstrategy` and `containedclass` in the cache model's XML file.

4.2 Correctness of Cache Models

An important issue is to assert the correctness of a cache model and, with it, the cache's returned results.

We have implemented a special "correctness check" mode for the cache, in which it runs as usual, but, in case of a cache hit, it also delegates a corresponding method call to the application server. After that, the result coming from the cache's store and the result returned from the application server are tested for equality. If they differ, the cache model is incorrect and must be reworked. If, on the other hand, no inconsistencies can be detected for a set of typical test cases, there is strong evidence that the cache model is correct. Still, the approach does not guarantee the cache model's correctness.[6]

In order to perform equality tests for a correctness check, the application programmer might have to provide comparison strategies for complex method results. In our implementation, the strategies may be customized via the cache model's XML file. The default comparison strategy is based on Java's `java.lang.Object.equals()` method. However, the corresponding results might be misleading or inappropriate in respect to a correctness check. E.g., consider a service method that returns a list of unordered result values originating from an SQL database query. Due to the behaviour of the underlying database system the order of the list's entries might differ between two identical method calls, even if the actual entries remain the same. A suitable comparison strategy has to take this into account and must check the result list's values irrespective of the their particular order.

As part of our prototype, we have developed a tool that is used to observe a method cache's behaviour. It allows for connecting to a client's cache at runtime and presents the cache's profile and correctness data which is collected during cache operation. Further, profiling and correctness checks may be turned on and off on a per method basis via the tool's user interface.

The tool may also assist a developer in finding out whether the applied cache model is reasonably precise (according to Definition 8). If the cache model is too imprecise, the stored cache results might have to be invalidated too often which leads to poor cache hit rates. The related inefficiencies can be detected by observing cache hit rates and invalidation rates at cache runtime. Based on the corresponding results, the developer may adjust the cache model to improve the its precision.

[6]Another more theoretical option would be to prove cache model correctness by analyzing the related method implementations. However, this task cannot be (well) automated and is extremely difficult to handle for a typical application programmer.

4.3 Cache Bypassing

Cache bypassing is any event that changes the application server's state without notifying the cache. The proposed cache concept clearly faces the bypassing problem for reasons, which are discussed next.

As in classical client server scenarios, there might be more than one client invoking service methods at a given application server.

A solution to this situation is an extension of the cache component and the application server such that a client's cache must register at the application server when starting up. If the application server encounters a write method invocation from a certain client, it notifies all registered caches of the event, so that they can handle related invalidations. A drawback of this solution is that it is invasive since the application server system or at least some contained components must be adapted.

A second solution is to place the cache right in front of the application server. Then, the cost of potentially performing remote method calls by the client cannot be avoided. However for a cache hit, one still saves all computation costs that otherwise would arise inside the application server. If there is more than one application server, e.g. a cluster of servers, this approach requires communication between the server for exchanging invalidation methods.

We are currently working on a client side solution where the clients exchange invalidation messages by communicating with a central *invalidation server*. The invalidation server receives invalidation messages from any client and forwards them to all other clients that potentially need the invalidation messages. We assume that this method is efficient since the messages are short and the related communication protocol is simple.[7] Still, related experimental results are yet missing. The described mechanism is well applicable to Web application scenarios where there is a fixed but relatively low number of clients (represented by servlet-enabled Web servers).

Another problem occurs if a subsystem of the application server (e.g. an underlying database) potentially changes its state but the application server is not (immediately) notified about the change. In this case, the subsystem should be customized so that it notifies the application server which in turn may trigger notifications of client-side caches: e.g. if an application server accesses a relational database, some other process might perform updates on that database too. Database triggers may then be used to notify the application server of the changes. It is important to note that related invalidation message must refer to the invalidation model instead of the changed server state itself. (Thus it must have the form $(\textit{index number}, \textit{index element})$.)

[7]A related message is just a tuple of the form $(\textit{index number}, \textit{index element})$ (see Definition 4 in Section 6).

Obviously, a time-out-based approach can also be followed to solve this problem. Method results are then automatically invalidated after a certain time limit has been exceeded. However, the time-out strategy has the potential of delivering stale results whenever an application server state changes before a related cached method result times out.

Many other so called *weak replication consistency* approaches might be applicable in our context and have been considered in the literatur (see [6, 34] for a list of related references). Much like a time-out-based approach, they provide for weaker cache consistency and coherency strategies such that results from the cache are not always equal to results from direct server access. Furthermore, transactional cache consistency mechanisms come into play when transactional behaviour should be supported on the client side, e.g. when using the Java Transaction API (JTA) (see [12] for a discussion of transactional cache protocols). Including such aspects in our system is part of our future work.

4.4 Cache Size

Controlling cache size is a topic that has been extensively dealt with in the literature. For the sake of completeness, we would like to mention that standard cache replacement strategies such as LRU, LFU or LRD can also be used for the method cache.

For our current prototype, we have implemented replacement strategies based on LRU and LFU. Experimental results show that the performance loss of LRU replacement is minor when compared to an ideal "no replacement" strategy (see also Section 5).

5. PERFORMANCE EXPERIMENTS

This section discusses experimental results from the usage of a method cache in a realistic E-commerce Web site scenario.

5.1 Experimental Setup

In order to drive our experiments we relied on RUBiS v1.2 [29], a performance test suite for application server systems developed at Rice University.

RUBiS authentically models an auction Web site based on eBay.com [9]. The Web site is implemented several times with different implementation variants. Among others, RUBiS comprises three variants based on EJB and Java servlet technology. Persistent user data, which is typical for an auction Web site, is stored in a relational database whose size and structure is comparable to the one used at eBay.com.[8]

The test load on a running RUBiS Web site is generated through a set of HTTP client emulators, modelling typical page access patterns of Web site users. To accomplish this, the emulators run on separate machines and create HTTP requests for a certain number of virtual users, where each user's behaviour is emulated individually. Internally, the clients model user behaviour by means of a state engine. A transition from one user state to the next is chosen randomly based on a given probability table. The resulting URLs are potentially parametrized and cover the entire Web site's functionality such as browsing, logging in, bidding, adding items etc. The load on the auction Web site can then be varied by changing the number of virtual users on the client emulators.

[8]The research group behind RUBiS compares scalability and other criteria of different applications server technologies and implementations. For further information, please refer to their papers ([5, 4])

About 15% of the resulting requests produce write access on the server side (in other words, updates on the underlying database). According to [5] this is accurately reflects the read-write mix at eBay.com and is quite typical for user access patters of Web applications.

In order to get relevant experimental results we had to adjust the code of the RUBiS client emulators in the following way:

- When accessing a dynamic page a client emulator originally does not only download the respective HTML page but also scans the the page for image references and tries to download the images. We turned this feature off for the following reasons: In a case of a real Web application a Web browser typically downloads frequently accessed images only once and stores them in its local cache. Thus, downloading (the same) images over and over again at every page access does not accurately reflect of a comparative "real" Web site. Besides, we wanted to focus our experiments on the client side effects of the method cache on HTML page generation and transmission.

- We have noticed that the client emulators were implemented in way such they gave up on trying to download the same dynamic page after five unsuccessful requests. After that, the related client-side user session was terminated and the total number of active virtual users decreased. As an effect, the number of virtual users was continuously reduced at the client emulators until there existed just as many users as the web application could still well process. In other words, the number of virtual users slowly dropped to a level where no user sessions exceeded the limit of five unsuccessful requests. While this might reflect user behaviour in a realistic way, it does not fulfill the intended requirement of constantly stressing the Web application with a well defined number of virtual users. For this reason we turned this feature off and set no upper limit for the number of unsuccessful page requests.

For our experiments, we chose to use an EJB variant from RUBiS, which is entirely based on stateless session beans, as it is the best performing EJB variant according to [5]. We tested it both with and without a method cache, while keeping other hard- and software settings fixed.

In order to develop the cache model, 35 service methods had to be considered for model dependencies and thereof, 30 were considered as read methods. Ten abstract indexes were defined according to Section 3 and one auxiliary Enterprise Java Bean had to be written to improve the cache model's precision.[9]

[9]The additional enterprise bean is required for the definition

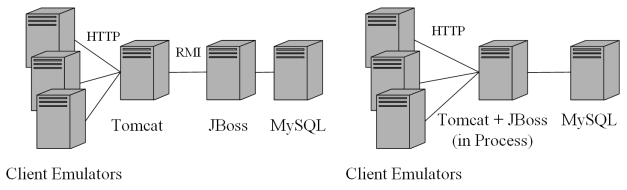

Figure 9: The two variants for the experimental setup: a) Web and application server on two machines (RMI variant) b) Both servers in-process and on the same machine (in-process variant)

Before starting any performance experiments, we checked the correctness of the cache model according to Section 4.2.

The experiments were performed in a closed network with up to 6 PCs running under MS Windows XP. For the first variant of the experimental setup one machine acted as the Web and application server hosting Tomcat v4.0.3 [36] and JBoss v2.4.6 [13]. In this configuration JBoss and Tomcat ran within the same Java virtual machine process. Further, invocation of bean methods happened via regular method calls and *not* over Java's Remote Method Invocation (RMI).[10] In the following, we are going to refer to this configuration as the "in-process variant".

For the second variant of the experimental setup one machine acted as a (separate) Web server and another machine hosted the JBoss application server. In this case, bean method invocations happened over RMI and over the network. We are going to refer to this variant as the "RMI variant". Figure 9 depicts the two setup variants.

The Java Developer's Kit (J2SDK) v1.4 from Sun was used as the related Java environment. The Java Virtual Machines (JVMs) for the respective server processes were started with an initial and maximum heap size of 256MB. Apart from this, the JVM's default parameters were used.

A separate machine acted as the database server, running the auction site's database under MySQL v3.23.38-max [24]. The database's file size is about 1 GB. It contains 1 million user entries, 33000 items for sale and around 10 auction bids per item. (For further details about the database see [5]). Moreover, three machines acted as client emulators.

of certain index functions. E.g., one cache model index is designed in a way such that its index elements are represented by user IDs of registered web site users. However, not all service methods whose cache model annotations refer to this index allow for direct access of a related user ID via their arguments or results. Instead, some methods only offer access to a user's nick name. As there is a one-to-one relationship between a user's IDs and its nick name, every nick name can be (uniquely) mapped to a user ID by querying the related user table of the underlying database. The auxiliary enterprise bean just preforms these kind of computations and so it was easy to implement. *In particular, it does not alter or interfere with any RUBiS application code.* The auxiliary bean's methods are included in the cache model and so the related method results are cached. This way, the evaluation of index functions referring to the bean remains efficient in case of a cache hit.

[10]This tight integration of Web and application server is a special feature of JBoss.

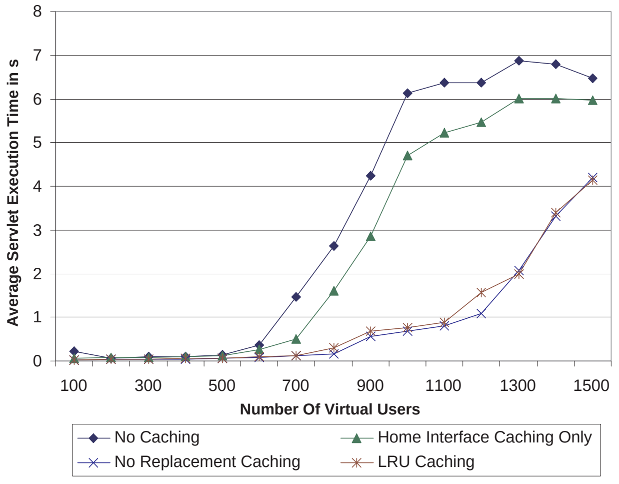

Figure 10: Average servlet execution time in seconds as a function of the number of virtual users for the RMI variant.

All machines had the same hardware configuration: a 1.18 GHz Pentium 4 Processor, 512 MB RAM, and a standard PC hard disk drive. By monitoring the related system resources, we ensured that neither network bandwidth nor process load on the client emulator machines represented a potential bottleneck for the experiments.

5.2 Results

Figure 10 and 11 show the average servlet execution times for the two setup variants as a function of the number virtual users that hit the site in parallel from the client emulators. The average values are based on all servlet executions that were performed during a run. Every data point represents a run of 5 minutes with a ramp up time of 1 minute. Servlet execution was timed by inserting code for measuring system time right before and right after the code for the respective servlet execution.

In addition to the ramp up time, we granted the Web site a warm-up-phase of 4 minutes when running it with the method cache. The warm-up-phase is required to fill the cache with an initial set of method results, so that the cache hit rate remains constant for a related run. When using the method cache we tried an LRU as well as a no replacement strategy. For LRU the maximum cache size was set to 2000 entries. This means that no more than 2000 method results were stored in the method cache at any point of time.

In respect to the RMI variant, we also distinguished between no caching at all and caching of EJB home interfaces *only*. In the latter case EJB home interfaces are cached after looking them up through the Java Naming and Directory Interface (JNDI). Caching home interfaces is a common EJB programming technique and so we wanted to compare its performance benefits with the additional benefits of the

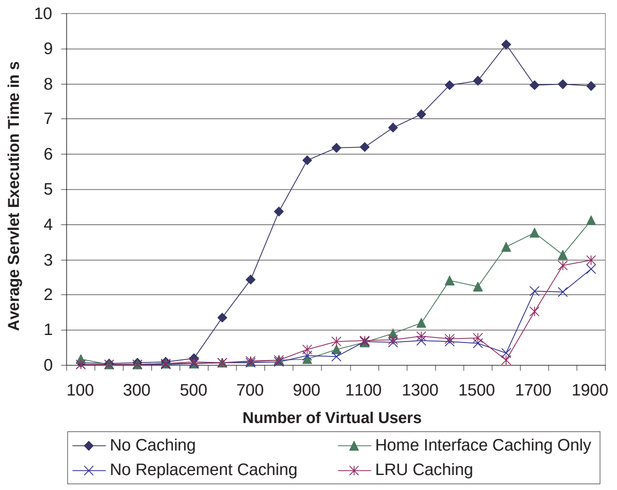

Figure 11: Average servlet execution time in seconds as a function of the number of virtual users for the in-process variant.

method cache. (Please note that home interface caching is also provided by the method cache.)

As one can see, the execution times increase with the load of virtual users. However, the average execution times when using the cache remain considerably lower than the corresponding times without the cache. E.g. at a load of 1100 virtual users, the average execution time with the cache is still under 1 second for either variant while it is over 6 seconds when not using the cache.

"Home interface only caching" has a positive effect on servlet execution times for both the RMI and the in-process variant. As discussed below, Figures 14, 17, 16 and 17 lead to similar results. Thus, from a performance perspective, caching home interfaces must be well recommended.

Figure 10 illustrates that for the RMI variant, "home interface only caching" is out-performed by the use of a method cache. (Figures 14 and 16 from below lead to the same conclusion.)

Surprisingly, "home interface only caching" has a tremendous effect on the in-process variant (Figure 11) although the corresponding JNDI look-ups are in-process and should be rather efficient. In this case, the use of the method cache still improves servlet execution times but obviously the major gain is provided by caching home interfaces.

Figure 12 and 13 illustrate the average method execution times of EJB method calls as a function of the number virtual users. While for the RMI variant the execution time gain is very significant, the respective gain for the in-process variant is considerable but lower. The reason for the difference are the savings related to time-consuming RMI calls in case of the RMI variant which do not exist for the in-process variant.

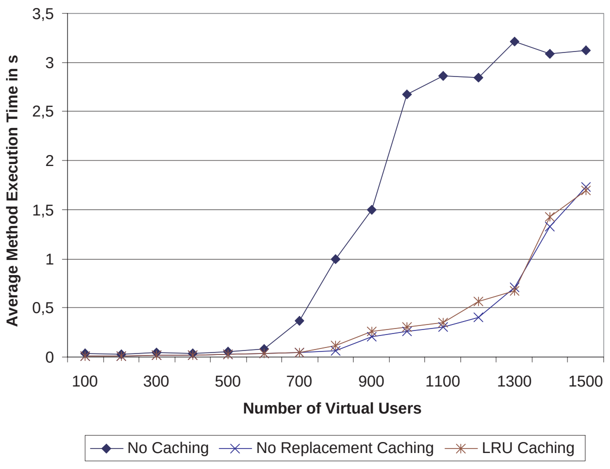

Figure 12: Average method execution time in seconds as a function of the number of virtual users for the RMI variant.

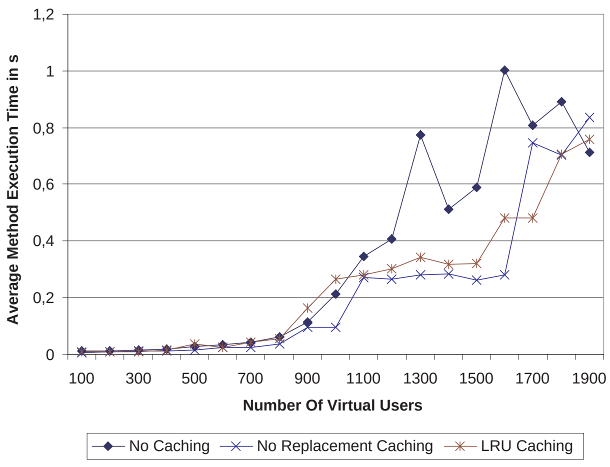

Figure 13: Average method execution time in seconds as a function of the number of virtual users for the in-process variant.

Figure 14 and 15 present the throughput in requests per second for the two variants on the side of the client emulators. The corresponding data was recorded during the kind of runs that have been described above. For both variants the throughput was considerably improved when using the method cache. E.g. for the RMI variant, the maximum throughput was raised from about 90 to over 150 requests per second. Much as for servlet execution times the impact of "home interface only caching" is relatively small for the RMI variant but noticeable for the in-process variant.

When comparing the two setup variants from Figure 14 and 15 one can see that in all cases, the RMI variant scales considerably better as it reaches and sustains higher levels of throughput. This result supports the common practice of

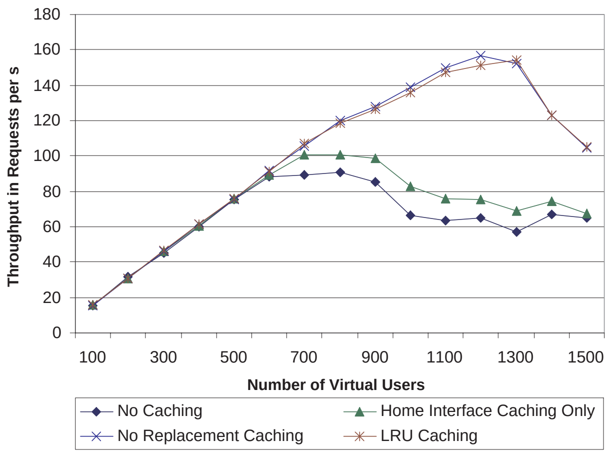

Figure 14: Throughput in requests per second on the client emulator side as a function of the number of virtual users for the RMI variant.

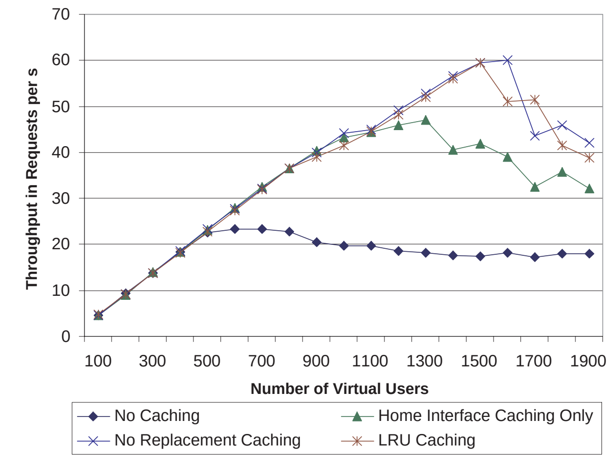

Figure 15: Throughput in requests per second on the client emulator side as a function of the number of virtual users for the in-process variant.

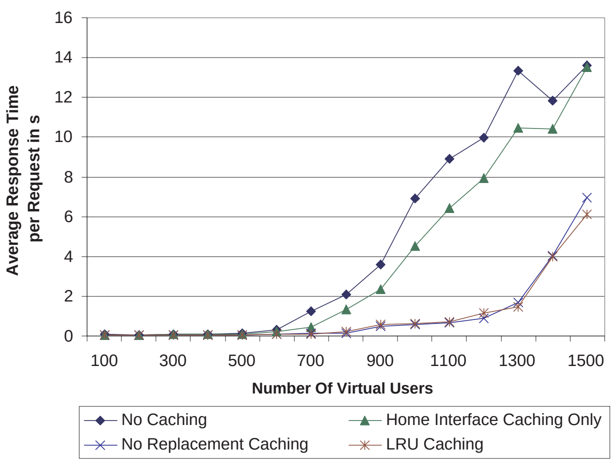

Figure 16: Average response time per request on the client emulator side as a function of the number of virtual users for the RMI variant.

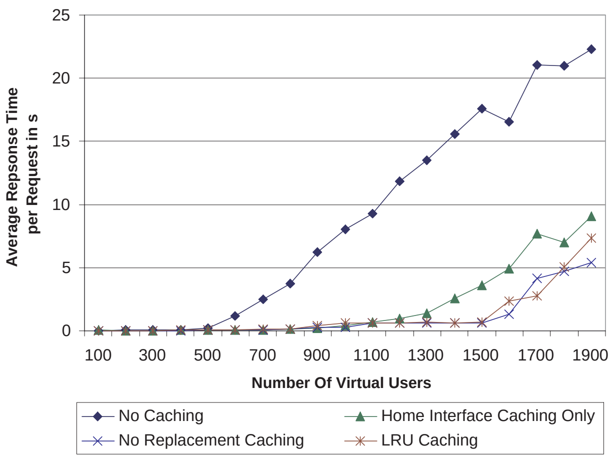

Figure 17: Average response time per request on the client emulator side as a function of the number of virtual users for the in-process variant.

running Web and application server on separate machines in order to reach better scalability.

Figures 16 and 17 show the average response times for HTTP requests on the client emulator side. Here, the performance improvements of using method caching (and home interface caching) are striking: E.g. at a load of 1100 clients, the RMI variant without caching requires about 9 seconds in order to serve a page while it needs under 1 second when using the method cache. Again, the benefit of "home interface only caching" is very distinctive for the in-process variant. Adding method caching for this case still leads to a further improvement, but it is rather low.

For all performance measures we have presented, the LRU replacement strategy performs about as good an "ideal" no replacement strategy.

Note that for very long lasting runs of the client emulators the performance of the no replacement strategy slowly degrades. The reason for this is that the JVM heap space is exhausted by storing more and more method results. As the JVM runs short on memory, a lot of CPU time is consumed by garbage collections and eventually the system runs out of memory. Still, the described effects did not affect the data points from above, since the related test runs only last for a few minutes.

Figure 18 depicts the average cache hit rate as a function of the number of method calls when using LRU caching. The

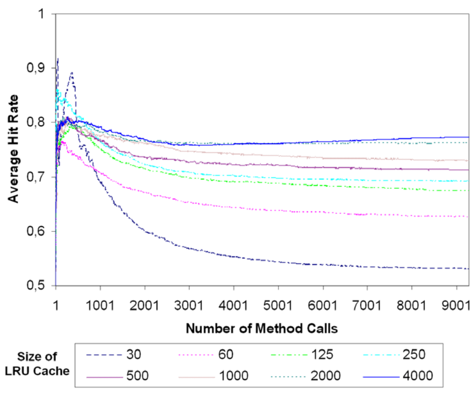

Figure 18: Average cache hit rate in respect to the number of EJB method calls produced by the emulator clients when using the LRU replacement strategy.

hit rate was observed for different LRU cache sizes ranging from a maximum of 30 to 4000 cachable method results. It is interesting that even small cache sizes result in long term hit rates of over 50%. By increasing the cache size the long term hit rate can be raised of up to almost 80%.[11]

6. OUR CONTRIBUTION IN RESPECT TO RELATED WORK

6.1 Web Application Caching

In the last two years, research as well as industry has made various efforts to improve the performance of Web applications by means of caching. It is beyond the scope of this paper to discuss all the related approaches in detail (please refer to [23] and [16]). Instead we will compare our approach against existing systems on a more general level and briefly discuss the advantages and disadvantages.

Figure 19 shows the tiers of a typical Web application architecture and highlights where caches potentially come into play:

- Application data caching happens somewhere in between the database and the application server tier. If it is done right in the front of the database ([8, 17, 35]), abstractions of database queries are associated with query results in the cache. In case of a cache hit, the query result is immediately returned by the cache as opposed to running the database query engine. At the application server side, application data is cached either programmatically through runtime objects whose structure has been designed by the application developer ([15, 19]) or it is controlled by an object-relational mapping framework ([11, 25, 28]).

[11]The initial peaks in the diagram relate to the fact that the first steps of the virtual users are often similar at the beginning of the experiment (e.g. entering the website via the home page) but diverge from each other later on. This way, especially good hit rates are reached during the first 500 method calls.

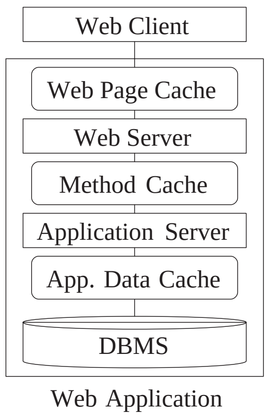

Figure 19: Common tiers of Web application architectures and related options for caching.

- Web page caching usually occurs in front of a servlet- or script-enabled Web server. Beyond the simple task of caching static pages, there are also many approaches for caching dynamically generated Web pages ([1, 27, 30, 38]).
- In contrast, the method cache is inserted at the "backend" of a servlet- or script-enabled Web server from where application server calls are initiated. To the best of our knowledge our approach is the first one enabling caching at this position.

One major question that all dynamic Web caching strategies must deal with is when and how to invalidate cache content. Related solutions are discussed below:

In [3, 16] and [18] URLs of dynamic pages on the Web server side are associated with dependent SQL queries on the database level. If a database change affects a corresponding query, the related pages in the cache are invalidated. In [3, 16] dependencies between queries and URLs are automatically detected through sniffing along the communication paths of a Web application's tiers.

A general flaw of this URL to SQL query mapping strategy is that it does not account for states from intermediate tiers such as an application server. Although such states might be relevant for dynamic page generation, they can only contribute to a related invalidation policy if they are reflected in the database state. A good example for this problem are stateful session beans from EJB. Clearly, our approach does not encounter this problem as it explicitly deals with application server states.

Other cache strategies for dynamic Web page caching require a developer to provide explicit dependencies between URLs of pages to be cached and URLs of other pages that invalidate the cached ones ([27]). Much as in our approach, the dependencies are declared as abstract named events that may be parametrized. An event parameter usually represents a request parameter of the cached page's URL.

Often, server-side page generation scripts or database sys-

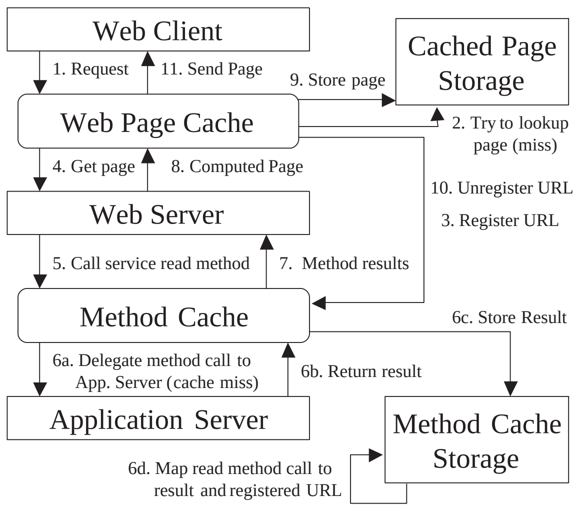

Figure 20: Execution steps of an integrated Web cache and method cache at a Web page miss.

tems may also invalidate a cached page by invoking invalidation functions of the Web cache's API ([1, 30, 38]).[12] Unfortunately these strategies are invasive which means that application code (e.g. page generation scripts) has to be changed. Finally, time out-based invalidation policies such as discussed in Section 4.3 are adopted by most dynamic Web caches.

An explicit fragmentation of dynamic Web pages via annotations in page generation scripts helps to separate static or less dynamic aspects of a page from parts that change more frequently ([7, 10]). Also, dependencies such as described in the previous paragraph can then be applied to page fragments instead of entire pages. In this respect, our approach enables an even more fined grained fragmentation as it treats dependencies on a level where page scripts invoke service methods from the application server. A great benefit, is that explicit page fragmentation annotations (such as supported by [10]) may then become obsolete.

However, when just using the method cache, the corresponding page generation scripts still have to be executed at every page access as only the results of service methods being invoked by the script code will be cached. In the next section, we discuss how page generation can be avoided by integrating dynamic Web page caches with our approach.

6.2 Integration of Web Page Caches

The presented method-based cache can very well improve URL based caching strategies for dynamic Web pages. The basic idea is that invalidation information which is provided by the method cache can not only be used to invalidate method results but also to invalidate cached Web pages. This way, no further invalidation policy is required for the related dynamic web page cache.

[12]Typical examples for server-side script languages are Active Server Pages (ASP, [20]) and Java Server Pages (JSP, [33]).

Figure 20 illustrates how a dynamic page access is processed in a related system architecture: first, a Web client's HTTP request with the URL of a queried page reaches the Web page cache (1). We assume that the corresponding page is not (yet) cached (2). In order to facilitate integration of the two caches, the Web page cache registers the requested URL at the method cache (3). Then the request is delegated to the Web server (4). A servlet or script runs to generate the requested page and, as an effect, service methods are called from inside the servlet or script code (5). The method cache receives the related method calls and, as usual, either returns cached method results or delegates the calls to the application server (6a). Moreover, it attaches every newly cached or looked up method result with the URL that has been registered in step (3). Hence, besides mapping read method calls to corresponding method results, the method cache now also maps method calls to URLs from dymamic pages whose generation caused those calls (6d). After the page generation has been completed (7, 8), the page is cached inside the Web cache (9). Finally the Web cache unregisters the URL from the method cache (10) and sends the generated page to the Web client (11).

Assuming that no state for generating Web pages is kept at the Web server, a cached page only needs to be invalidated if one of the service method results that were computed during page generation becomes invalid. As the URLs of pages depending on certain service method calls are stored by the method cache at step 5, consistent page invalidation is straight forward: if the method cache invalidates a read method result, it takes all URLs associated with it (at step (6d)) and triggers invalidation of the related pages at the Web cache.

If a page computation invokes a model write method (see Section 3), the corresponding page is not cachable, since it (potentially) has side effects at the application server: E.g., in an E-commerce scenarios, this typically happens when users add an article to a shopping cart or submit an order.

We would like to stress that this strategy is only consistent if all state for page generation (e.g. important user session information) is kept at the application server or in subordinate systems. Thus, it demands a natural separation of tasks where the Web server focuses on web page rendering and the application server is responsible for business logic.

6.3 Other Related Approaches

Our approach is heavily influenced by the concept of function materialization as first presented in [14]. In the following, we will briefly highlight the assumptions made in [14] and explain why the related technique cannot be applied in context of service interfaces.

In their paper, the authors discuss the *precomputation* of function results (or in our terms read method results) for a given set of objects which is stored in an object-oriented database (OODB).

They assume that a function's parameter list is relatively short and that the corresponding parameter types are restricted to persistent object types. Since the number of objects in the database is finite, the set of argument combinations for a function is also finite when considering a certain database state. Thus, precomputation of all potential function values may be possible. Further, a precomputed function exclusively operates on database objects, so that its result depends on database states *only*. This enables an automated extraction of potential data dependencies between functions by analyzing on the functions' implementations code.

Unfortunately, none of the related assumptions apply to the case of application server scenarios. The dependencies between method implementations may be arbitrarily complex and in general they cannot be automatically analyzed. Also, the set of objects involved in method calls is usually not known until method execution time.

A rather simple approach to method result caching could be found as part of the Torque framework in the Apache DB Project ([2]). In order to cache data of business objects, Torque allows for caching attributes as well a method results.

In the Torque framework, the support for method result caching is quite basic as only manager classes for storing and retrieving method call results are offered. The actual code for accessing the cache and transparently delegating method calls must be written by hand. Also, there is no concept of automatically generating a transparent layer of cache classes that implement a set service interfaces such as in our approach. Furthermore, an invalidation strategy must be manually implemented for every cachable service method by following an event notification design pattern. Hence, in contrast to our approach, there is no generic invalidation model which allows for expressing invalidation dependencies on a descriptive level and in a central place. Besides, a built-in support for checking the correctness of invalidation strategies is missing.

7. CONCLUSION AND FUTURE WORK

This paper has presented the concept of a method cache — an approach for caching results of method-based service interfaces on the client side of an application server system.

A typical use case is a Web application whose performance needs to be improved. As our experimental results show, the overall throughput of a realistic test system can be considerably increased and its response time extremely reduced when using a method cache. Furthermore, the presented approach is applicable to real world programming languages and development standards such as Java and EJB-based application servers. Still, method cache access is almost seamless to the client code as well as to the server because the cache implements the application server's service interface. This allows for integrating a method cache even in late cycles of project development. The fact that we tested the method cache on the basis of an existing Web application is good demonstration of this feature.

The presented technology helps to overcome performance and scalability issues for existing application server standards such as EJB. Future experiments will have to reveal if any performance oriented design patterns for EJB such as "Session Façade" or "Value Objects" (see [19]) may even become obsolete this way. In this context we are currently working on a cache extension that provides a prefetching mechanism for results of read method calls.

It should be stressed that the concept of a method cache is not restricted to application server systems: even the implemented prototype can be applied to any software component that is abstracted by a set of method-based Java interfaces. Obviously, it can be realized for middleware technologies other than EJB such as CORBA ([26]), DCOM ([21]), .NET ([22]) or SOAP ([37]).

A cache model, which must be developed by an application programmer, ensures that cached method results are consistent with the state of an underlying software system. Based on our experience, we believe that developing such a cache model is a manageable task for a reasonably qualified developer and that it can be done at affordable cost.

We have presented a run time test for checking the correctness of a cache model (Section 4.2). It is up to the application developer to choose a proper set of tests and to run them for assuring cache model correctness.

The suggested cache model semantics are well suitable for the service interfaces we have dealt with so far. However, future experience might show that the model semantics must be refined in order to accomplish special or more complex dependencies between service interface methods.

In the context of Web application systems, we have suggested a way to integrate a typical front tier Web cache with a method cache: basically, the dependencies for invalidating method results on the method cache level are reused for triggering invalidation of cached Web pages. In order to enable this integration, only two features are required on the Web cache-side: a mechanism for programmatically invalidating cached pages and allowance for observing URLs which are accessed via the Web cache (see Section 6.2).

Due to the integration, conventional invalidation techniques for dynamic Web cache systems become obsolete. The Web application as a whole is then based on a two-level cache strategy, where level one caches entire Web pages and level two caches results of service method calls. The caching feature is still transparent to the application code and from a programming point of view the invalidation policy for the caches is located in a single document (the cache model XML file from Section 3.3).

Examining the performance impact of the integration of a Web page cache and the method cache will be part of our future work. Furthermore, we are in the process of implementing and studying consistency protocols for multiple client-side method caches (see Section 4.3).

In the terms of aspect oriented programming (AOP), the presented caching feature may be considered as a separate concern. In essence, the cache model can be regarded as an aspect language to define the cross cutting points of the caching aspect with the rest of the application. The cache class generator then represents an aspect weaver.

8. REFERENCES

[1] J. Anton, L. Jacobs, Y. Liu, J. Parker, Z. Zeng, and T. Zhong. Web caching for database applications with oracle Web cache. In *Proceedings of the ACM SIGMOD Conference*, Madison, Wisconsin, USA, June 2002. ACM Press.

[2] Apache Group. The Torque framework of the Apache DB project. `http://db.apache.org/torque`.

[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 *Proceedings of the ACM SIGMOD Conference*, Santa Barbara, California, USA, May 2001. ACM Press.

[4] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. A comparison of software architectur for E-business applications. Technical Report TR02-389, Rice University, 2001.

[5] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the OOPSLA Conference*, Seattle, Washington, USA, November 2002. ACM Press.

[6] G. V. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a caching service for distributed CORBA objects. In *In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, Hudson River Valley, New York, USA, April 2000.

[7] A. Datta, K. Dutta, H. Thomas, and D. VanderMeer. A comparative study of alternative middle tier caching solutions to support dynamic Web content acceleration. In *Proceedings of the 27th VLDB Conference*, Rome, Italy, August 2001. Morgan Kaufmann.

[8] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A middleware system which intelligently caches query results. In *Middleware*, pages 24–44, 2000.

[9] Ebay, 2002. `http://www.ebay.com`.

[10] ESI – edge side includes, 2002. `http://www.esi.org`.

[11] Excelon. Javlin – the EJB data cache manager, 2002. `http://www.exln.com/products/javlin`.

[12] M. J. Franklin, M. J. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Transactions on Database Systems*, 22(3):315–363, 1997.

[13] Jboss, 2002. `http://www.jboss.org`.

[14] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proceedings of the ACM SIGMOD Conference*, Denver, Colorado, USA, May 1991. ACM Press.

[15] S. Kounev and A. Buchmann. Improving data access of J2EE applications by exploiting asynchronous messaging and caching services. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, August 2002. Morgan Kaufmann.

[16] W.-S. Li, W.-P. Hsiung, D. V. Kalshnikov, R. Sion, O. Po, D. Agrawal, and K. S. Candan. Issues and evaluations of caching solutions for web application acceleration. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, August 2002. Morgan Kaufmann.

[17] 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 *Proceedings of the ACM SIGMOD Conference*, Madison, Wisconsin, USA, June 2002. ACM Press.

[18] Q. Luo and J. F. Naughton. Form based proxy caching for database-backed Web sites. In *Proceedings of the 27th VLDB Conference*, Rome, Italy, August 2001. Morgan Kaufmann.

[19] F. Marinescu. *EJB Design Patterns*. Wiley, USA, 2002.

[20] Microsoft. Active Server Pages (ASP). `http://www.asp.net`.

[21] Microsoft. Microsoft Distributed Common Object Model (DCOM). `http://www.microsoft.com/com/tech/dcom.asp`.

[22] Microsoft. Microsoft .NET framework (.NET). `http://msdn.microsoft.com/netframework`.

[23] C. Mohan. Chaching technologies for Web applications, 2001. Tutorial at VLDB Conference 2001, Rome, Italy, `http://www.almaden.ibm.com/u/mohan/Caching_VLDB2001.pdf`.

[24] MySQL. The MySQL relational database system, 2002. `http://www.mysql.com`.

[25] EJB performance analysis, 2000. `http://objectbridge.sourceforge.net/performance/ejb-performance-analysis.html`.

[26] OMG. Common Object Request Broker Architecture (CORBA). `http://www.corba.org`.

[27] Persistence Software. Dynamai – a technical white paper, 2001. `http://www.persistence.com/products/white_papers_register.php`.

[28] Persistence Software. Persistence benchmark – extreme EJB performance with PowerTier, 2001. `http://www.persistence.com/products/white_papers_register.php`.

[29] The RUBiS project, 2002. `http://www.cs.rice.edu/CS/Systems/DynaServer/RUBiSWWW`.

[30] Spider Software. Accelarting content delivery: The challenges of dynamic content, white paper, 2001. `http://www.spidercache.com`.

[31] Sun. Enterprise Java Beans (EJB). `http://java.sun.com/products/ejb`.

[32] Sun. Java 2 Enterprise Edition (J2EE). `http://java.sun.com/j2ee`.

[33] Sun. Java Server Pages (JSP). `http://java.sun.com/products/jsp`.

[34] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994.

[35] Times Ten Team. Mid-tier caching: The TimesTen approach. In *Proceedings of the ACM SIGMOD Conference*, Madison, Wisconsin, USA, June 2002. ACM Press.

[36] Jakarta Tomcat, 2002. `http://jakarta.apache.org/tomcat`.

[37] W3C. Simple Object Access Protocol (SOAP). `http://www.w3.org/TR/SOAP`.

[38] XCache Technologies. XCache – a dynamic content Web cache. `http://www.xcache.com`.