Improving Performance with Application-Level Caching

Louis Degenaro, Arun Iyengar, and Isabelle Rouvellou
IBM Research
T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{degenaro,aruni,isabelle}@watson.ibm.com

Abstract— **Caching can be employed at many levels within a computer system such as at the processor level, within databases, Web browsers, servers, and proxies. For many software applications, system support for caching is insufficient for improving performance; application-level caching is needed as well. This paper provides an overview of techniques for building application-level caches. We also describe how we deployed application-level caching for improving performance in real settings. One of the novel features of our caching system is its method for automatically recomputing cached entities comprised of sets of objects based on object lifetimes. We also provide support for multiple contexts, transactions, locking, and application-specific cache API functions.**

I. Introduction

Caching is crucial for improving performance in computer systems. It can take place at several points within a system. A processor cache might have one or more caches for reducing the latency for accessing data. A database might also have a cache for storing frequently requested data. Caching within Web systems can occur at several different points such as within browsers, servers, and proxies. Browser and proxy caching reduce the latency for fetching remote documents and network utilization; server CPU cycles may also be reduced. Server caching can reduce processing cycles consumed by the server, particularly if the server generates considerable dynamic content which can be cached [7], [3].

For many applications, existing caches are insufficient for improving performance. In such systems, it may be possible to improve performance by employing caches explicitly managed by the application. We refer to this type of caching as *application-level caching.* This paper describes how application-level caching can be effectively designed. We then describe our experiences with deploying application-level caching in real settings.

II. Designing an Application-Level Cache

Often a cache, general purpose or custom built, is a critical part of the infrastructure needed for the successful deployment of an application. Performance benefits are achieved because once a result has been calculated, it can be stored in the cache and subsequently reused. Cached objects are typically stored in a hash table or balanced tree and indexed by a string, byte stream, or numerical value.

An application-level cache would typically be designed with an API which allows an application writer to explicitly manage the cache contents. The API allows an application to explicitly add, delete, or modify cached data. The API may also provide additional functionality such as enhanced features for achieving cache consistency. The major advantage of such an application-level cache is that it enables application writers to take advantage of their knowledge of the application specific structure and data access patterns when caching data.

The following sections describe features we incorporated in our design of a generic application-level cache.

A. Cache Architectures

One method for deploying a cache would be as a library which is linked in with the application (Figure 1). The cache then runs as part of the application's process. If the cache does not need to be shared by multiple processes, this approach is preferable to having the cache run as a separate process because it avoids the overhead and complexity of interprocess communication. We refer to this approach as the unshared architecture.

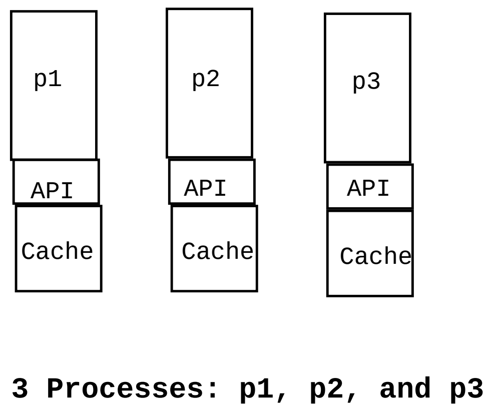

Fig. 1. The unshared caching architecture. A cache is associated with each process.

If an application is distributed across multiple processes which need to access a cache, the unshared architecture may be unsatisfactory because it requires each process to manage a different cache. In order for a cached object o to be accessible to n processes for $n > 1$, o must be stored in n caches. This can have a number of drawbacks. Extra space is required for storing all n copies.

Another problem is maintaining consistency. A cache infrastructure is required to coordinate operations in order to maintain consistency in a multiple application and distributed object environment. Several applications (clones or altogether different applications) located on the same or physically different processors may have a need for fast access to a common set of objects from cache. The cache infrastructure must efficiently coordinate cache operations to assure a consistent view of shared cached entities under multiple application access circumstances. If o is updated, inconsistencies can arise if all n cached copies are not updated atomically. The overhead and complexity for achieving atomic updates may be significant. Each of the n processes would have to independently materialize o. This results in significantly more overhead than just materializing o once.

Another approach is to have a cache operate as a long-running process which communicates with multiple processes comprising the application (Figure 2). This approach is known as the *shared architecture*. Using the shared architecture, it is only necessary to cache one copy of an object. This avoids the problems just mentioned of caching multiple copies of objects in different caches. There are two drawbacks to the shared architecture, however. One is that the latency for accessing a cached object may be higher because interprocess communication is required. A cached object might have to be fetched from a remote processor. A second drawback is that if the request rate to a shared cache is high, the cache can become a bottleneck.

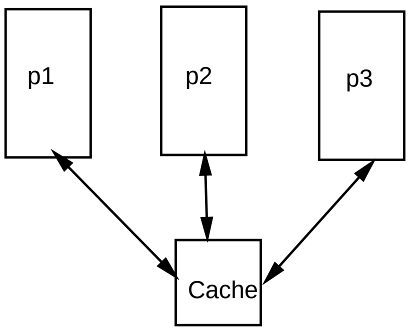

Fig. 2. The shared caching architecture. A single cache is shared by multiple processors.

The shared and unshared approaches can be combined into a *hybrid architecture* in which the system contains a single shared cache and processors (or processes) have unshared caches as well (Figure 3). Unshared caches can store read-only data. Unshared caches can also be used to store mutable data frequently requested by a particular processor; when this type of caching takes place, a method is needed for maintaining coherent caches after updates.

Another approach is to use a *distributed shared cache* in which a single logical cache is partitioned across several processors (Figure 4). Data could be partitioned among

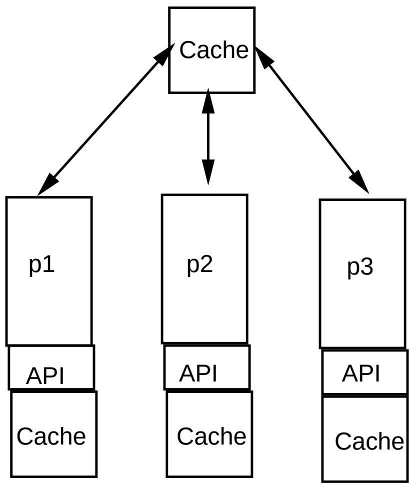

Fig. 3. A hybrid architecture comprised of multiple unshared caches and a single shared cache.

the caches by applying a hash function to the keys used to reference cached data. Alternatively, data could be partitioned in a more sophisticated manner in order to place cached copies of objects in proximity to processors which access the objects frequently. A distributed shared cache can achieve much higher throughputs than a shared cache running on a single processor. The processors on which a distributed shared cache runs might be the same processors executing other parts of the application; alternatively, a distributed shared cache could run on processors dedicated to caching.

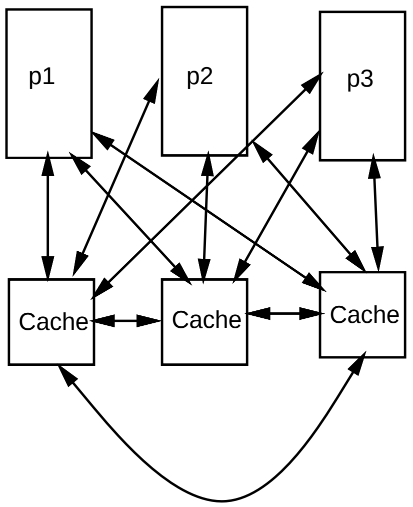

Fig. 4. A distributed shared cache partitioned across three processors.

A distributed shared cache could replicate frequently accessed objects on multiple processors. If replicated objects are updated, a cache consistency mechanism is needed to ensure that all copies are consistent. This can add considerable complexity to the cache design.

B. Cache Replacement

A key problem in cache design is deciding which object or objects to replace when the cache overflows. A commonly used replacement strategy is to replace the least recently used object (LRU). This approach often works well and is easy to implement. The LRU replacement algorithm can be implemented by maintaining a list of cached objects. Whenever an object is accessed, it is moved to the front of the list. The object at the end of the list is the least recently used one and the one to replace.

For Web caching, the GreedyDual-Size algorithm often outperforms LRU [2]. The GreedyDual-Size algorithm associates a value, V, with each cached object, p. When an object is first cached, V is set to the cost c of bringing the object into the cache divided by the size of the object. When a replacement needs to be made, the object with the smallest value of V, V_{min}, is removed from cache, and all cached objects reduce their values by V_{min}. When an object is accessed, its value is restored to the cost of bringing it into cache.

The definition of the cost, c, for bringing an object into cache depends on the critical resources the replacement algorithm is designed to conserve and the goal of the replacement algorithm. If the goal is to maximize the hit ratio, then c is set to the size of the object and V is thus 1. If the goal is to maximize the byte hit ratio, then c is set to 1. If the goal is to minimize latencies for fetching objects remotely, then c is set to the estimated cost for fetching an object remotely. If the goal is to minimize CPU cycles required to materialize an object, then c is set to the estimated number of CPU cycles for materializing the object.

A naive implementation of GreedyDual-Size would require updating V values for all cached objects whenever an object is replaced in the cache. However, a different method of computing V eliminates the need for updating values for objects which are already cached. Instead of decrementing V values for all previously cached objects, the value for the new object is inflated by V_{min}.

Ideally, a cache should have the ability to store objects in main memory as well as on disk. Main memory is needed for optimal performance. Caching on disk can be used when main memory overflows. Caching on disk is also essential for data which cannot be lost in the event of a system crash. Multithreading can be used to avoid wasting CPU cycles when a cache transaction becomes blocked due to a disk access.

A cache which can reach a warm state quickly after startup is highly desirable. This can be achieved by maintaining copies of frequently accessed cached data on disk. When the cache is restarted, copies on disk are read into main memory.

When a cache is storing an object from a remote server, it is essential for the cache to know when the object has become obsolete. Staleness tolerance of the data retrieved from a cache varies from application to application, some having zero tolerance while others are less strict. In the zero tolerance case, if at the same instant the same data were requested from both the original source and the cache, and if the results returned by each were not identical, then the cached data is stale.

One method for maintaining updated caches is for the server to notify the cache when an object changes. This approach results in strong consistency but can incur overhead at the server if the number of objects and caches is large. Techniques for achieving efficient server-driven consistency in Web caching are discussed in [10]. One problem with achieving server-driven consistency on the Web is the lack of a standard protocol for servers to send out invalidation or update messages to caches. In application-level caching where the application defines the protocol for communicating between remote servers and caches, this would not be an issue.

Another approach for achieving cache consistency is for the cache to poll the server for changes. In order to reduce the polling frequency, objects may have expiration times associated with them as is common on the Web. A cache only polls the server if it receives a request for an object which has expired. At this point, the cache sends a message to the server. If the object has changed, the new object is sent along with an expiration time. If the object has not changed, the server indicates this and sends an updated expiration time; the server does not need to resend the object since this would consume unnecessary network bandwidth.

C. Optimizing Performance with Multiple Caches

Some objects may be required by several applications. The several applications may be identical copies of each other (clones) or different applications altogether. The several applications may be co-located on the same processor or located on physically different processors.

Applications would usually like to have fast and consistent access to the objects they use. In general, the closer an application is to the objects it references, the faster the accessibility to them. If an application can access the objects it needs from a local cache, the access time will generally be quicker than accessing the objects it needs from a remote cache or from the original data sources.

If each application kept is own local cache of objects, and no object appeared in more than one cache, then there would be no duplicate cached object consistency issues. But problems arise when objects appear in more then one cache simultaneously. Absent any coordination infrastructure, each application is free to make changes to any objects contained in its local cache; but these changes would not be seen by other applications also containing those same objects in different local caches. So the choice seems to be fast access to objects but with poor consistency using uncoordinated caches, or slow access to objects but with good consistency using no caches. For example, a cache local to application A may contain various fragments used to compose a web page. Another cache local to application B, which is located on a physically different processor, contains the same fragments. If application A changes one or more of the common fragments, application B may not be aware of the changes made by application A, since only the cache local to application A reflects the changes made.

Thus, when application B obtains the common fragments to build web pages, it will receive stale data from its local cache.

We have designed a coordinated cache infrastructure to enable caching of objects shared by multiple caches used by applications. An important part of our design incorporates the use and maintenance of relevant statistics in order to determine measures of goodness derived by caching each individual object. These goodness measures are used to select objects which are the best candidates for each cache.

Figure 5 is a diagram depicting components which work together cooperatively to provide fast and consistent access to cached objects by applications. The network provides the communications medium through which the components communicate with each other. Applications access objects through a local cache. A shared cache provides access to objects faster than the original data source but not quite as fast as local caches. Objects not in cache are fetched from original data sources.

Each cache maintains statistics regarding object access frequency $A(o)$, cost to fetch $F(o)$, update frequency $U(o)$, and cost to update the cached copy $C(o)$. The desirability of caching any particular object $d1(o)$ in a local or shared cache can then be calculated:

$$d1(o) = (A(o) * F(o)) - (U(o) * C(o))$$

Objects with high desirabilities are given preference for keeping in the cache, and objects with low desirabilities are usually not kept in cache. The size of objects can also be used to influence the desirability $d2(o)$ of caching them:

$$d2(o) = d1(o)/size(o)$$

Again, the higher the calculated desirability for keeping a particular object in a particular cache, the greater the benefit derived to applications wishing to utilize them.

For example, a cache may contain various fragments used to compose a web page. Let's presume an application is utilizing fragments a1, a2, and a3. For the application's locally accessible cache, let's presume that the access frequencies, $A(o)$, for each of these fragments are 10, 100, and 1000 per unit time, respectively; that the costs to fetch the fragments, $F(o)$, are 50, 100, and 13 respectively; that the update frequencies, $U(o)$, for each of these fragments are 1, 2, and 3 respectively; and that the costs, $C(o)$, to update the cached copy for each fragment are 1, 2 and 3 respectively. Then the caching desirabilities for the local cache employed by the application are calculated as:

$$d1(a1) = (10 * 50) - (1 * 1) = 499$$

$$d1(a2) = (100 * 100) - (2 * 2) = 9996$$

$$d1(a3) = (1000 * 13) - (3 * 3) = 12991$$

With the given presumptions, the results show that the most desirable object to cache is fragment a3 with a value of 12991, followed by fragment a2 with a value of 9996, followed by a1 with a value of 499.

D. Automatically Computing Sets of Objects Based on Object Lifetimes

Some of the applications we have dealt with require caching sets of objects. Each object has not only an expiration time but also a start time when it first becomes valid. If the start and end times for objects are known in advance, it is possible for the cache to automatically recompute a cached set after it changes. This section describes an algorithm we have developed for accomplishing this.

Some objects always exist: they have no start and end points. Other objects have lifetimes: they are born, exist for some period of time, then expire. Caches need to effectively handle objects having limited lifetimes, bounded by a birth time and/or an expiration time. A time-sensitive cache is best able to quickly provide reusable time sensitive data.

When some staleness is tolerable, one method employed to address the question of "How (potentially) stale is stale?" is the assignment of expiration times. Objects are placed in the cache, each with an associated expiration (or end) time. If an object is requested from the cache prior to its associated expiration time, then it is made available (cache hit). But once the expiration time arrives, any request from the cache for the object fails (cache miss) until the object is replaced with a new version having an associated expiration time in the future.

Continuing with the fragment composed web page theme, let's assume that a web page under construction by our application is composed of three cached fragments: top, middle, and bottom. Further, let's assume that the cache contains these fragments, having expiration times of 09:00, 10:00, and 11:00, respectively. For purposes of demonstration, let's also assume that the cache does not get updated with new versions of these fragments.

If our application attempts to construct a web page at 08:30 using the top, middle, and bottom fragments, all will be retrievable from the cache. But if web page construction occurs at 09:30, our application will be able to get only the middle and bottom fragments from the cache, since the top fragment will have expired. If web page construction occurs at 10:15, only the bottom fragment will be available from the cache, and at 11:01 none of the fragments will be obtainable from the cache due to object expiration times having been exceeded.

Cache staleness occurs not only when an object expires, but also when a previously nonexistent object materializes. Converse to expiration times which dictate when a cached entity is no longer valid, start times predict when a cached entity becomes valid. An object can be placed into the cache with a start time greater than the current time. If a request for that object arrives prior to its associated start time, the request fails (cache miss). Once time has progressed to arrive at or go beyond the object's start time, a request for the object from the cache succeeds (cache hit).

Continuing the above example, let's now suppose that the three fragments had, in addition to their expiration times, start times of 02:00, 03:00, and 02:00, respectively. If our application attempts to construct a web page at 01:45,

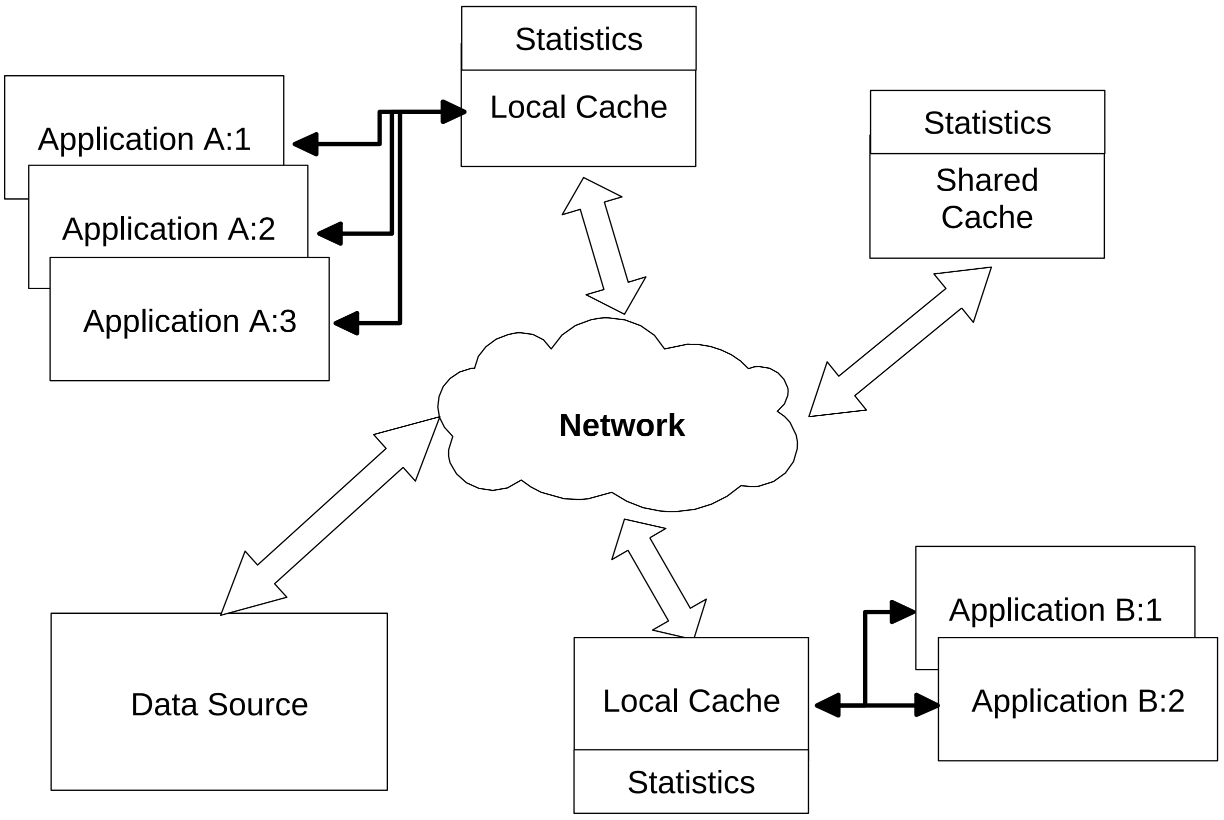

Fig. 5. Use of statistics by multiple caches.

none of the fragments would be retrievable from the cache since the current time does not exceed the sundry start times of the objects in cache. If web page construction occurs at 2:15, the top and bottom fragments would be retrievable from the cache. If web page construction occurs at 3:45 the middle fragment would, in addition to the top and bottom fragments, be retrievable from the cache.

We have designed a time-aware cache, where an object o is deemed valid in the cache if:

rule 0: startTime(o)$\leq$ currentTime and endTime(o) $\geq$ currentTime.

Sometimes it is desirable to cache an object that is actually a collection of objects. Each of the objects comprising the collection may have its own individual start and end time. Consider a collection that is a query result composed of several objects. Each of the several objects in the query result meets the query criteria, each object has its own individual lifetime, and all of the several objects together form the query result. Similarly, consider a web page that consists of several fragments. Each of the several fragments are used to create the web page, each fragment has its own individual lifetime, and all of the several fragments together form the web page.

Figure 6 is a diagram that shows the relationships between objects and collections with respect to time. Each object, e.g. o1-o9, has its own start and end time. Object o1 has a start time of t0 and an end time of t2, while object o2 has a start time of t2 and an end time of t4. Object o8 has a start time of t9 but its end time is after t9. Object o9 has an end time of t0 but its start time is before t0.

Each object in the cache is visible to users during its lifetime only. For example, object o4 is only visible during the time period (t3, t5); object o5 is visible during the interval (t3, t6); object o8 is visible during the time period after t9, and object o9 is visible during the interval before t0. A request for object o7 at times t0, t1, t2, t3, t8, and t9 would result in a cache miss, but a request for the same object at times t4, t5, t6, and t7 would result in a cache hit.

Each collection, e.g. c1-c2, has its own first time and last time which determine its lifetime boundaries, and each contains zero or more objects. Each collection is visible to users during its lifetime only. For example, collection c1 is visible during the interval (t1, t8) only; collection c2 is visible during the time period (t3, t9) only.

As noted above, each individual object has its own start time and end time. Each object (o1-o9) either participates in a collection or not based on both time (start and end times) and non-time criteria. With respect to time, an object is included in a collection if:

- rule 1: collection start time $\leq$ object start time $\leq$ collection last time, or
- rule 2: collection start time $\leq$ object end time $\leq$ collection last time

Let's presume that collection c1 contains odd and even numbered objects, and that collection c2 contains only even numbered objects.

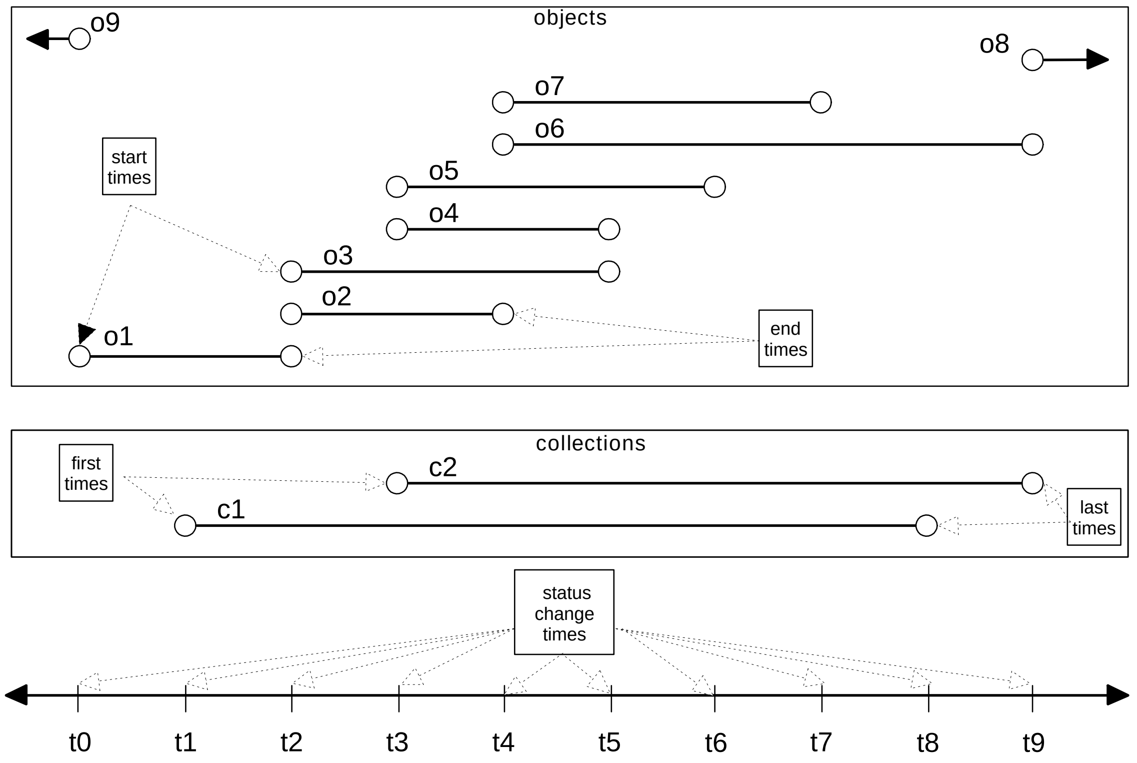

Fig. 6. Caching collections of objects with start and end times.

Collection c1 contains objects o1-o7. Object o1 is included in collection c1 due to rule 2. Objects o2-o5 and o7 are included in collection c1 due to both rule 1 and rule 2. Object o6 is included in collection c1 due to rule 1.

Collection c2 contains objects o2, o4, o6, and o8. Object o2 is included in collection c2 due to rule 2. Objects o4 and o6 are included in collection c2 due to both rule 1 and rule 2. Object o8 is included in collection c2 due to rule 1. Objects o1, o3, o5, and o7 are not considered since they are not even.

Whenever a request for a collection is presented to the cache, the result provided contains only those objects valid at the current time. For example, at time t6 a request for collection c1 would contain objects o5-o7; at time t8 a request for collection c1 would contain object o6 only; at time t2 a request for collection c1 would contain objects o1-o3.

Continuing our fragment example, one could imagine that collection c2 is actually our company's web home page constructed from various fragments; and that objects o2 and o6 are both middle fragments, perhaps a featured promotional item; and that times t3, t4, and t9 are 03:00, 06:00, and 10:00 respectively. Then if an application requests our web home page (c2) at 04:30 (between t3 and t4) the cache will provide one containing a book sale promotion (o2 as the middle fragment); if requested at 07:45 (between t4 and t9), the cache will provide one containing a record sale promotion (o6 as the middle fragment).

In Figure 6, each of the times t0-t9 are status change times. A status change time occurs whenever there is a start time or an end time for an object or a first time or a last time for a collection. The cache uses these status change times to determine an update time for each collection relative to the current time. The update time is the next closest status change time beyond the current time. The cache uses the update time in conjunction with the current time to determine when the visible part of the collection needs to be reconciled. Whenever a cache request arrives and the update time exceeds the current time, the cache simply provides the requested object (presuming it exists in the cache). However, if a cache request arrives and the current time exceeds the update time for the requested object, the cache must reassess the visibility of the subject object and those it contains.

With respect to collection c1, at time t6 the update time would be t7 because of object o7's end time; at time t7 the update time would be t9 because of object o8's start time, object o6's end time, and the collection's last time.

Again continuing our fragment example from above, recall our company's web home page, collection c2, contains a promotional item as the middle fragment, object o2 or o6 depending upon the time. Let's say that our application requests our company's web home page (collection c2) at 03:30 (time t3.5, somewhere between t3 and t4). This would result in a cache hit due to rule 0 (because the collection is an object). During the course of events resulting in the cache hit for our application, the cache determines whether or not a new update time needs to be calculated.

Let's presume that the previous calculated update time was 06:00 (t4). Then a new update would not need to be recalculated during this retrieval.

Sometime later, let's say at 07:45 (t5) our application again requests our company's web page (collection c2). The cache determines that the current time 07:45 (t5) exceeds the previously calculated update time 06:00 (t4) for the collection retrieved from the cache. In addition to other adjustments, the cache removes the old middle fragment (object o2) promoting a book sale from the web page, adds the new middle fragment (object o6) promoting a record sale to the web page, and calculates a new update time of 10:00 (t9) for the web page. Recall that collection c2 consists of even objects only. The new update time of 10:00 (t9) is due to both the expiration time of the collection itself and the start time of another object (o8).

E. Dealing with Context-dependent Data

Many applications define and operate under different *contexts*. A context is a scope in which variables and/or keys are resolved to specific values. One particularly relevant type of context is a transaction context. A transaction is a series of operations applied to a set of objects in a manner that transforms the system from one (application-defined) consistent state to another (application-defined) consistent state. Transactional contexts isolate the effects of transactions such that concurrently executing transactions do not interfere with each other; such interference, in general, leads to data inconsistencies. Adding support for context in the cache enables the implementation of basic concepts such as committed and uncommitted data. In itself, contexts do not provide full support for all forms of application transactional semantics, but they can help the application writer to enforce them.

The context associated with the data needs to become a first class entity, so that an object can be stored in a manner which takes both the key identifying the object and the context into consideration. The cache also maintains the set of keys associated with specific contexts. We introduce a composite index that combines the key and context into a single entity. Lookup is performed using the composite index. When a client is accessing the cache to look up an object, both the key and the context are used to retrieve the object. When an object needs to be deleted from the cache (e.g., this may occur when a cache overflows), the object to be deleted is identified based on both its key and context. Since the cache is maintaining lists of keys associated with specific contexts, the key needs also to be deleted from the list corresponding to its associated context.

In some cases, a context may expire (for example if a transaction is rolled back), or a cache may decide to delete all objects corresponding to a context because the context is no longer of sufficient importance. In either case, the cache must delete all objects corresponding to the context. The set of keys associated to this context is retrieved by the cache, the objects identified by those keys are deleted, and so is the list containing the keys.

F. Locking Entries in the Cache

Some applications require basic locking mechanisms. Locking provides concurrency control that is often desired when building a software system in general. Concurrency control is also a mechanism necessary to enable transactions to execute concurrently in a manner that guarantees consistency. Locking as a general scheme provides some transactional support in the cache; the remainder can be in the application. The application can add specific methods to the cache in order to implement specific transaction semantics (see Section III-B.2). We allow the application writer to specify a set of lock types (by default, the list has two elements: READ and WRITE) as well as a table defining (by enumeration) sets of compatible locks. In many cases, the knowledge of the application structure is such that fine grain functional locks can be defined. In order to avoid deadlocks, locks have lifetimes associated with them. After the lifetime of a lock has expired without the lock being properly released, the lock is no longer valid, and the object is purged from the cache.

If a process attempts to grab a lock on an object incompatible with the one currently held, two scenarios are supported.

- The process waits for the lock to be freed (the maximum wait time is specified by the application) with the waiting processes handled in a FIFO manner.
- The process is not queued and is responsible for polling the cache to determine when the object is available.

III. Application-Level Caching in Real Settings

This section describes the design of application-level caching solutions in real settings. The solutions are based on extensions of a General-Purpose Software (GPS) Cache originally developed by A. Iyengar in [6]. The first specific caching solution we designed and implemented was for the WebSphere Accessible Business Rules framework. It is described in Section III-A. We are currently studying other applications and their caching needs.

Ad-hoc caches are often built by applications to act as fast storage for specific objects. Such caches may also support basic query mechanisms (beyond retrieve by key), update mechanisms specific to the type of objects stored, or specific interactions with a database (e.g., to prefetch several objects into the cache). Through ad-hoc use of database locks and structuring of the cache API, the application writer also often implements application transactional semantics. Section III-B, as an example, describes some desired caching features of an electronic market place exchange application.

Although the core GPS cache cannot provide all possible extensions, we are adding the basic extensions (e.g., locking, context support) described in Section II to the GPS cache and a layering approach which adds an application-specific API on the (extended) GPS cache to deal with complex API requirements specific to an application. When the GPS cache runs on a processor remote from the application,the application-specific API functions can be structured to run on either the cache or application processor.

Running the application-specific API functions on cache processors often results in greater efficiency because fewer remote calls to the cache are needed.

A. *Caching in the WebSphere Accessible Business Rules (ABR) framework*

WebSphere Accessible Business Rules (ABR) is a framework designed to facilitate the externalization of *business rules.* We use the terms *business rules* and *business logic* interchangeably to identify any logic that is subject to frequent change or for which it is undesirable to inaccessibly encapsulate as code within applications.

In the past, business rules were routinely embedded within application programs. The left-hand side of Figure 7 shows a monolithic application designed with embedded business logic. With the advent of ABR, the once embedded business logic is replaced by trigger points. The center portion of Figure 7 shows a rule-enabled application design containing trigger points accessing externalized business logic; at application program run time, whenever a trigger point is encountered, an appeal is made to the ABR rules system to run the externalized business logic.

ABR consists of several artifacts and components, but the two that are relevant here are trigger points and rules. Trigger points are ABR artifacts that are placed in application code in lieu of embedded business logic. Rules are ABR artifacts that trigger points utilize to execute desired externalized business logic. We refer to an application that contains one or more ABR trigger points that utilize ABR rules as a *rule-enabled application.*

Externalization of business logic has several benefits. Exposure of the business rules allows non-programmers to view and easily modify application behavior by simply manipulating persistent data within the ABR rules system. Reuse and sharing of business logic amongst applications becomes more feasible when rules can easily be obtained from a run time library, such as the ABR rules system. Other benefits can well be imagined. See [8] for more details about ABR.

However, there is one major potential disadvantage to such externalization of business logic: performance cost. If one or more calls to a remote server is necessary whenever a trigger point in a rule-enabled application needs to find and execute rules, then application performance will likely suffer. In order to gain the benefits of rule externalization via trigger points utilizing rules, while at the same time retaining good performance characteristics, a caching infrastructure is employed by ABR. The right hand side of Figure 7 shows a *rule-cached application* design containing trigger points accessing cached business logic.

Figure 8 depicts typical deployments using ABR. Tier 1 consists of application programs (web browsers running applets, Java applications, etc.) which may contain ABR trigger points. Tier 2 consists of web application servers and distributed object servers which may contain ABR trigger points as well. Tier 3 is the data source for the denizens of Tier 2. ABR caching trigger points interact with a particular Tier 2 distributed object server where the Enterprise Java Beans (EJB's) [9], [1] that contain the rules (i.e., the externalized business logic) are deployed. We call this particular server the *rule server.*

When a trigger point is encountered in a running rule-enabled application, the trigger point determines the business logic that needs to be applied. The trigger point uses context information to locate the one or multiple rules required. For example, a trigger point context might be "isSeniorCitizen" which would indicate that the rule(s) desired determine whether or not a person is considered to be a senior citizen. Absent any caching infrastructure, determining which rules to apply based upon the trigger point context is carried out by interacting with the remote rule server. The result of this determination is referred to as the trigger point context "query results" or "rule set".

We wanted ABR applications to perform as well or nearly as well as applications that were not rule-enabled. We hypothesized that caching ABR rule sets local to the trigger points would give greatly improved performance, nearly equal to that of embedded business logic. However, caching rule sets locally in this environment presented the difficult problem of one way communications between client (our trigger points) and server (our ABR EJB's) as dictated by the EJB programming model. That is, there was no architected way to have the server contact the client when rules changed on the server.

We made the following assertions and observations:

1. The use of ABR rules would be read mostly. Rules would not be created, deleted or updated very often.
2. It is desirable to have the rules sets cached local to trigger points.
3. Whenever the relatively rare event of a rule change occurs on the server, flushing the entire cache of rules is acceptable (though not very efficient) for the first release.
4. There is no EJB call back mechanism, so the cache will have to poll the server to detect rule changes occurring there. The EJB programming model is that of client-server. Clients make calls to objects located on the server. Objects on the server do not make calls to clients with change notifications.

Our cache is on the client. But there is no mechanism for the server to notify the client of changes occurring on the server that may affect the client-side cache. Thus, our client-side cache must poll the server to determine whether any changes that affect the state of the client-side cache have occurred.

As mentioned previously, we decided not to write a custom cache for ABR, but rather to employ a General-Purpose Software (GPS) Cache. We had to augment the GPS Cache by providing a polling mechanism to detect changes on the server in order to flush the cache as appropriate. We gave the ABR administrator the ability to control the polling interval; the default chosen was ten minutes.

Each application utilizing ABR by way of embedded trigger points incorporates a GPS Cache. We used the trigger point context as the key to store rule sets in and retrieve rule sets from the GPS Cache. Whenever a cache miss

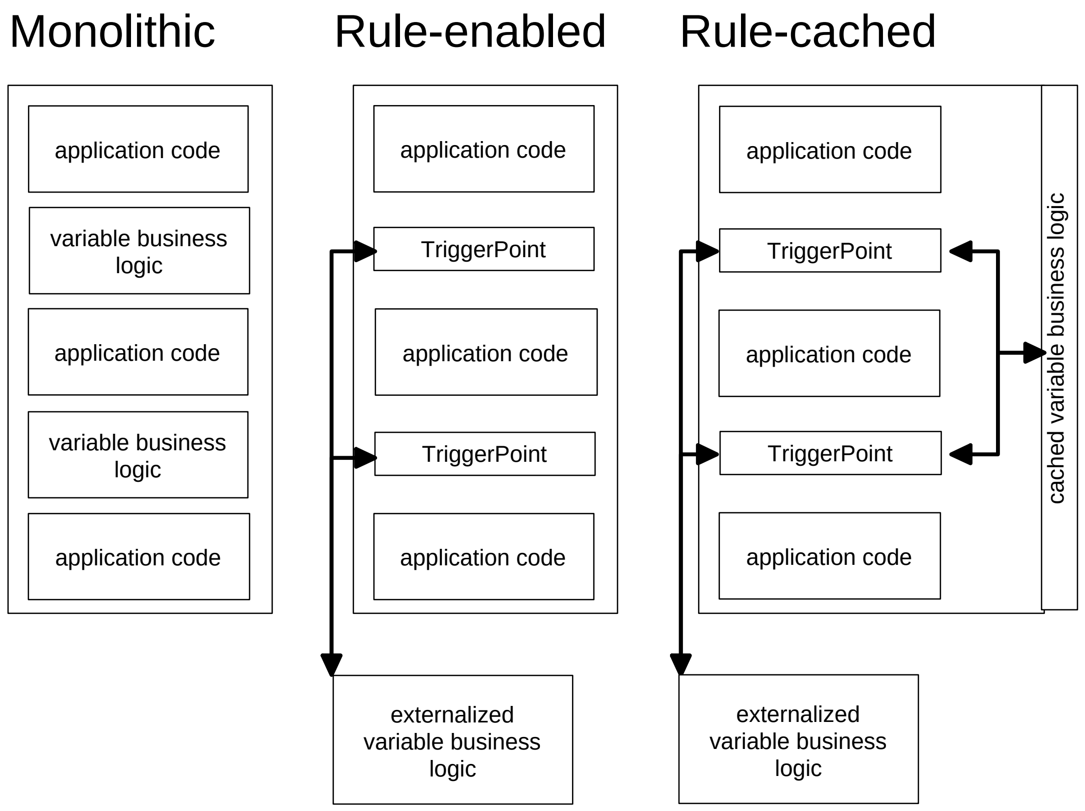

Fig. 7. Variable business logic is externalized using ABR.

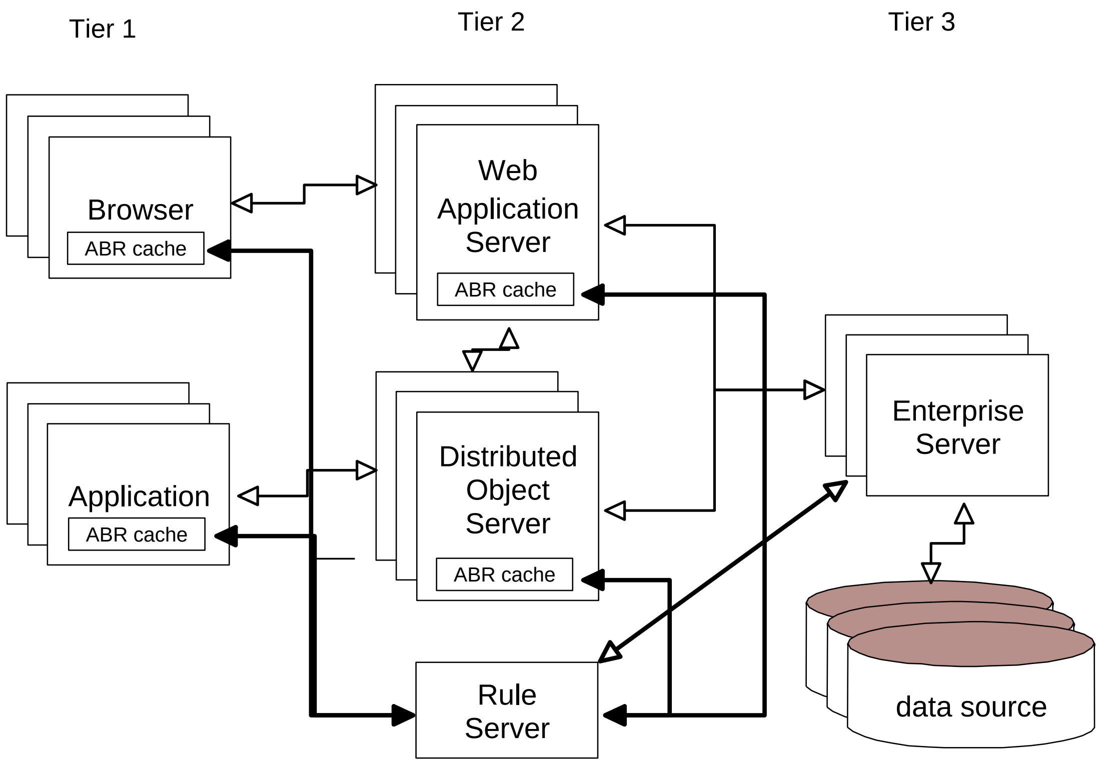

Fig. 8. A typical ABR deployment.

occurs for a given trigger point context, the remote rule server is interrogated by the trigger point to produce the required rule set, which is then cached for possible future use.

A rule set is a collection of individual rules. Each individual rule has its own start and end times. A start time and end time for the collection can be determined based upon the start and end times of the individual rules comprising the collection, as discussed in Section II-D. Since the GPS Cache is a time-aware cache, the maintenance of rule sets with respect to time is straightforward. This first version of ABR does not expand the time window to incorporate rules that have start times in the future or rules that have end times in the past into GPS Cached rule sets. Therefore, we did not need to implement the full algorithm described in Section II-D.

With regard to the algorithm described in Section II-C, when the cache becomes full, we rely upon the GPS Cache to remove rules sets based upon access frequency only. The other statistics, cost to fetch, update frequency, and cost to update are presumed to be the same for all rule sets.

We hope to have ABR more fully utilize the time-aware and statistical capabilities of the GPS Cache in future versions.

We ran some tests using a 666 MHz machine with 512 MB of memory running Windows 2000, DB2 6.1 fixpack 5, and WebSphere Advanced Edition 3.5.2. The client test application containing the trigger points and the ABR EJS were located on the same physical machine. No rules were changed during any of the tests, according to our first assumption. With no caching of rules, we witnessed a rate of 39 trigger points per second. Utilizing the GPS Cache, we saw a rate of 66,666 trigger points per second.

B. Caching in an Electronic Market Place Exchange System

Electronic market places bring buyers, sellers and other parties together and facilitate multilateral commerce transactions amongst the participants. In an exchange subsystem, a buyer and seller can perform the following actions.

- Create Exchange Positions (bids or asks) by filling up a form and posting the form over the web. Once posted, the position is in a draft state.
- Submit Exchange Position (bids or asks). After the create is done, the user can either modify the draft position, cancel, or submit the position for active trading.
- Cancel exchange position. Users can cancel exchange positions when in "draft" or" notmatched" states.
- Edit Exchange Position: Users can edit a position in a "notmatched" or "partially matched" state.
- ViewExchange Position: Users can view positions in all states.
- View Orderbook Exchange Positions: User can view the public orderbook of the exchange subsystem per offering.

The bids and asks submitted to the exchange subsystem are matched by the system. The positions are cached for real-time trading performance with the following application specific requirements.

- Positions submitted by users are kept in the cache if they are active positions.
- Positions that are active in the orderbook have the following states: "partially matched" or "not matched". All other positions are kept in a persistent state.
- Positions that change state when fully matched in the cache are moved into the database.
- Positions that are canceled are also moved into the database
- Positions that are edited by the user are moved into the database. A new position is created with the edition.

Fig 9 shows an overview of such a system.

B.1 Application-specific Manipulation and Update of Cached Objects

In the example described above, an entry in the cache is a set of positions for a particular offering (e.g., IBM stock). From this application perspective, it is desirable to be able to explicitly append a position to a particular entry or remove one. The application-specific augmentation of the cache introduces append and remove methods which in turn call the base GPS add, retrieve, and remove methods.

B.2 Supporting Application Transactional Behavior

In a number of cases, applications are structured in such a way that the application writer can enforce the transactional semantics of its applications relatively easily using the locking capabilities of the cache as well as its support for the notion of context. In particular, it has been our experience that in many cases, the application caches objects that are known to be read only or that the operations which access and modify those objects are well known to the application writer and can be explicitly defined (through enumeration or other unambiguous specifications). In other words, in those cases, unlike database systems, a programmer does not have to deal with enforcing transactional semantics without being able to make any (or close to any) assumptions about the way data will be accessed. Another point to mention is that in many cases, the transactional semantics of the applications differ from the traditional short ACID transactions supported by databases and other middleware systems. Correctness is defined by the application and differs from the traditional notion of ACID transaction isolation, which is that the transactions can always see a serializable view of the data [5]. This application-defined correctness makes application-driven solutions more appealing. [4] gives an overview of transactional cache consistency algorithms which support traditional ACID transaction semantics in the most general case (e.g. without assumptions on data accesses).

In electronic market place exchange systems, positions may be created before they are submitted to the exchange system for matching. When a user decides to submit a position, the position is then sent to the orderbook in the cache. The matching algorithm will match the positions if there are potential matches. The list of positions for a given offering at the time of the match needs to be "exact".

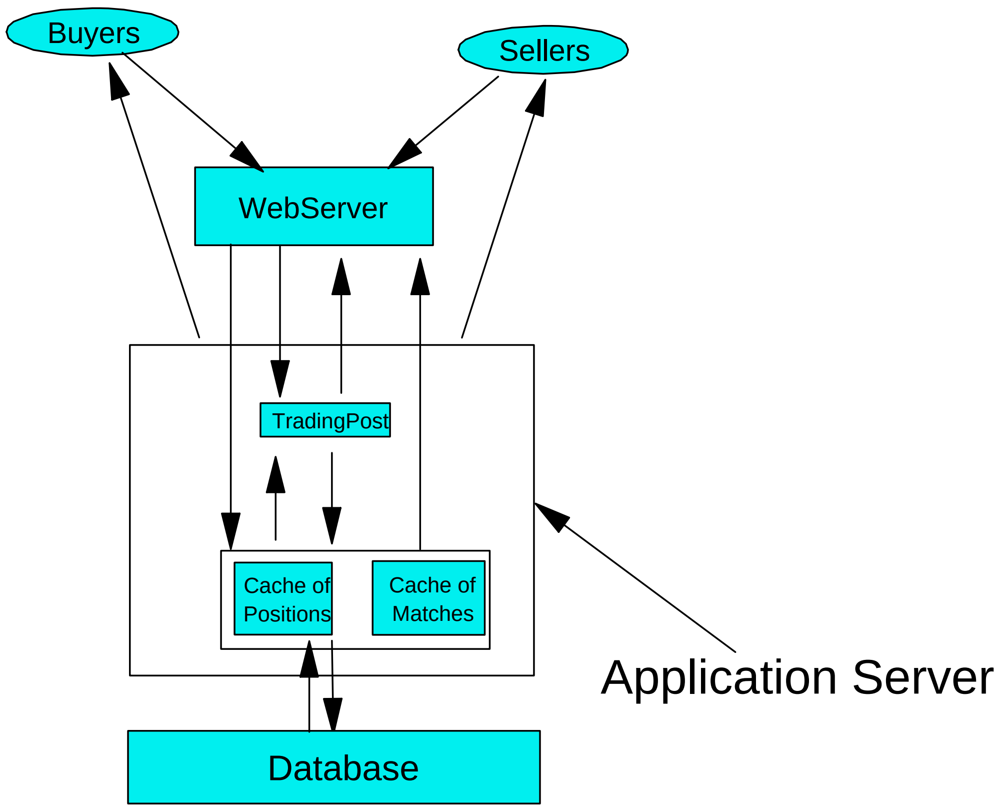

Fig. 9. An Electronic market place exchange system.

The interactions with the cache when submitting a position are illustrated below:

```
1 - Lock offering id associated with it.

2 - If the lock is successful, then:
    retrieve positions for the specific
    offering

    add position to cache

    match positions

    positions which are no longer active
    are pushed to the database

    if database fails, then

         reset lock in the cache

         return false, and ignore matched
         positions

    Else

         update the cache with the active
         positions

         purge positions that are no longer
         active from the cache

         reset lock

3 - If lock is not successful, then
      return normally without a match
```

IV. Conclusions

This paper has described our experience with application-caching. In order to improve the performance of software applications, we have designed a general-purpose software cache which provides a general-purpose API for all applications and the capability for a particular application to define specialized API functions. The cache provides limited transactional support via contexts and locking. Applications can provide additional transactional support via application-specific API functions.

We described our experiences with caching for the Accessible Business Rules framework for WebSphere. Applications such as these cache sets of objects whose lifetimes can be predicted in advance. We described an algorithm we have developed for automatically computing these sets by a cache. We also described some preliminary work we have done on caching in electronic market place exchange systems.

Acknowledgments

Jakka Sairamesh has collaborated with us on caching in electronic market place exchange systems. Thomas Mikalsen contributed to our understanding of transactional caching. Jon Peterson helped us obtain performance numbers for the ABR cache.

References

[1] Ayers et al. *Professional Java Server Programming.* Wrox Press, 1999.

[2] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[3] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM'99*, March 1999.

[4] M. Franklin, M. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternative and Performance. *ACM Transactions on Database Systems*, 22:315–363, September 1997.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1993.

[6] A. Iyengar. Design and Performance of a General-Purpose Software Cache. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99)*, February 1999.

[7] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[8] I. Rouvellou, L. Degenaro, K. Rasmus, D. Ehnebuske, and B. Mc Kee. Extending Business Objects with Business Rules. In *Proceedings of TOOLS Europe 2000*, 2000.

[9] Sun Microsystems Inc. Enterprise JavaBeans - Downloads and Specifications. http://java.sun.com/products/ejb/docs.html.

[10] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Server-Driven Consistency for Dynamic Web Services. In *Tenth International World Wide Web Conference Proceedings (WWW10)*, pages 45–57, May 2001.