Enhancing the Performance of J2EE Applications through Entity Consolidation Design Patterns

Reinhard Klemm
Collaborative Applications Research Department
Avaya Labs Research
Basking Ridge, New Jersey, U.S.A.
klemm@research.avayalabs.com

***Abstract*— J2EE is a specification of services and interfaces that support the design and implementation of Java server applications. Persistent and transacted *entity Enterprise JavaBean* objects are important components in J2EE applications. The persistence and transaction semantics of entity Enterprise JavaBeans, however, lead to a sometimes significantly decreased performance relative to traditional Java objects. From an application performance point of view, a J2EE-compliant object persistence and transaction mechanism with a lower performance penalty would be highly desirable. In this article, we present and evaluate two J2EE software design patterns aimed at enhancing the performance of entity Enterprise JavaBeans in J2EE applications with large numbers of JavaBean instances. Both design patterns consolidate multiple real-world entities of the same type, such as *users* and *communication sessions*, into a single *consolidated* entity Enterprise JavaBean. The entity consolidation results in a smaller number of entity JavaBean instances in a given J2EE application, thereby increasing JavaBean cache hit rates and database search performance. We present detailed experimental assessments of performance gains due to entity consolidation and show that consolidated Enterprise JavaBeans can accelerate common JavaBean operations in large-data J2EE applications by factors of more than 2.**

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

I. INTRODUCTION

In this article, we extend our earlier work on performance-enhancing J2EE software design patterns published in [1]. To make the article self-contained and thus easier to read, we include a comprehensive description of the research presented in [1]. The focus of our work is entity Enterprise JavaBeans (EJBs) [2]. Entity EJB objects take advantage of a plethora of platform services from EJB containers in J2EE application servers [3]. 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.

A common and important component in J2EE application servers is an in-memory EJB cache that speeds up access to entity EJBs in an application's working set [4]. Yet, common entity EJB operations such as creating, accessing, modifying, and removing entity EJBs tend to execute much more slowly than analogous operations for traditional Java objects (J2SE objects, also often referred to as *Plain Old Java Objects* or simply *POJOs*) that do not implement the functional equivalent of the J2EE platform services. The performance of data-intensive J2EE applications, i. e., those with large numbers of entity EJBs, can therefore be much slower than desired.

Although not mandated by the EJB specification, entity EJBs are typically stored as rows in relational database tables 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 [5]. 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 cached EJBs and the database is the responsibility of the proprietary J2EE platform and can therefore be only minimally influenced by the EJB developer. Hence, we cannot discuss the direct impact of the design patterns 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, much 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 two J2EE software design patterns for accelerating J2EE applications. Both patterns result in specialized EJBs that we call *consolidated* EJBs (*CEJBs*). By applying the first pattern, we obtain *fixed-size* consolidated EJBs (*fCEJBs*). Fixed-size CEJBs are the topic of our earlier work published in [1]. The second, new pattern generates *variable-size* consolidated EJBs (*vCEJBs*). Both CEJB patterns attempt to optimize the caching and database storage of EJBs

for enhanced execution speed of common EJB operations (creating, accessing, modifying, and removing entities).

We devised these two software design patterns 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 and communication sessions (hence our *User* and *Session* EJB examples later in this article). Due to the 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 arrived at the CEJB design patterns. The technical discussion in this article will show that our design patterns are more generally applicable in a wide range of J2EE applications.

The J2EE and entity Enterprise JavaBeans specifications that we refer to in this article have meanwhile been supplanted by updated standards and with a new terminology: The J2EE 1.4 specification has been replaced with Java EE 6 [6], and the entity EJBs in the Enterprise JavaBeans specification 2.1 have been replaced with entities according to the Java Persistence API 2.0 [7]. The software design patterns in this article remain equally relevant in the context of the new specifications and require mostly syntactic changes.

The remainder of this article is organized as follows. In Section II, we describe some of the related work. Section III contains an overview of the key idea behind both CEJB software design patterns. Section IV presents the fCEJB pattern and its use in J2EE applications. We describe the details of fCEJB allocation, the mapping of entities to fCEJBs, the storage of entities within fCEJBs, and retrieval of entities from fCEJBs. Similarly, Section V contains a detailed explanation of the vCEJB pattern. We compare the performance of fCEJBs, vCEJBs, and traditional EJBs in Section VI. A summary and an outline of future work conclude the article in Section VII.

II. Related Work

The performance penalty of using EJBs in J2EE applications has been well documented in the relevant literature, some of which we review in this section. A substantial number of articles present various remedies for this performance penalty, ranging from performance-tuning of application servers to alternative object persistence mechanisms to performance-enhancing EJB software design patterns. However, to our knowledge, our CEJBs are the first application-level approach that yields verified, substantial performance improvements in a wide range of J2EE applications where alternatives to EJBs are not acceptable, practical, or desirable. In our earlier research presented in [1] we introduced fCEJBs as a performance-enhancing J2EE software design pattern. However, in the presence of entities that do not have the cluster property that we describe in Section IV, fCEJBs perform no better than traditional EJBs. Our new vCEJB design pattern aims at addressing this shortcoming of fCEJBs.

Much research has been devoted to speeding up J2EE applications by tuning EJBs and J2EE application server parameters. Pugh and Spacco [8] and Raghavachari et al. [9] 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 [10][11] 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. [12] 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 [13] and [14], while Cecchet et al. [15] and Rudzki [16] discuss performance implications of selected J2EE design patterns. The CEJB design patterns improve specifically the performance of EJB caches and database searches for EJBs. The Aggregate Entity Bean Pattern [17] 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. No performance measurements are reported in [17].

Leff and Rayfield [4] 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 [5]. The authors point out that caching is one of the greatest benefits of using entity EJBs provided that the EJB cache is properly configured and entity EJB transaction settings are optimized.

Our CEJB design patterns comply 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 [18] provides an extensive comparison of EJB data persistence and several J2SE-based data persistence mechanisms and their relative performance. The comparison includes EJB, JDO, Java Database Connectivity (JDBC), Orthogonal Persistence (OPJ), JavaBeans Persistence (JBP), and Java Object Serialization (JOS). Interestingly, the comparison revealed that EJBs had the worst performance among the compared persistence mechanisms, while JDOs had the best

performance. The author states that "acceptable EJB performance seems unattainable at present unless dramatic changes are made to the application object model to avoid fine-grain objects when mapped to EJB". The fCEJB and vCEJB design patterns are an application-level approach to avoiding the mapping of fine-grained objects to EJBs and thus the performance penalty associated with using EJB-based persistence in J2EE applications. Not included in the study in [18] is another popular J2SE persistence mechanism, *Hibernate*. The performance of Hibernate – in comparison to the object database *db4o*, but not in comparison to EJBs – is discussed in [19].

Trofin and Murphy [20] 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, execute in unmodified EJB containers and improve performance by multiplexing multiple logical entities into one entity as seen by the EJB container.

III. CEJB Goals and Concept

The intention of both of our CEJB software design patterns is to narrow the performance gap between EJBs and POJOs in J2EE applications with large numbers of EJBs. 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 only in memory.
- Accessing EJBs requires the execution of *finder* methods to locate the EJBs in the EJB 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 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 implies the removal of the EJB objects from the EJB cache, if they are stored there, and the deletion of 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 (EJB cache plus database) as a performance bottleneck for EJBs that POJOs do not suffer from. One way of decreasing the performance gap between EJBs and POJOs, therefore, is to increase the EJB cache hit rate, thereby reducing the database access frequency. In case of EJB 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 EJB 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 to a separate EJB. Examples of such entities are users and communication sessions, to stay with the example of the Mercury system in Section I. 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. The difference between fCEJBs and vCEJBs is in the way the entities are organized within each CEJB and the resulting impact on the overall pool of CEJBs. In the remainder of this article, when we speak of "entities", we implicitly assume "entities of the same type" unless otherwise noted.

IV. Fixed-Size Consolidated EJBs

In this section, we present the key idea, design methodology, and some practical aspects of developing fCEJBs.

A. Concept of the fCEJB Pattern

In the case of fCEJBs, we store up to N POJO entities in the same EJB (the fCEJB), where N is a constant that is determined at application design time. We store the entities in arrays of size N inside the fCEJB. Hence, locating an entity within an fCEJB can be accomplished through simple array indexing operations requiring only constant time. The challenge for developing fCEJBs 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 real-world entities and K_C is the primary key space of the fCEJBs. Function m maps a given entity primary key k, for example a communication session ID, to a tuple (k_1, k_2) where

- k_1 is an artificial primary key for an fCEJB that will store the entity,
- k_2 is the index of the array elements inside the fCEJB that store the POJO with primary key k.

The mapping function m has to ensure that no more than N entities are mapped to the same fCEJB. On the other hand, m also has to attempt to map as many entities to the same fCEJB as possible. Otherwise, fCEJBs 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 an fCEJB

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

- The local home interface is responsible for creating new *Sessions* through a method *create(String sessionID, long startTime)* and finding existing ones

through method *findByPrimaryKey(String sessionID)*.

- The local interface allows a client to call getter and setter methods for the *sessionID* and *startTime* properties of *Sessions*. It also contains a method *businessMethod(long newStartTime)* that changes the value of the *startTime* of the EJB.
- The bean implementation is the canonical bean implementation of the methods in the local and local home interfaces. For the sake of brevity, we omit details of the (quite trivial) bean implementation here.

In Figures 1-3, we present an fCEJB *CSession* that we derive from the *Session* EJB. To arrive at *CSession*, we first map the persistent (CMP) fields in *Session* to

- a transient *String* array *sessionIDs*,
- a transient *long* array *startTimes*,
- a persistent *String* field *encodedSessionIDs*,
- and a persistent *String* field *encodedStartTimes*,

as shown in lines 2-9 in Figure 3. Note that we do not implement *sessionIDs* and *startTimes* as *persistent* array fields. Instead, we encode *sessionIDs* and *startTimes* as persistent *Strings encodedSessionIDs* and *encodedStartTimes*, respectively, during J2EE *ejbStore* operations (Figure 3, lines 32-45). To do so, *ejbStore* creates a #-separated concatenation of all elements of *sessionIDs* and one of all elements of *startTimes* where # is a special symbol that does not appear in *sessionIDs* or *startTimes*. This technique allows us to store the sessionIDs and start times as VARCHARs in the underlying database and avoid the much less time-efficient storage as VARCHAR *for bit data* that persistent array fields require. During J2EE *ejbLoad* operations (Figure 3, lines 18-30), the *encodedSessionIDs* and *encodedStartTimes* are being demultiplexed into the transient arrays *sessionIDs* and *startTimes*, respectively. The *CSessionBean* 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, lines 11-16, assigns an *objectID* to the persistent *objectID* field. We will discuss the choice of the *objectID* later. The method also allocates and initializes the transient *sessionIDs* and *startTimes* arrays. The size of the arrays is determined by the formal parameter *N*.

In the *CSessionLocal* interface in Figure 2, we add an *index* parameter to all getter and setter methods and to the *businessMethod*. We also add the lifecycle methods *createSession* and *removeSession*. The getter and setter methods in *CSessionLocal* with the *index* parameter have to be implemented by *CSessionBean* because they are different from the abstract getter and setter methods in *CSessionBean* that are applied to the persistent *encodedSessionIDs* and *encodedStartTimes* fields. The new getter and setter methods access the indexed slot in the array fields *sessionIDs* and *startTimes*. An example of a setter method is shown in lines 62-64 in Figure 3. Similarly, we have to change the *businessMethod*, which now accesses the indexed slot in the transient *sessionIDs* and *startTimes* arrays rather than operating on persistent entity fields (lines 58-60 in Figure 3). The *createSession* method in lines 47-51 in Figure 3 first ensures that the indexed slots in the *sessionIDs* and *startTimes* are empty. If not, this session has been added before and a *DuplicateKeyException* is raised. If the slots are empty, *createSession* will assign the state of the new communication session to the indexed slots in the arrays. The *removeSession* method in lines 53-56 in Figure 3 ensures that the indexed *sessionIDs* and *startTimes* slots are not empty, i. e., the referenced session is indeed stored in this *CSession*. If so, *removeSession* deletes the state of this communication session by setting the indexed slot in the *sessionIDs* to *null*.

Figure 4 shows a class *ObjectIDMapping* that encapsulates an exemplary mapping function *m* from *Session* primary keys (*Strings*) to *CSession* primary keys (*objectIDs*). We will discuss *m* in conjunction with the code example given in Figure 5 that retrieves a *CSession* through an *ObjectIDMapping* and executes the *businessMethod* on the retrieved *CSession*. The argument for the constructor of an *ObjectIDMapping* is *N*, the maximum number of entities consolidated in a *CSession*, as shown in line 6 in Figure 4. The mapping function *m* is computed by a call to the *setObjectID* method in line 2 in Figure 5. This method maps a *Session* primary key, *objectIDArg*, to the tuple (*objectID*, *index*). In Figure 5, the *Session* primary key is *voiceCall-05-12-2012a*. The *objectID* is derived from *objectIDArg* by replacing *objectIDArg's* last character *c* with an underscore followed by *c – index*, where we interpret *c* as the ordinal value of the character in the ASCII character table (lines 14 and 16 in Figure 4). In line 15 in Figure 4, the value of *index* is computed as the result of the operation

$$c \; modulo \; N,$$

i. e.,

$$c = q \cdot N + index,$$

where

$$0 \leq index < N,$$

and q is the integer quotient of c and N. In our example, *c* is the ordinal value of *a*, the last character of *voiceCall-05-12-2012a*, so *c* = 97. If we assume *N* = 20, then *index* = 17, and *c – index* = 80. Therefore, *objectID = voiceCall-05-12-2012_80*. While *getObjectID()* (line 3, Figure 5) identifies the *CSession* in which we store an entity with *objectIDArg* as its primary key, *getIndex()* (line 4, Figure 5) identifies the slots in the CMP array fields in the *CSession* that store the given entity. In the example, the real-world entity with primary key *voiceCall-05-12-2012a* is thus stored in slot 17 in the *CSession* with primary key *voiceCall-05-12-2012_80*. Figure 6 depicts the mapping from the *Session* primary key *voiceCall-05-12-2012a* to *CSession* primary key *voiceCall-05-12-2012_80* and slot 17 in the *CSession*.

Although our definition of *m* is somewhat complex, its computation is fast and it maps at most *N* entities to each

CSession, which is a key requirement for *m*. If the *Session* primary keys had numerical suffixes such as *100*, *101*, *102* instead of alphabetical suffixes *a*, *b*, *c*, and so forth, we could modify the *setObjectID* method in Figure 4 such that *c* is the value of the integer following the year (2012) in the suffix. If our *Session* sample EJBs had entirely numeric primary keys *k*, the mapping function *m* could have been conveniently defined as

$$m(k) = (k - (k \; modulo \; N), k \; modulo \; N).$$

Many EJBs have numeric primary keys, especially if the developer delegates the assignment of primary keys to the application server, in which case the server can use consecutive integers as EJB primary keys. This is very helpful in situations where the real-world entities that the EJBs represent have no "natural" unique primary key. An example would be a product or an order for a product. We chose a string primary key for our *Session* example to demonstrate that the fCEJB pattern does not rely on a numeric primary key.

C. Design Considerations for fCEJBs

By creating a simple façade session bean we can completely hide *CSessions* from the rest of the application and expose only *POJOs* to clients. With a façade session bean, the two-step process of first building an *idMapping* and then retrieving the desired *CSession* as shown in Figure 5 can be collapsed into one step. The façade bean is quite straightforward and obvious to program and therefore we do not show it here. For more complicated entities than our *Sessions*, consolidation through fCEJBs 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 fCEJBs at the request and under the directions of the developer. The tool would also need to support the following scenarios:

- If *Session* implements customized *ejbLoad*, *ejbStore*, *ejbActivate*, or *ejbPassivate* methods, these need to be adapted in *CSessionBean* to reflect the fact that the state of a *Session* is stored across different arrays in the *CSessionBean*.
- *Finder* and *select* queries for *Session* must be re-implemented for the fCEJB, and with less J2EE platform support, because they need to access both a *CSession* and the arrays within a *CSession*.
- If *Session* has customized *ejbHome* methods, we need to add functionally equivalent *ejbHome* methods to *CSession*. Changes to the original *Session ejbHome* methods are only necessary if these methods access the state of a specific *Session* EJB after a prior *select* method. In this case, the *CSession ejbHome* methods need to retrieve *POJO* instead of *Sessions*.
- If *Session* is part of a container-managed relationship (CMR), consolidation through fCEJBs requires removal of the CMRs and re-implementation of the CMRs without direct J2EE support.

The mapping function *m* has a strong impact on the performance of fCEJBs 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 primary 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. The primary key space of our sample *Session* entities fulfills the cluster property because our *Sessions* have largely lexicographically consecutive *sessionIDs* such as *voiceCall-05-12-2012a*, *voiceCall-05-12-2012b*, *voiceCall-05-12-2012c*, and so on.

```
1  public interface CSessionLocalHome extends EJBLocalHome {
2      CSessionLocal create(String objectID, int numElements) throws CreateException;
3      CSessionLocal findByPrimaryKey(String objectID) throws FinderException;
4      CSessionLocal getSession(String objectID, int numElements) throws FinderException;
5  }
```

Figure 1. Local home interface for *CSession*.

```
1  public interface CSessionLocal extends EJBLocalObject {
2      void createSession(int index, String sessionID, long startTime) throws DuplicateKeyException;
3      void removeSession(int index) throws RemoveException;
4      String getSessionID(int index);
5      void setSessionID(int index, String sessionID);
6      long getStartTime(int index);
7      void setStartTime(int index, long startTime);
8      void businessMethod(int index, long newStartTime);
9  }
```

Figure 2. Local interface for *CSession*.

```
1  public abstract class CSessionBean implements EntityBean {
2      private transient String[] sessionIDs = null;
3      private transient long[] startTimes = null;
4      public abstract String getObjectID();
5      public abstract void setObjectID(String objectID);
6      public abstract String getEncodedSessionIDs();
7      public abstract void setEncodedSessionIDs(String encodedSessionIDs);
8      public abstract String getEncodedStartTimes();
9      public abstract void setEncodedStartTimes(String encodedStartTimes);
10
11 public String ejbCreate(String objectID, int N) throws CreateException {
12     setObjectID(objectID);
13     sessionIDs = new String[N];
14     startTimes = new long[N];
15     return null;
16 }
17
18 public void ejbLoad() {
19     StringTokenizer encodedSessionIDs = new StringTokenizer(getEncodedSessionIDs(), "#"),
20                     encodedStartTimes = new StringTokenizer(getEncodedStartTimes(), "#");
21     int numElements = encodedSessionIDs.countTokens();
22     if (sessionIDs == null) {
23         sessionIDs = new String[numElements];
24         startTimes = new long[numElements];
25     }
26     for (int index = 0; index < numElements; index++) {
27         sessionIDs[index] = encodedSessionIDs.nextToken();
28         startTimes[index] = new Long(encodedStartTimes.nextToken()).longValue();
29     }
30 }
31
32 public void ejbStore() {
33     StringBuffer encodedValues = new StringBuffer();
34     for (int index = 0; index < sessionIDs.length; index++) {
35         encodedValues.append(sessionIDs[index]);
36         encodedValues.append("#");
37     }
38     setEncodedSessionIDs(encodedValues.toString());
39     encodedValues = new StringBuffer();
40     for (int index = 0; index < startTimes.length; index++) {
41         encodedValues.append(startTimes[index]);
42         encodedValues.append("#");
43     }
44     setEncodedStartTimes(encodedValues.toString());
45 }
46
47 public void createSession(int index, String sessionID, long startTime) throws DuplicateKeyException {
48     if (sessionIDs[index] != null) throw new DuplicateKeyException("Session exists already");
49     sessionIDs[index] = sessionID;
50     startTimes[index] = startTime;
51 }
52
53 public void removeSession(int index) throws RemoveException {
54     if (sessionIDs[index] == null) throw new RemoveException("Session does not exist");
55     sessionIDs[index] = null;
56 }
57
58 public void businessMethod(int index, long newStartTime) {
59     startTimes[index] = newStartTime;
60 }
61
62 public void setSessionID(int index, String sessionID) {
63     sessionIDs[index] = sessionID;
64 }
```

Figure 3. Portion of the *CSessionBean* relevant to the fCEJB discussion.

```
1  public class ObjectIDMapping {
2      private int N,
3              index;
4  private String objectID;
5
6  public ObjectIDMapping(int N) {
7      this.N = N;
8      index = -1;
9      objectID = null;
10 }
11
12 public void setObjectID(String objectIDArg) {
13     int lastElementIndex = objectIDArg.length() - 1,
14         c = objectIDArg.charAt(lastElementIndex);
15     index = c % N;
16     objectID = objectIDArg.substring(0, lastElementIndex) + "_" + (c - index);
17 }
18
19 public int getIndex() {
20     return index;
21 }
22
23 public String getObjectID() {
24     return objectID;
25 }
26 }
```

Figure 4. A class for mapping *Session* primary keys to *CSession* primary keys and array index slots.

```
1  ObjectIDMapping idMapping = new ObjectIDMapping(N);
2  idMapping.setObjectID("voiceCall-05-12-2012-a");
3  CSessionLocal cSession = csessionLocalHome.findByPrimaryKey(idMapping.getObjectID());
4  cSession.businessMethod(idMapping.getIndex(), System.currentTimeMillis());
```

Figure 5. Accessing a *CSession* EJB.

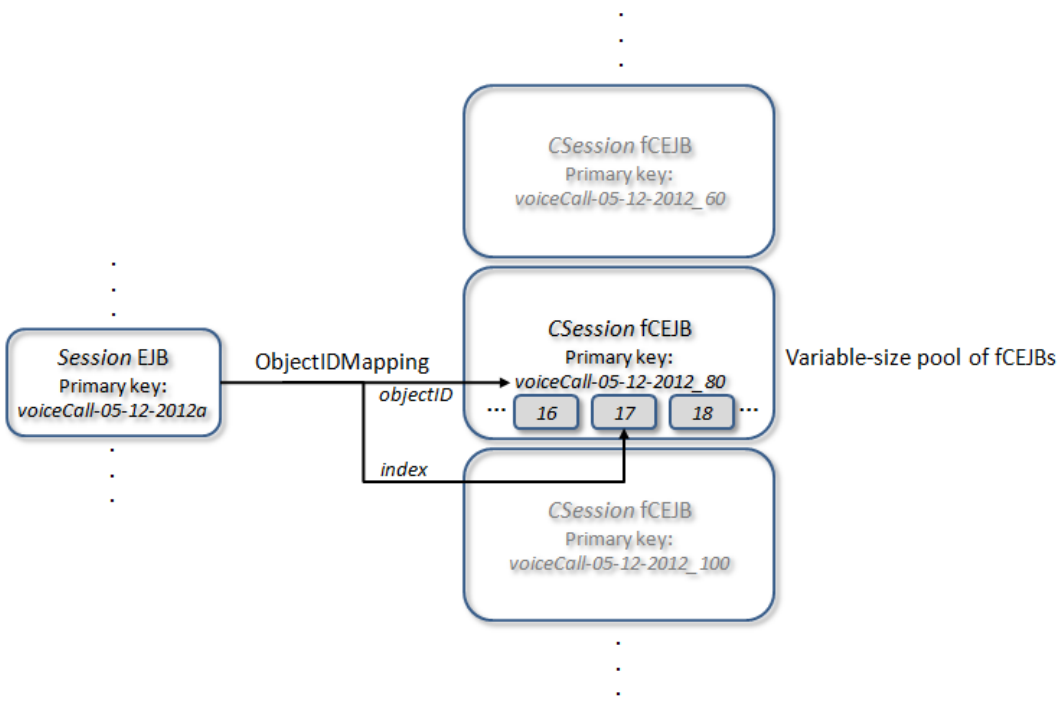


Figure 6. Mapping a *Session* primary key to a tuple (*objectID*, *index*): *objectID* is the primary key of a *CSession*, *index* is the slot in the *CSession* that stores the original *Session* entity.

V. VARIABLE-SIZE CEJBS

In this section, we describe the key idea behind vCEJBs, the design methodology, and practical aspects of developing vCEJBs.

A. Concept of the vCEJB Pattern

The fCEJBs pattern stores a fixed number of entities in each fCEJB, while the size of the pool of all fCEJBs varies with the total number of entities. In contrast, the vCEJB pattern creates a fixed-size pool of vCEJBs but each vCEJB stores a variable number of entities. Variable-size CEJBs constitute a distributed EJB equivalent of hashtables. A hashtable contains a fixed number of slots, each of which can hold a variable number of entities that are mapped to the slots based on a mapping (hash) function. A direct implementation of a hashtable as a single EJB could lead to a prohibitively slow performance for a large number of hashtable entries because

- the time for synchronizing the EJB state with the underlying database at the beginning and/or end of a transaction would be very long,
- the amount of parallelism in accessing the hashtable would be severely limited.

Therefore, we distribute the content of the hashtable across several EJBs, one EJB for each hashtable slot. The resulting EJBs are our vCEJBs. Unlike an fCEJB, a vCEJB imposes no predefined limit on the number of entities stored in the vCEJB.

The primary keys of the vCEJBs are integers ranging from 0 to *N* - 1 for a chosen value of *N* that we will discuss later. We define a mapping function from the entities' primary keys to the interval $[0..N-1]$ that determines which vCEJB stores which entity. The entities are represented as POJOs and are stored in a Java hashtable (a *java.util.HashMap*) in the vCEJBs. To store all entities of a given type in an application, *N v*CEJBs are allocated in a fixed-size pool at application startup time.

To demonstrate the fCEJB pattern, we chose the example of a *Session* entity because its primary key space has the desired cluster property that makes it amenable to the fCEJB pattern. In contrast, we will illustrate the vCEJB pattern with the example of a *User* EJB whose primary keys do not exhibit the cluster property. We assume that the primary key of our *User* entity is a unique *userID* such as a first name/middle name/last name combination, passport number, social security ID, employee number, telephone number, or similar. The uneven distribution of these identifiers makes it extremely difficult to define a mapping function *m* that would evenly map *User* entities to fCEJBs. As we will see, the performance of vCEJBs does not depend on the cluster property, and therefore vCEJBs are the preferable choice for *User* entities.

In the following explanations, we assume that *User* has two fields *firstName* and *lastName* in addition to the *userID*, Furthermore, *User* is implemented with the canonical getter/setter interfaces and local and local home interfaces. We omit additional implementation details because they are irrelevant to our vCEJB discussion.

B. Developing a vCEJB

We derive a vCEJB *CUser* from *User* in two steps. In the first step, we create a POJO equivalent of *User*, which we call *POJOUser* (omitted from the figures for the sake of brevity). *POJOUser* contains three private instance variables *userID*, *firstName*, and *lastName*, and the canonical getter and setter methods for the three variables. In the second step, we create *CUser* as an entity EJB as depicted in Figures 8-10. *CUser* has three CMP fields, *objectID* of type *Integer*, *N* of type *int*, and *users* of type *java.util.HashMap* (lines 2-7 in Figure 10). The methods in *CUserBean* pertinent to our discussion are *ejbCreate*, *createUser*, *getUser*, *setUser*, *changeUser*, and *removeUser*.

A *CUser* acts as a container for *POJOUsers* in a way that is similar to EJB containers managing EJBs. Unlike EJB containers, on the other hand, a *CUser* cannot hold objects of different classes. The lifecycle methods for a *POJOUser* (*createUser*, *removeUser*) can be found in the local interface for *CUser* (Figure 9), whereas the lifecycle methods for a *User* reside in the local *home* interface for the *User* EJB (Figure 8). EJB containers are automatically instantiated by the application server, whereas *CUsers* have to be created by the J2EE application. This also implies that the number of vCEJBs depends on the application rather than the application server.

To consolidate *Users* into *CUsers* in a given J2EE application, the application first creates a pool of *N CUsers* with *objectIDs* ranging from 0 to *N* - 1 in increments of 1. Subsequently, the application can create, find, modify, execute business methods on, and remove *POJOUsers* inside *CUsers*. To do so, the application first executes *findByPrimaryKey* on the *CUserLocalHome* interface (see Figure 8) with the argument

new Integer(Integer.abs(userID.hashCode()) % N),

where *userID* is the return value of the *getUserID* method for the *POJOUser* in question and *%* denotes an integer division. In other words, the application maps hash values of the *POJOUser* identities to *CUser* identities in an attempt to evenly distribute *POJOUsers* across *CUsers*. Notice that due to integer arithmetic and the definition of the *hashCode* method for Java strings, the result of the *hashCode* method can be negative and therefore we apply the *Integer.abs* method to guarantee values in the range $[0..N-1]$. The return value of the *findByPrimaryKey* method is the *CUser* vCEJB that already contains or will contain the *POJOUser* that we are interested in. Figure 7 illustrates the mapping of *User* primary keys to *CUser* primary keys.

To store a new *POJOUser* in the *CUser* vCEJB, the application executes the *createUser* method on the *CUser* as shown in lines 16-24 in Figure 10. First, this method ensures that the *POJOUser* indeed belongs in this *CUser* based on the mapping of *POJOUsers*' *userIDs* to *CUser* object identities, as described in the previous paragraph. Then, the method checks whether there is already a *POJOUser* with the same identity stored in this *CUser*. This is the equivalent of the EJB container checking for duplicate object identities

when creating an entity EJB. Finally, the method stores the mapping from the *userID* to the *POJOUser* in the vCEJB's internal *HashMap*.

The equivalent of an EJB *finder* method for *POJOUsers* is the *getUser* method in the *CUserLocal* interface (Figure 9, line 3). After a prior call to the *findByPrimaryKey* method on the *CUserLocalHome* interface to obtain the appropriate *CUser*, the application calls the *getUser* method on the local interface of that *CUser* to obtain the desired *POJOUser*. The application can now execute business methods on the returned *POJOUser*. The *users* field (a *HashMap*) in *CUser* is a so-called *dependent value* object in the J2EE world. By extension, the same applies to *POJOUsers* inside the users *HashMap*. Hence, the EJB container returns a *copy* of a *POJOUser* whenever the *getUser* method is invoked, as prescribed by the Enterprise JavaBeans specification. To reflect changes to the state of a *POJOUser* due to business method calls, the application has to store the *POJOUser* back to *users*. The *setUser* and *changeUser* methods in Figure 10 in lines 33-38 and 40-49, respectively, serve this purpose. The *changeUser* method is useful in situations where we want to change the state of a *POJOUser* without a need to know the previous state of this *POJOUser*. Rather than calling *getUser* followed by *setUser*, one call to *changeUser* will suffice in that situation, hence reducing the number of accesses to the users *HashMap* and consequently the number of *HashMap* copy operations from two to one.

To delete a *POJOUser*, the application calls the *removeUser* method on the *CUser* (lines 51-56 in Figure 10). Like the *setUser*, *getUser*, and *changeUser* methods, *removeUser* first checks that the referenced *POJOUser* indeed exists in this *CUser* vCEJB. Then, *removeUser* deletes the *POJOUser* from the *users HashMap*.

C. Design Considerations for vCEJBs

By creating a simple façade session bean we can completely hide *CUsers* from the rest of the application and expose only *POJOUsers* to clients. With a façade session bean, the two-step process of first retrieving a *CUser* and subsequently accessing a *POJOUser* turns into one step for clients. The façade bean is straightforward and we will therefore not show it here.

Our sample *User* EJB is very simple. For more complicated entities, consolidation through vCEJBs requires more effort but, as with fCEJBs, is straightforward and could be automated by a tool as part of a J2EE development environment. The following is a list of considerations during vCEJB creation in the context of the *User* EJB that Section V.B did not address.

1. If the original *User* EJB implements the *ejbLoad*, *ejbStore*, *ejbActivate*, or *ejbPassivate* methods, the *CUser* methods *getUser*, *setUser*, and *changeUser* need to be modified. For example, the content of a *User ejbLoad* method needs to be moved into the *getUser* and *changeUser* methods after some modifications. The modifications reflect the fact that the state of a *User* is stored in a *POJOUser* and needs to be retrieved from a *HashMap* rather than from the CMP fields of a *User*.
2. *Finder* and *select* queries for *User* must be re-implemented for the vCEJB because they need to access the *users HashMap*. Notice that the *getUser* method in our example is derived from the *findByPrimaryKey* method for the *User* EJB. More complicated *finder* methods in *User* would require more complicated *getUser* methods in *CUser*.
3. If *User* has *ejbHome* methods, we need to add functionally equivalent *ejbHome* methods to *CUser*. Changes to the original *User ejbHome* methods will only be 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 *POJOUsers* instead of *Users.*
4. If *User* is part of a container-managed relationship (CMR), consolidation through vCEJBs requires removal of the CMRs and re-implementation of the CMRs without direct J2EE support.
5. Variable-size CEJBs aggravate the existing problem of variable-size data structures in EJBs. EJBs with variable-size data structures as CMP fields and databases as persistent storage require a design-time decision for the maximum length of each database column that stores a variable-size CMP field. If such a maximum size is exceeded a runtime error will occur during EJB storage in the database. *CUser* contains a variable-size CMP field (*users*) even though *User* does not. To safely use vCEJBs, we require knowledge of the maximum number of EJBs that are stored in each CEJB and have to appropriately size the database column that stores the *users HashMap.*

D. Configuring the vCEJB Pool Size N

By consolidating a large number of EJBs into a small number of vCEJBs, the number of rows in a relational database required to store entity EJB state can be substantially reduced. At the same time, the degree of locality in EJB operations increases, which has a positive effect on the efficacy of the EJB cache in a J2EE application server. In our example, we can reduce the number of database rows and EJB cache entries for storing n user entities from n to N.

With *CUsers*, the time for retrieving a user entity is divided into time for searching cached or uncached *CUsers* and time for an in-memory search within a *HashMap*. In the extreme case of $N = 1$, there is only one *CUser* and it is likely to be present in the EJB cache of the application server. In this case, the vCEJB is essentially a persistent and transacted *HashMap*. Even if the *CUser* is not cached, it can be located very quickly in the database. Most of the search time is spent in memory within the *HashMap* of the *CUser*. However, this *HashMap* grows potentially very large (to n entries) and can itself turn into a performance bottleneck. Moreover, the time for synchronizing the in-memory representation of *CUser* with the database at transaction commit time could be very long. The same applies to loading the *CUser* from the database at the beginning of a transaction with J2EE commit options B and C [2]. Note that for a

clustered J2EE application server commit options B and C are mandatory. Lastly, like fCEJBs, vCEJBs can restrict the degree of parallelism in the J2EE application. If two transactions attempt to access two POJOs that happen to be in the same vCEJB, one of the transactions may lock out the other transaction. The likelihood of this situation increases with decreasing values for *N*.

The other extreme is $N = n_{max}$, where n_{max} denotes the maximum number of concurrently existing user entities throughout the lifetime of the application, provided such a maximum exists. With $N = n_{max}$, we arrive at the same situation as with EJBs except that with vCEJBs every access to an embedded POJO requires an additional step relative to EJBs. In other words, with $N = n_{max}$ we can expect a performance *penalty* relative to using EJBs.

The ideal value for *N* therefore lies between these extremes. Clearly, this value depends on the size and structure of the EJB cache in the J2EE application server, the implementation of the EJB container, the database specifics, and the hardware on which the application server and the database run. Since we typically have no insight into the inner workings of a J2EE application server or the database, there is no general way of determining the best choice for *N*. In addition, the value of n_{max} may not be known and n_{max} may not even exist, which complicates the configuration of *N*.

One of our future research directions is therefore a self-adjusting vCEJB technique, where a session façade bean for vCEJBs would create vCEJBs dynamically as needed. The session façade bean would monitor the vCEJB performance and dynamically shrink or enlarge the size of the vCEJB pool accordingly, similar to automatic hashtable resizing techniques. After creation or destruction of vCEJBs, the façade bean would reallocate the existing POJOs across the modified set of vCEJBs. By appropriately sizing the vCEJB pool, the façade bean would also ensure that the size of the *HashMap* in each vCEJB does not exceed the limit imposed by the maximum size of the corresponding database column (see bullet item 5 in Section V.C). We believe that such a self-adjusting vCEJB technique may be beneficial in applications with slowly changing sets of real-world entities where dynamic reallocations would take place rarely and thus the performance cost of the reallocation itself would be limited.

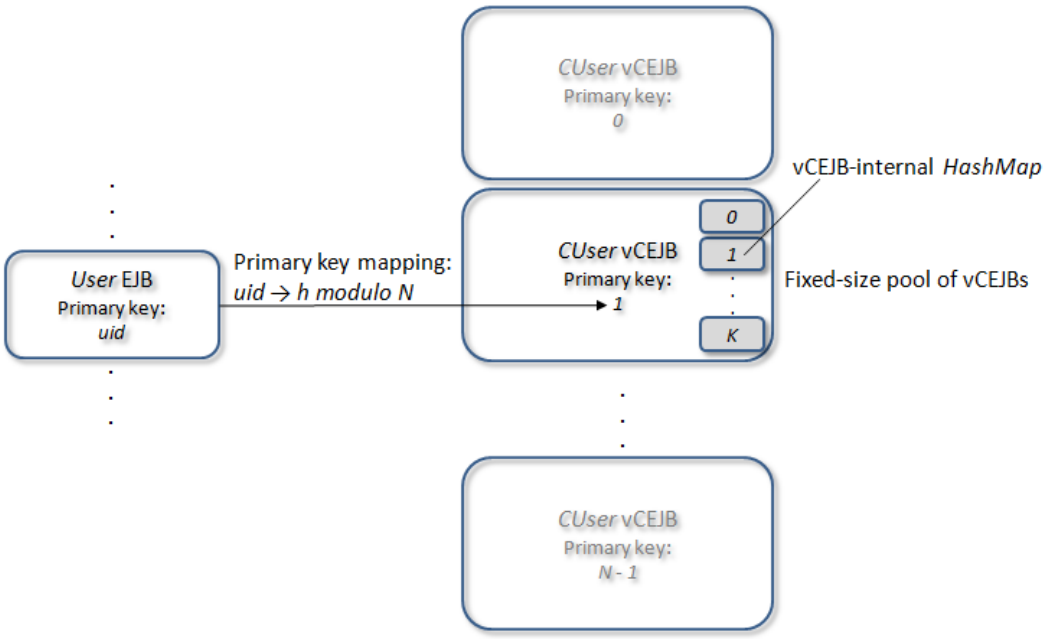


Figure 7. Mapping of a *User* primary key *uid* to a *CUser* primary key, where *h* is the absolute value of the hashcode for *uid*.

```
1  public interface CUserLocalHome extends EJBLocalHome {
2  CUserLocal create(Integer objectID, int N) throws CreateException;
3  CUserLocal findByPrimaryKey(Integer objectID) throws FinderException;
4  }
```

Figure 8. Local home interface for the vCEJB *CUser*.

```
public interface CUserLocal extends EJBLocalObject {
void createUser(POJOUser user) throws DuplicateKeyException, CreateException;
POJOUser getUser(String userID) throws FinderException;
void setUser(POJOUser user) throws FinderException;
void changeUser(String userID, String firstName, String lastName) throws FinderException;
void removeUser(String userID) throws FinderException;
}
```

Figure 9. Local interface for the vCEJB *CUser*.

```
public abstract class CUserBean implements EntityBean {
    public abstract Integer getObjectID();
    public abstract void setObjectID(Integer objectID);
    public abstract int getN();
    public abstract void setN(int N);
    public abstract HashMap getUsers();
    public abstract void setUsers(HashMap users);

public Integer ejbCreate(Integer objectID, int N) throws CreateException {
    setN(N);
    setObjectID(objectID);
    setUsers(new HashMap());
    return null;
}

public void createUser(POJOUser user) throws DuplicateKeyException, CreateException {
    HashMap allUsers = getUsers();
    int p = getN();
    if (Integer.abs(user.getUserID().hashCode()) % p != getObjectID().intValue())
        throw new CreateException("Cannot store user in this CEJB.");
    if (allUsers.get(user.getUserID()) != null) throw new DuplicateKeyException();
    allUsers.put(user.getUserID(), user);
    setUsers(allUsers);
}

public POJOUser getUser(String userID) throws FinderException {
    HashMap allUsers = getUsers();
    POJOUser user = (POJOUser) allUsers.get(userID);
    if (user == null) throw new FinderException();
    return user;
}

public void setUser(POJOUser user) throws FinderException {
    HashMap allUsers = getUsers();
    if (allUsers.get(user.getUserID()) == null) throw new FinderException();
    allUsers.put(user.getUserID(), user);
}

public void changeUser(String userID, String firstName, String lastName)
    throws FinderException {
    HashMap allUsers = getUsers();
    POJOUser pUser = (POJOUser) allUsers.get(userID);
    if (pUser == null) throw new FinderException();
    pUser.setFirstName(firstName);
    pUser.setLastName(lastName);
    allUsers.put(userID, pUser);
}

public void removeUser(String userID) throws FinderException {
    HashMap allUsers = getUsers();
    if (allUsers.get(userID) == null) throw new FinderException();
    allUsers.remove(userID);
}
```

Figure 10. Portion of the *CUserBean* implementation relevant to the vCEJB discussion.

VI. PERFORMANCE EVALUATION

This section contains an assessment of the comparative performance of fCEJBs, vCEJBs, and traditional EJBs in a test environment that simulates different usage profiles.

A. Methodology

We compared the performance of "traditional" EJBs with one real-world entity per EJB, fCEJBs, and vCEJBs in a J2EE test application. The entities that the test application creates have lexicographically consecutive strings as primary keys (as shown in our *Session* example in Section IV). For fCEJBs, the application uses the mapping function *m* in Figure 4. The test application executes a sequence of operations either on traditional EJBs (*EJB mode*), fCEJBs (*fCEJB mode*), or *vCEJBs* (*vCEJB mode*). In EJB mode, the application executes the following sequence of steps:

1. Create *n* EJBs. We call each entity creation a *creation operation*.
2. Find EJB with randomly selected primary key and read its state through *getter* operations. Repeat *n* times. We call each such operation a *find and read operation*.
3. Find EJB with randomly selected primary key and execute a business method on it. The business method changes the state of the EJB, and thus requires synchronization with the underlying database at transaction commit time. Repeat *n* times. We call each such operation a *find and change operation*.
4. Delete all 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 EJB 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 fCEJB mode, the application performs the same steps on fCEJBs instead of EJBs. Also, in step 4 in fCEJB mode, the application sequentially deletes all entities in each fCEJB but not the fCEJB itself. We varied the maximum number *N* of entities per fCEJB, from 2 to 250 in consecutive runs of the test application. The performance of the test application peaked around $N = 20$. We therefore present only the performance results for $N = 20$.

In vCEJB mode, the application first creates *N* vCEJBs, followed by the same steps as the test application in fCEJB mode but with vCEJBs instead. We varied *N* in consecutive runs of the test application in vCEJB mode and determined that the performance of the test application peaked roughly at $N \approx n/10$, i. e., when approximately 10 entities are stored in each variable-size vCEJB on average. We will only present the performance results for $N = n/10$.

We configured the test application with two different transaction settings in two different experiments: in *long transaction mode*, each of the four steps 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 removal. 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 EJB 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 11-16 display the results of our performance testing with the test application in long and short transaction modes for the three different values of *n*. Each figure shows the time that each entity creation, entity find/read, entity find/change, and entity removal operation takes in milliseconds when using traditional EJBs, fCEJBs, and vCEJBs, respectively. In each figure, for each of the four types of entity operations, there is one bar indicating the speed of the operation when using EJBs, fCEJBs, and vCEJBs, respectively. In addition, we show the speedup for the operation when using fCEJBs instead of EJBs and the speedup when using vCEJBs instead of EJBs. The speedup in the figures is defined as the time for an EJB operation divided by the time for the equivalent f/vCEJB operation. Speedup values greater than 1 indicate results where f/vCEJBs outperform EJBs, values of less than 1 indicate EJBs performing better than f/vCEJBs. For the vCEJB performance tests, our figures do not show the time for creating the *N* vCEJBs because we consider this fixed overhead at application startup time.

In long transaction mode, fCEJBs significantly outperformed EJBs. For $n = 50000$ (Figure 13), for example, creating entities through fCEJBs was more than twice as fast as with EJBs, finding and reading entities was more than 5 times faster, finding and changing entities was more than 7 times faster, and deleting entities with CEJBs was more than 14 times faster. Our performance tests also show that fCEJBs are consistently faster than vCEJBs.

Because in fCEJB mode the mapping function *m* in our test application clusters the primary keys of the entities, the fCEJBs consolidate almost the maximum possible number of entities (20 per our definition of *N*). Hence, the number of fCEJBs necessary to store all entities in the test application is about $1/20^{th}$ that of the number of EJBs in EJB mode, which translates into much improved application server caching behavior and accelerated database search times. Once an fCEJB has been retrieved, extracting the desired entity from the fCEJB is a simple and fast array indexing operation. It is only insignificantly slower than retrieving the state of a traditional EJB from the EJB fields and faster than retrieving an entity from the internal *HashMap* in a vCEJB. Writing the state of an fCEJB back to the underlying database is much faster than the analogous operation for a vCEJB with its large internal data structure, which explains why fCEJBs perform reading and changing operations much faster than

vCEJBs. It also explains why the latter are only about 23% faster than EJBs for this type of operation (see find and read operations in Figure 13). However, if the chosen mapping function *m* for fCEJBs in a given application does not yield the desirable cluster property, vCEJBs may outperform fCEJBs, which is why we developed the vCEJB pattern as an alternative to the fCEJB pattern.

Although fCEJBs perform better than vCEJBs in our tests due to the distribution of the primary keys and the selection of the fCEJB mapping function *m*, there is still a significant speed-up when using vCEJBs as opposed to EJBs, with the exception of creation operations. For n = 50000 (Figure 13), finding and reading entities is more than twice as fast in vCEJB mode than in EJB mode, finding and changing entities is about 23% faster, and removal is more than three times faster in vCEJB mode than in EJB mode. Variable-size CEJBs are only at a performance disadvantage over EJBs in the case of creating entities. Here, retrieving an entry in the potentially large CEJB-internal *HashMap* for the purpose of checking for *DuplicateKeyExceptions*, the subsequent storage of a new entity in this *HashMap*, and the occasional re-sizing of the *HashMap* costs more time than the consolidation of entities saves.

Unlike in EJB mode, entity deletion in either CEJB mode 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 entities in both CEJBs modes is much faster than in EJB mode.

In short transaction mode, our performance testing shows a very different outcome (Figures 14-16). For example, Figure 16 (n = 50000) shows that both types of CEJBs only offer performance advantages over EJBs for finding and reading operations. Fixed-size CEJBs are about as fast as EJBs for finding and changing operations and for entity removal but much slower in creating entities. Variable-size CEJBs are consistently slower than EJBs except for finding and reading entities. 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. J2EE applications that eagerly commit every EJB state change will still experience a significant speed-up as a result of consolidation but only if EJB read operations outnumber EJB write operations by a significant margin.

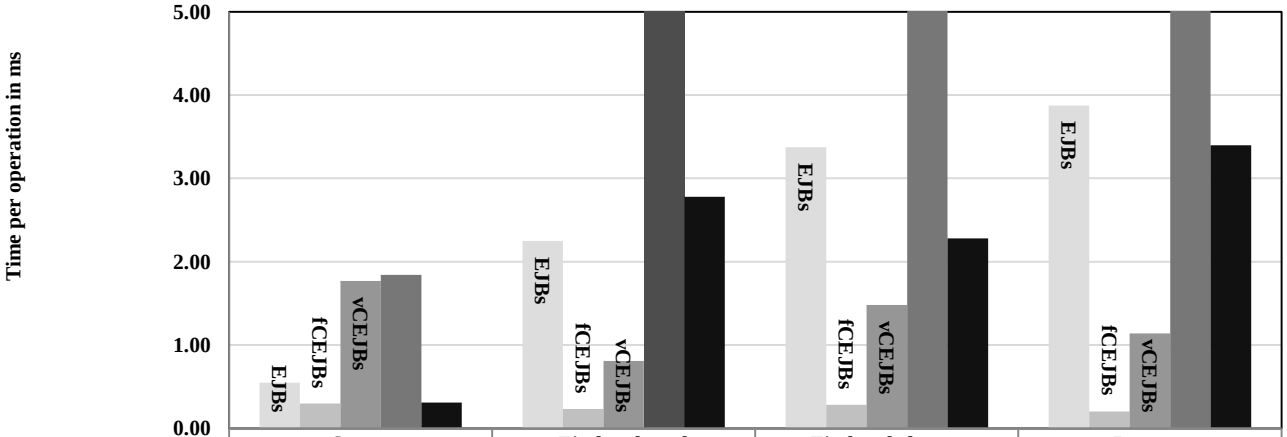
In conclusion, fCEJBs provide strong performance advantages over EJBs if (1) the application contains a large number of EJBs, (2) it accesses EJBs either in long-lived transactions or in short-lived transaction with a large EJB read to write ratio, and (3) if a mapping function *m* can be found for the EJB primary key space that exhibits the cluster property. If no such function can be found but (1) and (2) are true, vCEJBs can be used to considerably increase application performance.

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 the number of EJB accesses, whereas other J2EE applications may have irregular EJB accesses and also contain computationally or I/O-intensive tasks. Our *Session* and *User* EJBs are simple while EJBs in common J2EE applications can be more complex and may also be linked to each other. However, we believe that our test application realistically captures the performance *differences* between EJBs and f/vCEJBs in a large class of J2EE applications that are characterized by high 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.

VII. Conclusion and Future Work

We presented two J2EE software design patterns that consolidate multiple entities in J2EE applications into special-purpose entity EJBs that we call consolidated EJBs (CEJBs). Our first design pattern maps entities to fixed-size CEJBs (fCEJBs), whereas our second pattern constructs variable-size CEJBs (vCEJBs). Consolidation increases the locality of data access in J2EE applications, thus making EJB 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 especially fCEJBs can outperform traditional EJBs by a wide margin for common EJB operations. For example, the fCEJB 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 fCEJBs. In applications that do not lend themselves to the fCEJB design pattern, the second design pattern, vCEJBs, can enhance the application performance, albeit by smaller factors. In our experiments, we measured a speed-up of entity finder and access operations by a factor of more than two for vCEJBs versus traditional EJBs. Both types of CEJBs conform to the EJB specification and can therefore be used in any J2EE application on any J2EE application server.

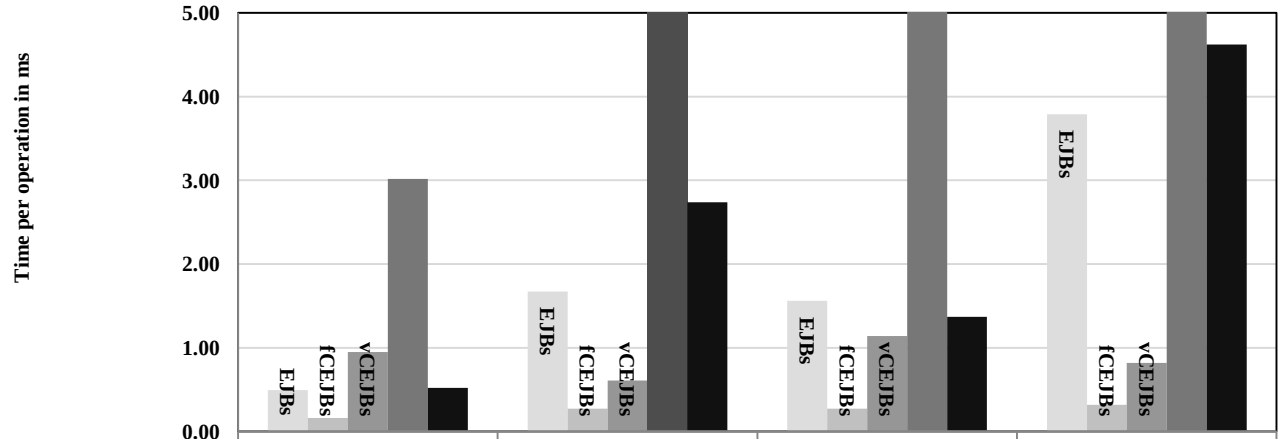
We have several future research goals for CEJBs. First, we would like to modify CEJBs in such a way that applications with short-lived transactions and a small ratio of EJB read to EJB write operations perform better than our current patterns. Secondly, we intend to investigate mapping functions for fCEJBs that (1) perform well if the primary key space for EJBs is irregular or unpredictable (such as user names, phone numbers, or national IDs), and (2) that can be automatically defined without requiring complex developer decisions. Thirdly, we would like to address a currently open question for our f/vCEJB design patterns: how can we modify the f/vCEJB patterns so that they are beneficial in *most* J2EE applications and thus could ultimately become a standard way of implementing entities in J2EE applications? Lastly, a tool that would assist the developer in converting traditional EJBs into CEJBs would be highly desirable.



	Create	Find and read	Find and change	Remove
EJBs	0.55	2.25	3.38	3.88
fCEJBs	0.30	0.23	0.28	0.20
vCEJBs	1.77	0.81	1.48	1.14
Speedup fCEJBs->EJBs	1.84	9.62	12.01	19.00
Speedup vCEJBs->EJBs	0.31	2.78	2.28	3.40

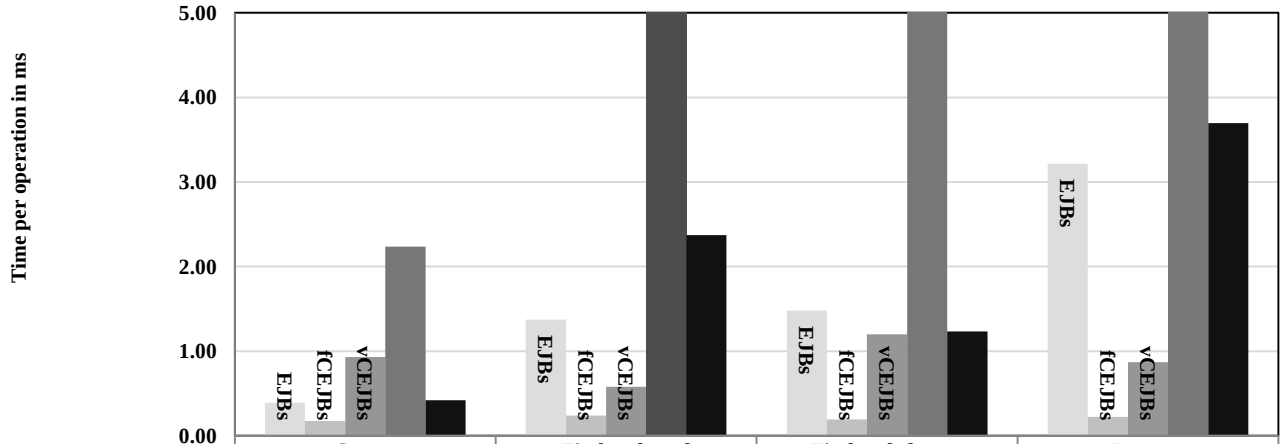
Figure 11. Test application performance in long transaction mode, n = 1000.

.



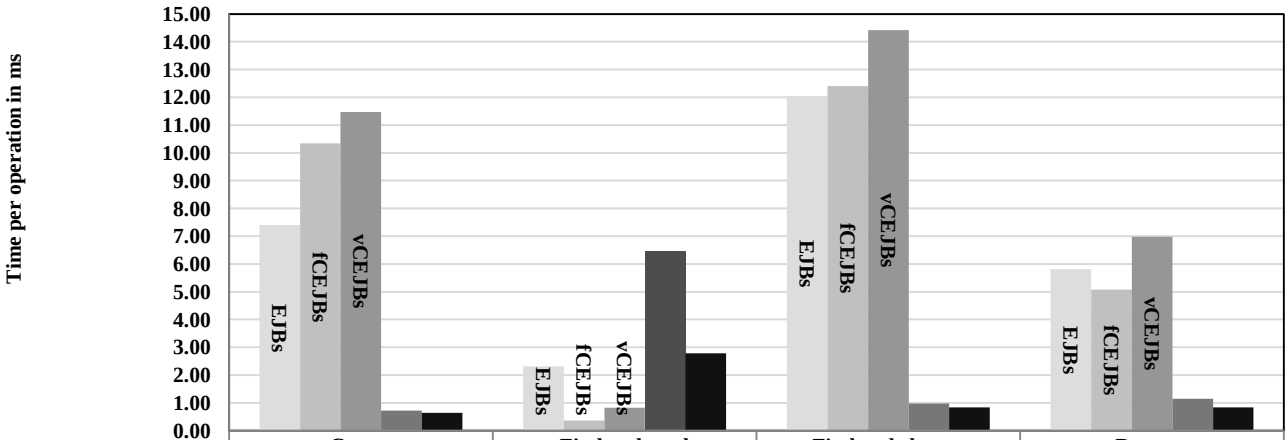
	Create	Find and read	Find and change	Remove
EJBs	0.50	1.67	1.56	3.79
fCEJBs	0.16	0.27	0.27	0.32
vCEJBs	0.95	0.61	1.14	0.82
Speedup fCEJBs->EJBs	3.02	6.14	5.72	11.84
Speedup vCEJBs->EJBs	0.52	2.74	1.37	4.62

Figure 12. Test application performance in long transaction mode, n = 10000.



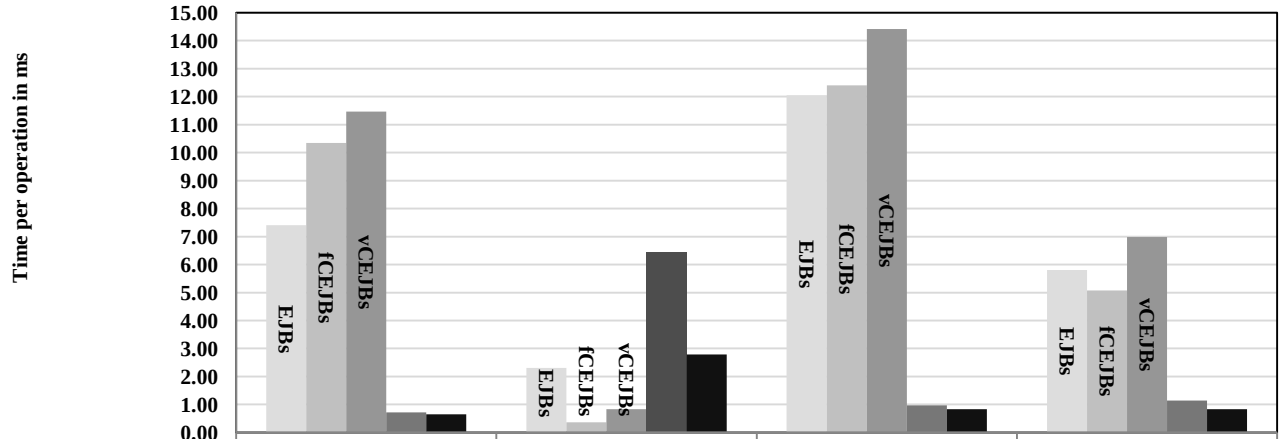
	Create	Find and read	Find and change	Remove
EJBs	0.39	1.38	1.48	3.21
fCEJBs	0.17	0.24	0.20	0.23
vCEJBs	0.93	0.58	1.20	0.87
Speedup fCEJBs->EJBs	2.24	5.75	7.58	14.28
Speedup vCEJBs->EJBs	0.42	2.37	1.23	3.69

Figure 13. Test application performance in long transaction mode, n = 50000.



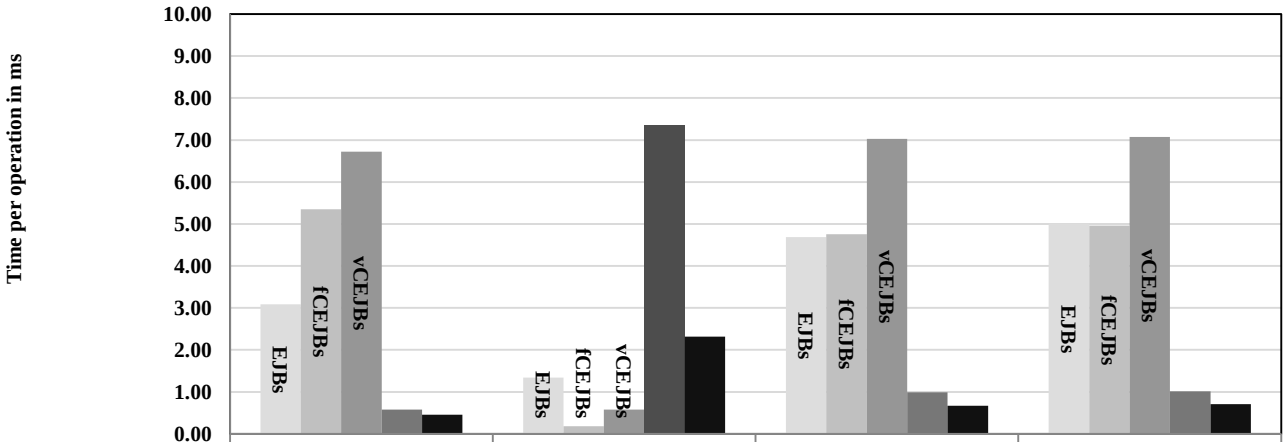
	Create	Find and read	Find and change	Remove
EJBs	7.41	2.31	12.05	5.81
fCEJBs	10.34	0.36	12.41	5.08
vCEJBs	11.47	0.83	14.42	6.98
Speedup fCEJBs->EJBs	0.72	6.44	0.97	1.14
Speedup vCEJBs->EJBs	0.65	2.79	0.84	0.83

Figure 14. Test application performance in short transaction mode, n = 1000.



	Create	Find and read	Find and change	Remove
EJBs	7.41	2.31	12.05	5.81
fCEJBs	10.34	0.36	12.41	5.08
vCEJBs	11.47	0.83	14.42	6.98
Speedup fCEJBs->EJBs	0.72	6.44	0.97	1.14
Speedup vCEJBs->EJBs	0.65	2.79	0.84	0.83

Figure 15. Test application performance in short transaction mode, n = 10000.



	Create	Find and read	Find and change	Remove
EJBs	3.08	1.34	4.69	5.01
fCEJBs	5.35	0.18	4.75	4.96
vCEJBs	6.72	0.58	7.03	7.07
Speedup fCEJBs->EJBs	0.58	7.34	0.99	1.01
Speedup vCEJBs->EJBs	0.46	2.32	0.67	0.71

Figure 16. Test application performance in short transaction mode, n = 50000.

VIII. ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers of this article for the numerous and detailed suggestions for improvements to the manuscript. The clarity and quality of the article has greatly benefited from the reviewers' suggestions.

REFERENCES

[1] R. Klemm, "The Consolidated Enterprise Java Beans Design Pattern for Accelerating Large-Data J2EE Applications", The Seventh International Conference on Software Engineering Advances (ICSEA), Nov. 2012, retrieved February 1, 2013, from http://bit.ly/VgMuXs.

[2] Oracle Inc., "Enterprise JavaBeans Specification 2.1," retrieved September 28, 2012, from http://bit.ly/Ovip59.

[3] Oracle Inc., "J2EE v1.4 Documentation", retrieved February 1, 2013, from http://bit.ly/Ys0j2B.

[4] 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.

[5] 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.

[6] Java Community Process, "JSR 316: Java Platform, Enterprise Edition 6 (Java EE 6) Specification", retrieved February 1, 2013, from http://bit.ly/YsbKYb.

[7] Java Community Process, "JSR 317: Java Persistence 2.0", retrieved February 1, 2013, from http://bit.ly/XCa6WN.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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 May 25, 2013, from http://bit.ly/18fl5w9.

[16] 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.

[17] C. Larman, "The Aggregate Entity Bean Pattern," Software Development Magazine, Apr. 2000, retrieved September 28, 2012, from http://bit.ly/PgBoxe.

[18] M. Jordan, "A Comparative Study of Persistence Mechanisms for the Java Platform," Sun Microsystems Technical Report TR-2004-136, Sep. 2004, retrieved May 25, 2013, from http://bit.ly/ZkxPhT.

[19] P. Van Zyl,, D. G. Kourie, and A. Boake, "Comparing the Performance of Object Databases and ORM Tools," Proceedings of the 2006 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT '06), 2006, retrieved May 25, 2013, from http://bit.ly/1135N9a.

[20] 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.