

The Consolidated Enterprise Java Beans Design Pattern for Accelerating Large-Data J2EE Applications

Reinhard Klemm

Collaborative Applications Research Department

Avaya Labs Research

Basking Ridge, New Jersey, U.S.A.

Email: klemm@research.avayalabs.com

Abstract— J2EE is a specification of services and interfaces that support the design and implementation of Java server applications. A key concept in J2EE is *Entity Enterprise Java Beans* (EJBs). Their purpose is to persist the state of application objects and to share objects between transactions. Although typically desirable, the persistence in entity EJBs can also incur a heavy performance penalty. In this article, we describe a novel software design pattern aimed at improving the performance of entity EJBs in J2EE applications with large numbers of EJB instances. The pattern maps multiple real-world entities of the same type (e.g., users) to a single *consolidated* entity EJB (*CEJB*), thereby significantly reducing the number of required entity EJB instances. Consequently, CEJBs can increase EJB cache hit rates and database search performance. We present detailed quantitative assessments of performance gains from CEJBs and show that CEJBs can accelerate some common EJB operations in large-data J2EE applications by factors between 2 and 14.

Keywords—caching; Enterprise Java Beans; object consolidation; software design patterns; software performance

I. INTRODUCTION

Enterprise Java Beans (EJBs) [1] take advantage of a wide range of platform services from EJB containers in J2EE application servers. Examples of platform services are data persistence, object caching and pooling, object lifecycle management, database connection pooling, transaction semantics and concurrency control, entity relationship management, security, and clustering. EJB containers obviate the need for redeveloping such generic functionality for each application and thus allow developers to more quickly build complex and robust server-side applications. However, common EJB operations, in particular *entity* EJB operations, such as creating, accessing, modifying, and removing EJBs, tend to execute much more slowly than analogous operations for Java (J2SE) objects (*Plain Old Java Objects* or *POJOs*) that do not implement the functional equivalent of the J2EE platform service [2].

One of the platform services for entity EJBs that can incur a heavy performance penalty is data persistence. Although not mandated by the EJB specification, entity EJBs are typically stored as persistent objects in relational databases and we will assume this type of storage in the remainder of this article. Furthermore, we will concentrate on entity EJBs with container-managed persistence (CMP) rather than bean-managed persistence (BMP). CMP entity EJBs have the advantage of receiving more platform assistance than BMP entity EJBs and are thus usually

preferable from a software engineering point of view. They also tend to perform better than BMP entity EJBs because of extensive application-independent performance optimizations that EJB containers incorporate for CMP EJBs [3]. For the sake of simplicity, we will refer to CMP entity EJBs simply as “EJBs”. Note that the mapping from EJBs to database tables and the data transfer between in-memory (cached) EJBs and the database is the responsibility of the J2EE platform and can therefore be only minimally influenced by the EJB developer. Hence, we cannot discuss the impact of the technique presented in this article on structural or operational details of the data persistence layer of the J2EE platform. Instead, we will discuss how our technique changes the characteristics of the EJB layer that is under the control of the EJB developer and show how these changes affect the overall performance of EJB operations.

In the past, a lot of research into improving J2EE application performance has focused on tuning the configuration of EJBs and of the EJB operating environment consisting of J2EE application servers, databases, Web servers, and hardware. In addition, some software engineering methods such as software design patterns and coding guidelines have been developed to address performance issues with J2EE applications. This article presents a novel software design pattern for accelerating J2EE applications that we call *consolidated* EJBs (*CEJBs*). We devised the pattern during a multiyear research project at Avaya Labs Research where we developed a J2EE-based context aware communications middleware called *Mercury*. Mercury operates on a large number of EJB instances that represent enterprise users (hence our *User* EJB examples later in this article). Due to a large frequency of retrieval, query, and update operations on these EJBs, Mercury suffered from slow performance even after tuning J2EE application server and database settings. Thus, we felt compelled to investigate structural changes to Mercury’s J2EE implementation as a remedy for the performance problems and we arrived at the CEJB design pattern.

The remainder of this article is organized as follows. In Section II, we describe some of the related work. Section III presents the CEJB software design pattern and its use in J2EE applications. We describe the details of CEJB allocation, the mapping of entities to CEJBs, the storage of entities within CEJBs, and retrieval of entities from CEJBs. Our presentation focuses on EJBs according to the EJB 2.1 specification. This specification has been supplanted by the EJB 3.1 specification [4] in the meantime. However, the salient ideas of our work remain valid with EJB 3.1. We

compare the performance of CEJBs and EJBs in Section IV. A summary and an outline of future work conclude the article in Section V.

II. RELATED WORK

Much research has been devoted to speeding up J2EE applications by tuning EJBs and J2EE application server parameters. Pugh and Spacco [5] and Raghavachari et al. [6] discuss the potentially large performance impact and difficulties of tuning J2EE application servers, connected software systems such as databases, and the underlying hardware. In contrast, CEJBs constitute an application-level technique to attain additional J2EE application speed-ups.

The MTE project [7][8] offers more insight into the relationship between J2EE application server parameters, application structure, and application deployment parameters on the one hand and performance on the other hand. The MTE project underscores the sensitivity of J2EE application performance to application server parameters as well as to the application structure and deployment parameters.

Another large body of research into J2EE application performance has investigated the relationship between J2EE software design patterns and performance. Cecchet et al. [9] study the impact of the internal structure of a J2EE application on its performance. Many examples of J2EE design patterns such as the session façade EJB pattern can be found in [10] and [11], while Cecchet et al. [9] and Rudzki [12] discuss performance implications of selected J2EE design patterns. The CEJB design pattern improves specifically the performance of bean caches and database searches for EJBs. The Aggregate Entity Bean Pattern [13] consolidates logically dependent entities of *different* types into the same EJB while CEJBs consolidate entities of the *same* type into an EJB. Converting EJBs into CEJBs can therefore be automated by a tool whereas the aggregation pattern requires knowledge of the specific application and the logical dependencies of its entities. Aggregation and CEJBs can be synergistically used in the same application to increase overall execution speed.

Leff and Rayfield [14] show the importance of an EJB cache in a J2EE application server for improving application performance. We can find an in-depth study of performance issues with entity EJBs in [3]. The authors point out that caching is one of the greatest benefits of using entity EJBs provided that the bean cache is properly configured and entity EJB transaction settings are optimized.

The CEJB technique complies with the EJB specification and therefore can be applied to any J2EE application on any J2EE application server. Several J2SE-based technologies, from Java Data Objects (JDO) to Java Object Serialization (JOS), sacrifice the benefit of J2EE platform services in return for much higher performance than would be possible on a J2EE platform. Jordan [15] provides an extensive comparison of EJB data persistence and several J2SE-based data persistence mechanisms and their relative performance.

Trofin and Murphy [16] present the idea of collecting runtime information in J2EE application servers and to modify EJB containers accordingly to improve performance. CEJBs, on the other hand, do not change EJB containers but

improve performance by multiplexing multiple logical entities into one entity as seen by the EJB container.

III. CONSOLIDATED EJBs

A. CEJB Goal and Concept

CEJBs are intended to narrow the performance gap between EJBs and POJOs in J2EE applications with large numbers of EJBs of the same class. A look at common operations during the life span of an EJB explains some of the performance differences between EJBs and POJOs:

- Creating EJBs entails the addition of rows in a table in the underlying relational database at transaction commit time, whereas POJOs exist in memory.
- Accessing EJBs requires the execution of *finder* methods to locate the EJBs in the bean cache of the J2EE application server or in the database, whereas access to POJOs is accomplished by simply following object references.
- Depending on the selected transaction commit options (pessimistic or optimistic), the execution of business methods on EJBs is either serialized or requires frequent synchronization with the underlying database. Calling POJO methods, on the other hand, simply means accessing objects in the Java heap in memory, possibly with application-specific concurrency control in place.
- Deleting EJBs also removes the corresponding database table rows at commit time. Deleting POJOs affects only the Java heap in memory.

The preceding list identifies the interaction between EJBs and the persistence mechanism as a performance bottleneck for EJBs that POJOs do not suffer from. The persistence mechanism includes the bean cache and the database. One way of decreasing the performance gap between EJBs and POJOs, therefore, is to increase the bean cache hit rate, thereby reducing the database access frequency. In case of bean cache misses and when synchronizing the state of EJBs with the database, we would like to speed up the search for the database table rows that represent EJBs. CEJBs are intended to significantly decrease the number of EJBs in a J2EE application. A smaller number of EJBs translates into higher bean cache hit rates *and* faster EJB access in the database due to a smaller search space in database tables for EJB *finder* operations. In other words, CEJBs reduce the number and execution times of database accesses by increasing the rate of in-memory search operations.

CEJBs are based on a simple idea. Traditionally, when developing EJBs we map each real-world entity in the application domain such as a user to a separate EJB. This approach can result in a large number of EJB instances in the application. With CEJBs, on the other hand, we consolidate multiple entities of the same type into a single “special” EJB. Specifically, we store up to N POJO entities in the same EJB (the CEJB), where N is an priori determined constant. Because N is determined at application design time, the CEJB-internal data structure for storing entities can be an array of size N . Hence, locating an entity within a CEJB can be accomplished through a simple array indexing operation

requiring only constant time. The challenge for developing CEJBs is devising an appropriate mapping function $m:K_E \rightarrow K_C \times [0, N-1]$, where K_E is the primary key space of the entities and K_C is the primary key space of the CEJBs. Function m maps a given entity primary key k , for example a user ID, to a tuple (k_1, k_2) where

- k_1 is an artificial primary key for a CEJB that will store the entity,
- k_2 is the index of the array element inside the CEJB that stores the POJO with primary key k .

The mapping function m has to ensure that no more than N entities are mapped to the same CEJB. On the other hand, m also has to attempt to map as many entities to the same CEJB as possible. Otherwise, CEJBs would perform little or no better than EJBs. Moreover, the computation of m for a given entity primary key has to be fast.

B. Developing a CEJB

Consider a simple entity represented as an EJB *User* with the J2EE-mandated local home interface, local interface, and bean implementation:

- The local home interface is responsible for creating new *Users* through a method `create(String userID, String firstName, String lastName)` and finding existing ones through method `findByPrimaryKey(String userID)`.
- The local interface allows a client to call getter and setter methods for the `firstName` and `lastName` properties of *Users*. It also contains a method `businessMethod(String firstName, String lastName)` with some business logic: the method simply assigns its parameters to the `firstName` and `lastName` properties of a *User*, respectively.
- The bean implementation is the canonical bean implementation of the methods in the local (home) interfaces. For the sake of brevity, we omit showing the (quite trivial) bean implementation here.

In Figures 1-4, we present a CEJB *CUser* that we derived from the *User* EJB. To arrive at *CUser*, we first map the persistent (CMP) fields in *User* to transient *String* arrays `firstNames` and `lastNames` and persistent *String* fields `encodedfirstNames` and `encodedlastNames`. Note that we do not implement `firstNames` and `lastNames` as *persistent* array fields. Instead, we encode `firstNames` and `lastNames` as persistent *Strings* `encodedFirstNames` and `encodedLastNames`, respectively, during `ejbStore` operations. To do so, `ejbStore` creates a #-separated concatenation of all elements of `firstNames` and one of all elements of `lastNames` where # is a special symbol that does not appear in first or last names. This technique allows us to store the first names and last names as VARCHARs in the underlying database and avoid the much less time-efficient storage as VARCHARs for *bit data* that persistent array fields require. During `ejbLoad` operations the `encodedFirstNames` and `encodedLastNames` are being demultiplexed into the transient arrays `firstNames` and `lastNames`, respectively. The *CUserBean* then uses the state of the latter two arrays until

the next `ejbLoad` operation refreshes the state of the two arrays from the underlying database.

The `ejbCreate` method in Figure 3 assigns an *objectID* to the appropriate persistent field. We will discuss the choice of the *objectID* later. The method also allocates and initializes the transient `firstNames` and `lastNames` arrays. The size of the arrays is determined by the formal parameter N .

In the *CUser* local interface, we add an *index* parameter to all *getter* and *setter* methods and to the *businessMethod*. We also add the lifecycle methods `createUser` and `removeUser`. The *getter* and *setter* methods in *CUserLocal* have to be implemented by *CUserBean* because they are different from the abstract *getter* and *setter* methods in *CUserBean*. The new *getter* and *setter* methods access the indexed slot in the array fields `firstName` and `lastName`. Similarly, we have to change the *businessMethod*, which now accesses the indexed slot in the `firstName` and `lastName` fields rather than the entire EJB state. The `createUser` method first ensures that the indexed slots in the `firstNames` and `lastNames` are empty. If not, this user has been added before and a *DuplicateKeyException* is raised. If the slots are empty, `createUser` will assign the state of the new user to the indexed slots in the arrays. The `removeUser` method ensures that the indexed `firstNames` and `lastNames` slots are not empty, i.e., the referenced user is indeed stored in this *CUser*. If so, `removeUser` deletes the state of this user from the `firstNames` and `lastNames` arrays.

Figure 5 shows a class *ObjectIDMapping* that encapsulates an exemplary mapping function m from *User* primary keys (*Strings*) to *CUser* primary keys (*objectIDs*). Figure 6 contains an example of retrieving a *CUser* through an *ObjectIDMapping* and executing the *businessMethod* on the retrieved *CUser*. The only argument for the constructor of an *ObjectIDMapping* is N , the maximum number of entities consolidated in a *CUser*. The mapping function m is computed in the `setObjectID` method. This method maps a *User* primary key, `objectIDArg`, to the tuple $(objectID, index)$. The *objectID* is derived from `objectIDArg` by replacing `objectIDArg`'s last character c (viewed as an integer) with $c - index$. The value of *index* is the result of c modulo N , i.e., $c = q \cdot N + index$ where $0 \leq index < N$ and q is the integer quotient of c and N . While the *objectID* identifies the *CUser* in which we store an entity with `objectIDArg` as its primary key, the *index* identifies the slots in the CMP array fields in *CUser* that store the given entity. Although our definition of m is somewhat complex, its computation is fast and it maps at most N entities to each *CUser*, which is a key requirement for m .

C. Design Considerations for CEJBs

By creating a simple façade session bean we can completely hide *CUsers* from the rest of the application and expose only *POJO* entities to clients. With a façade session bean, the two-step process of first retrieving a *CUser* and subsequently accessing a *POJO* entity shown in Figure 6 is reduced to one step. The façade bean is straightforward and therefore we do not show it here. For more complicated

```

public interface CUserLocalHome extends EJBLocalHome {
    CUserLocal create(String objectID, int numElements) throws CreateException;
    CUserLocal findByPrimaryKey(String objectID) throws FinderException;
    CUserLocal getUser(String objectID, int numElements) throws FinderException;
}

```

Figure 1. Local home interface for *CUser*.

```

public interface CUserLocal extends EJBLocalObject {
    void createUser(int index, String firstName, String lastName) throws DuplicateKeyException;
    void removeUser(int index) throws RemoveException;
    String getFirstName(int index);
    void setFirstName(int index, String firstName);
    String getLastName(int index);
    void setLastName(int index, String lastName);
    void businessMethod(int index, String firstName, String lastName);
}

```

Figure 2. Local interface for *CUser*.

```

public abstract class CUserBean implements EntityBean {
    private transient String[] firstNames = null;
    private transient String[] lastNames = null;
    public abstract String getObjectID();
    public abstract void setObjectID(String objectID);
    public abstract String getEncodedFirstNames();
    public abstract void setEncodedFirstNames(String encodedFirstNames);
    public abstract String getEncodedLastNames();
    public abstract void setEncodedLastNames(String encodedLastNames);

    public String ejbCreate(String objectID, int N) throws CreateException {
        setObjectID(objectID);
        firstNames = new String[N];
        lastNames = new String[N];
        for (int index = 0; index < N; index++) {
            firstNames[index] = null;
            lastNames[index] = null;
        }
        return null;
    }

    public void ejbLoad() {
        StringTokenizer encodedFirstNames = new StringTokenizer(getEncodedFirstNames(), "#");
        encodedLastNames = new StringTokenizer(getEncodedLastNames(), "#");
        int numElements = encodedFirstNames.countTokens();
        if (firstNames == null) {
            firstNames = new String[numElements];
            lastNames = new String[numElements];
        }
        for (int index = 0; index < numElements; index++) {
            firstNames[index] = encodedFirstNames.nextToken();
            lastNames[index] = encodedLastNames.nextToken();
        }
    }

    public void ejbStore() {
        StringBuffer encodedNames = new StringBuffer();
        for (int index = 0; index < firstNames.length; index++) {
            encodedNames.append(firstNames[index]);
            encodedNames.append("#");
        }
        setEncodedFirstNames(encodedNames.toString());
        encodedNames.setLength(0);
        for (int index = 0; index < lastNames.length; index++) {
            encodedNames.append(lastNames[index]);
            encodedNames.append("#");
        }
        setEncodedLastNames(encodedNames.toString());
    }
}

```

Figure 3. Methods in *CUserBean* relevant to the CEJB discussion, part I.

```

public void createUser(int index, String firstName, String lastName) throws DuplicateKeyException {
    if (!(firstNames[index] == null && lastNames[index] == null)) throw new DuplicateKeyException("User exists already");

    firstNames[index] = firstName;
    lastNames[index] = lastName;
}

public void removeUser(int index) throws RemoveException {
    if (firstNames[index] == null || lastNames[index] == null) throw new RemoveException("User does not exist");
    firstNames[index] = "";
    lastNames[index] = "";
}

public void businessMethod(int index, String firstName, String lastName) {
    firstNames[index] = firstName;
    lastNames[index] = lastName;
}

public void setFirstName(int index, String firstName) {
    firstNames[index] = firstName;
}

// other getter/setter methods go here...
}

```

Figure 4. Methods in *CUserBean* relevant to the CEJB discussion, part II.

```

public class ObjectIDMapping {
    private int N,
        index;
    private String objectID;

    public ObjectIDMapping(int N) {
        this.N = N;
        index = -1;
        objectID = null;
    }

    public void setObjectID(String objectIDArg) {
        int lastElementIndex = objectIDArg.length() - 1,
            lastCharacter = objectIDArg.charAt(lastElementIndex);

        index = lastCharacter % N;
        objectID = objectIDArg.substring(0, lastElementIndex) + (lastCharacter - index);
    }

    public int getIndex() {
        return index;
    }

    public String getObjectID() {
        return objectID;
    }
}

```

Figure 5. Class for mapping *User* primary keys to *CUser* primary keys and array index slots.

```

ObjectIDMapping idMapping = new ObjectIDMapping(N);
idMapping.setObjectID("rKlemm");
CUserLocal cUser = cuserLocalHome.findByPrimaryKey(idMapping.getObjectID());
cUser.businessMethod(idMapping.getIndex(), "Reinhard", "Klemm");

```

Figure 6. Accessing a *CUser* EJB.

entities than *Users*, consolidation through CEJBs requires more effort but is straightforward and could be supported by a tool. Ideally, such a tool would be offered as part of a J2EE development environment and convert EJBs into CEJBs at the request and under the directions of the developer. The tool would also need to support the following scenarios:

- If *User* implements customized *ejbLoad*, *ejbStore*, *ejbActivate*, or *ejbPassivate* methods, these need to be adapted in *CUserBean* to reflect the fact that the state of a *User* is stored across different arrays in the *CUserBean*.
- *Finder* and *select* queries for *User* must be re-implemented for the CEJB because they need to access both a *CUser* and the arrays within a *CUser*.
- If *User* has customized *ejbHome* methods, we need to add functionally equivalent *ejbHome* methods to

CUser. Changes to the original *User ejbHome* methods are only necessary if these methods access the state of a specific *User* EJB after a prior *select* method. In this case, the *CUser ejbHome* methods need to retrieve *POJO* entities instead of *Users*.

- If *User* is part of a container-managed relationship (CMR), consolidation through CEJBs requires removal of the CMRs and manual re-implementation of the CMRs without direct J2EE support.

The mapping function m has a strong impact on the performance of CEJBs and therefore needs to be defined carefully for the given application. The mapping function delivers its best performance if primary keys that occur in the application are *clustered*. Clustering here means that for every primary key k in the application there is a set of roughly N primary keys for other entities in the application that are similar enough to k to be mapped to the same *objectID* by m . The challenge is therefore to analyze the actual key space of the entities that are to be consolidated in a given application and to then define an efficient and effective mapping function based on this analysis.

IV. PERFORMANCE EVALUATION

A. Methodology

We compared the performance of *Users* and *CUsers* in a J2EE test application. It uses the mapping function m in Figure 5 because this function clusters the primary keys that we chose for the entities in the test application - lexicographically consecutive strings - to facilitate the generation of a large number of user entities. The test application executes a sequence of operations either on *Users* (*EJB mode*) or *CUsers* (*CEJB mode*). In EJB mode, the application executes the following sequence of steps:

1. Create n *User* EJBs.
2. Find *User* EJB with randomly selected primary key and read its state through *getter* operations. Repeat n times.
3. Find *User* EJB with randomly selected primary key and execute *businessMethod* on it, thus changing the EJB state. Repeat n times.
4. Delete all *User* EJBs through EJB *remove* operations.

Between any two consecutive steps, the test application creates 20000 unrelated EJBs in order to introduce as much disturbance as possible in the application server bean cache and in the connection to the underlying database. During our performance testing, however, it turned out that these cache disturbance operations had a negligible effect on the performance *differences* between the CEJB and EJB modes.

In CEJB mode, the application performs the same steps on *CUsers* instead of *Users*. Also, in step 4 in CEJB mode, the application sequentially deletes all entities in each CEJB but not the CEJB itself. We varied the maximum number N of entities per CEJB, from 2 to 250 in consecutive runs of the test application. The performance of the test application peaked around $N=20$. We only present the performance results for $N=20$.

We configured the test application with two different transaction settings in two different experiments: in *long transaction mode*, each step of the test application is executed in one long-lived transaction. In *short transaction mode*, the application commits every data change as soon as it occurs, i.e., after each entity creation, change, or deletion. Here, the application performs a large number of short-lived transactions. In successive runs of the test application, n iterated over the set $\{1000, 10000, 50000\}$. After each run, we restarted the database server and the application server and deleted all database rows created by the application.

We deployed the test application on an IBM WebSphere 5.1.1.6 J2EE application server with default bean cache and performance settings. The hardware is a dual Xeon 2.4 GHz server running Microsoft Windows 2000 Server. An IBM DB2 8.1.9 database provides the data storage. All EJBs use the WebSphere default commit option C.

B. Performance Analysis

Figures 7-12 display the results of our performance testing with the test application in long and short transaction modes for the three different values of n . The speedup in the figures is defined as the time for an EJB operation divided by the time for the equivalent CEJB operation. Speedup values greater than 1 indicate results where CEJBs outperform EJBs, values of less than 1 indicate EJBs performing better than CEJBs. In long transaction mode, CEJBs significantly outperformed EJBs. For $n=50000$, for example, creating users with CEJBs was more than twice as fast as with EJBs, finding and reading users was more than 5 times faster, finding and changing users was more than 7 times faster, and deleting users with CEJBs was more than 14 times faster.

Because the mapping function m in our test application clusters the primary keys of the user entities, the CEJBs consolidate almost the maximum possible number of entities (20 per our definition of N). Hence, the number of CEJBs necessary to store all user entities in the test application is about $1/20^{\text{th}}$ that of the number of EJBs in EJB mode, which translates into much improved application server caching behavior and accelerated database search times. Once a CEJB has been retrieved, extracting the desired entity from the CEJB is a simple and fast array indexing operation. However, if the chosen mapping function m for a given application does not achieve the cluster property, CEJBs may lose some of their performance advantage over EJBs.

In CEJB mode, entity deletion does not force the deletion of EJBs in the application server or the database. Instead, entity deletion in CEJBs is accomplished through the removal of entities *inside* EJBs. Not surprisingly therefore, deleting users in CEJB mode is much faster than in EJB mode where an EJB needs to be removed in the application server bean cache and the underlying database.

In short transaction mode, our performance testing showed a very different outcome. Here, CEJBs only offer performance advantages over EJBs for finding and reading users operations. CEJBs are about as fast as EJBs during finding and changing of users and during deletion of users but much slower in creating users. In short transaction mode, transaction commits after EJB state changes dominate the

execution time of the test application and void many performance advantages due to consolidation. Hence, J2EE applications that eagerly commit every EJB state change will experience a significant speed-up as a result of consolidation only if the EJB read to write ratio is very high.

In conclusion, CEJBs provide strong performance advantages over EJBs in a J2EE application if (1) the application contains a large number of EJBs, (2) it accesses EJBs either in long-lived transactions or in short-lived transaction with a large EJB read to write ratio, and (3) if a mapping function m can be found for the EJB key space that exhibits the cluster property.

Our test application is designed to execute a large number of common EJB operations in a repeatable fashion. As such, the test application is somewhat artificial. It does not involve human interactions and arbitrary timing delays due to human input. The pattern of EJB operations is highly regular and maximizes EJB accesses, whereas other J2EE applications may have irregular EJB accesses and also contain computationally or I/O-intensive tasks. Our *User* EJBs are simple while EJBs in common J2EE applications can be more complex and linked to each other. However, we believe that our test application realistically captures the performance differences between EJBs and CEJBs in a large class of J2EE applications that are characterized by large numbers of entities, a high frequency of EJB accesses with a large degree of regularity (e.g., certain data mining applications such as our Mercury system), and a predictable and regular primary key space for the entities.

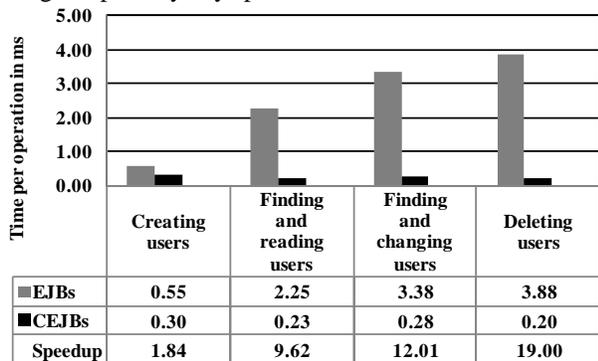


Figure 7. Test application performance: long transaction mode, n=1000.

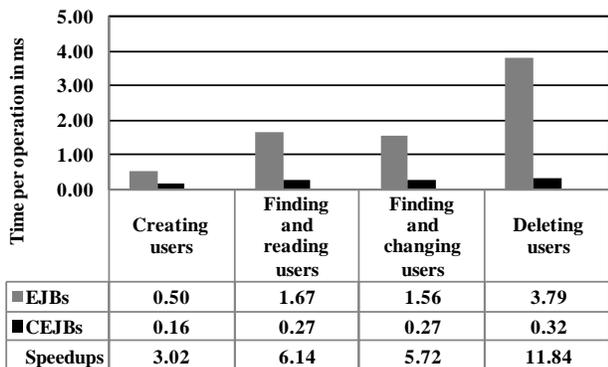


Figure 8. Test application performance: long transaction mode, n=10000.

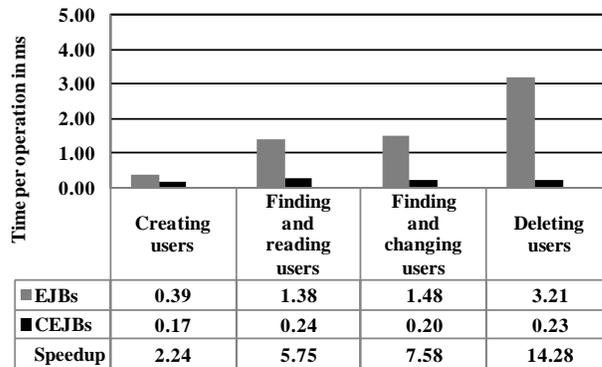


Figure 9. Test application performance: long transaction mode, n=50000.

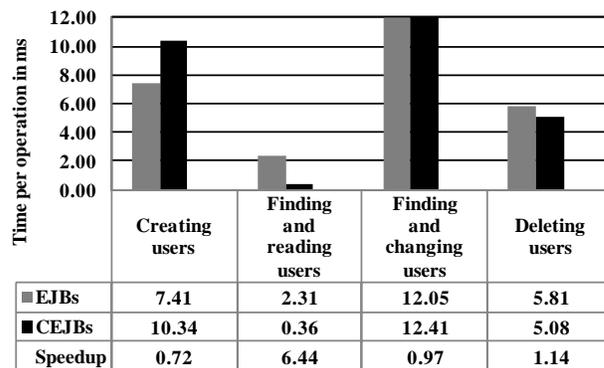


Figure 10. Test application performance: short transaction mode, n=1000.

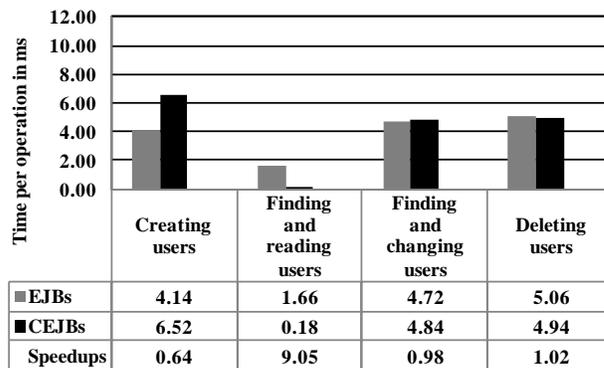


Figure 11. Test application performance: short transaction mode, n=10000.

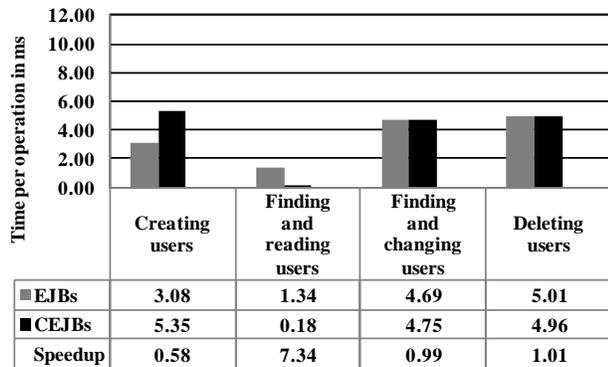


Figure 12. Test application performance: short transaction mode, n=50000.

V. CONCLUSION AND FUTURE WORK

We presented a J2EE software design pattern that consolidates multiple entities in J2EE applications into special-purpose entity EJBs that we call consolidated EJBs (CEJBs). Consolidation increases the locality of data access in J2EE applications, thus making bean caching in the application server more effective and decreasing search times for entity EJBs in the underlying database. In J2EE applications with large numbers of EJBs, CEJBs can therefore greatly increase the overall application performance. Using a test application we showed that CEJBs can outperform traditional EJBs by a wide margin for common EJB operations. For example, the CEJB equivalent of an EJB *findByPrimaryKey* operation is more than five times faster in one of our experiments, and the execution of a data-modifying business method on an EJB is more than seven times faster in CEJBs. CEJBs conform to the EJB specification and can therefore be used in any J2EE application on any J2EE application server.

We have three future research goals for CEJBs. First, we would like to modify CEJBs in such a way that applications with short-lived transactions and a small ratio of EJB read to EJB write operations perform better than our current solution. Secondly, we intend to investigate mapping functions for CEJBs that (1) perform well if the primary key space for EJBs is irregular or unpredictable, and (2) that can be automatically defined without requiring complex developer decisions. Thirdly, we would like to address a currently open question for our CEJB design pattern, which is how to adjust CEJBs so that they are beneficial in most J2EE applications and thus could ultimately become a standard way of implementing entities in J2EE applications.

REFERENCES

[1] Oracle Inc., “Enterprise JavaBeans Specification 2.1,” retrieved September 28, 2012, from <http://bit.ly/Ovip59>.

[2] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, “Performance Comparison of Middleware Architectures for Generating Dynamic Web Content,” Lecture Notes in Computer Science, Vol. 2672, Jan. 2003, pp. 242-261.

[3] S. Kounev and A. Buchmann, “Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services,” Proc. 28th International Conference on Very Large Databases (VLDB), Aug. 2002, retrieved September 28, 2012, from <http://bit.ly/QgduUf>.

[4] Oracle Inc., “Enterprise JavaBeans Specification 3.1,” retrieved September 28, 2012, from <http://bit.ly/SIMyPN>.

[5] S. Pugh and J. Spacco, “RUBiS Revisited: Why J2EE Benchmarking is Hard,” Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Oct. 2004, pp. 204-205.

[6] M. Raghavachari, D. Reiner, and R. Johnson, “The Deployer’s Problem: Configuring Application Servers for Performance and Reliability,” Proc. 25th International Conference on Software Engineering ICSE ’03, May 2003, pp. 484-489.

[7] S. Ran, P. Brebner, and I. Gorton, “The Rigorous Evaluation of Enterprise Java Bean Technology,” Proc. 15th International Conference on Information Networking (ICOIN), IEEE Computer Society, Jan. 2001, pp. 93-100.

[8] S. Ran, D. Palmer, P. Brebner, S. Chen, I. Gorton, J. Gosper, L. Hu, A. Liu, and P. Tran, “J2EE Technology Performance Evaluation Methodology,” Proc. International Conference on the Move to Meaningful Internet Systems 2002, pp. 13-16.

[9] E. Cecchet, J. Marguerite, and W. Zwaenepoel, “Performance and Scalability of EJB Applications,” Proc. 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA), Nov. 2002, retrieved September 28, 2012, from <http://bit.ly/Qge98e>.

[10] D. Alur, J. Crupi, and D. Malks, “Core J2EE Patterns,” Prentice Hall/Sun Microsystems Press, Jun. 2001.

[11] F. Marinescu, “EJB Design Patterns: Advanced Patterns, Processes, and Idioms,” John Wiley & Sons Inc., Mar. 2002.

[12] J. Rudzki, “How Design Patterns Affect Application Performance – A Case of a Multi-Tier J2EE Application,” Lecture Notes in Computer Science, No. 3409, Springer-Verlag, 2005, pp. 12-23.

[13] C. Larman, “The Aggregate Entity Bean Pattern,” Software Development Magazine, Apr. 2000, retrieved September 28, 2012, from <http://bit.ly/PgBoxe>.

[14] A. Leff and J. T. Rayfield, “Improving Application Throughput with Enterprise JavaBeans Caching,” Proc. 23rd International Conference on Distributed Computing Systems (ICDCS), May 2003, pp. 244-251.

[15] M. Jordan, “A Comparative Study of Persistence Mechanisms for the Java Platform,” Sun Microsystems Technical Report TR-2004-136, Sep. 2004, retrieved September 28, 2012, from <http://bit.ly/U3GGPf>.

[16] J. Trofin and J. Murphy, “A Self-Optimizing Container Design for Enterprise Java Beans Applications,” 8th International Workshop on Component Oriented Programming (WCOP), Jul. 2003, retrieved September 28, 2012, from <http://bit.ly/O4biAD>.