Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services

Samuel Kounev Alejandro Buchmann

Department of Computer Science
Darmstadt University of Technology
{*skounev,buchmann*}*@informatik.tu-darmstadt.de*

Abstract

The J2EE platform provides a variety of options for making business data persistent using DBMS technology. However, the integration with existing backend database systems has proven to be of crucial importance for the scalability and performance of J2EE applications, because modern e-business systems are extremely data-intensive. As a result, the data access layer, and the link between the application server and the database server in particular, are very susceptible to turning into a system bottleneck. In this paper we use the ECperf benchmark as an example of a realistic application in order to illustrate the problems mentioned above and discuss how they could be approached and eliminated. In particular, we show how asynchronous, message-based processing could be exploited to reduce the load on the DBMS and improve system performance, scalability and reliability. Furthermore, we discuss the major issues related to the correct use of *entity beans* (the components provided by J2EE for modelling persistent data) and present a number of methods to optimize their performance utilizing caching mechanisms. We have evaluated the proposed techniques through measurements and have documented the performance gains that they provide.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

1 Introduction

Over the past couple of years, the Java 2 Enterprise Edition Platform (J2EE) has established itself as major technology for developing modern e-business solutions. This success is largely due to the fact that J2EE is not a proprietary product, but rather an industry standard, developed as the result of a large industry initiative led by Sun Microsystems, Inc. The goal of this initiative was to establish a standard middleware framework for developing enterprise-class distributed applications in Java. Over 30 software vendors have participated in this effort and have come up with their own implementations of J2EE, the latter being commonly referred to as *J2EE Application Servers*.

In essence, the aim of J2EE is to enable developers to quickly and easily build scalable, reliable and secure applications without having to develop their own complex *middleware services*. Here we are talking about services such as transaction management, caching, resource-pooling, clustering, transparent failover, load-balancing and back-end integration to name just a few. Many of these services are also provided by traditional Transaction Processing Monitors (TPMs), but in a rather monolithic and highly proprietary manner. The J2EE platform allows developers to leverage middleware services provided by the industry without having to code using proprietary middleware APIs. Developers can concentrate on the business and application logic and rely on the J2EE Application Server to provide the infrastructure needed for scalability and performance.

However, the practice has proven that developing highly-performant and scalable J2EE applications is all but an easy undertaking. One of the most problematic issues to be addressed with respect to this, is the way business data is made persistent. The J2EE platform provides a variety of techniques for persistence, most of them utilizing existing DBMS technol-

ogy. However, unless these techniques are used correctly, persistence can easily turn into a system bottleneck. This is because, as already mentioned, modern e-business applications tend to be extremely data-intensive and therefore need highly-performant and scalable data management services. The integration with existing backend database servers is crucial with this respect and it is exactly in this area that processing inefficiencies and system bottlenecks are usually discovered.

In this paper we look at the data access issues and challenges in the development of J2EE applications. We discuss these issues in the context of ECperf - a newly released benchmark for measuring performance and scalability of application servers. Our goal is to use ECperf as an example of a realistic application in order to identify and discuss the areas in the management of persistent data that have the greatest impact on performance and that are susceptible to turning into system bottlenecks.

We propose different techniques to improve performance and eliminate data access bottlenecks. In particular, we discuss the major issues related to the correct use of *entity beans* as persistent data components and present a number of methods to optimize their performance utilizing caching mechanisms. In doing this, we consider both issues related to the application design, as well as issues related to the configuration of the deployment environment. Furthermore, we show how asynchronous, message-based processing could be exploited to reduce the load on the DBMS, leading to higher performance, reliability and scalability. We have evaluated the techniques that we propose through measurements and have documented the performance gains they provide.

2 The J2EE Platform and Persistence

J2EE applications are made up of components called *Enterprise JavaBeans (EJBs)*. The EJB specification [12], which is part of J2EE, defines the component model for developing *EJB components*. An EJB is a server-side software component containing some application and business logic. However, unlike other components, before EJBs can be used they need to be deployed in an *EJB Container*, which is normally provided by the J2EE application server in use. The EJB container has complete control over the lifecycle of its deployed EJBs and provides a number of middleware services that they can take advantage of. Typical services provided are transaction management, security, persistence, resource-pooling, transparent failover, clustering, load-balancing back-end integration and others. The only thing that is required of the application developer is to create the EJB components that define the business logic of the application and then specify the middleware services to be provided to these components by the container. The services required are declared in so-called *EJB Deployment Descriptors*, which the application developer has to provide together with the EJBs.

There are two types of EJB components - **session beans** and **entity beans**. Both are implemented as normal Java classes that are required to implement some predefined interfaces. Session beans are used to model business processes/services, while entity beans are used to model business data. For example, in a banking application, a session bean could be used to implement the process of transferring money from one account to another, while entity beans could be used to represent the bank account data. The services a session bean provides are implemented as methods of the respective session bean class and clients can invoke these methods in order to use the services provided. The container can instantiate multiple instances of the beans and in this way service multiple clients simultaneously.

Entity beans are the natural method provided by J2EE for modelling persistent business data. They provide an object-oriented model in which business data is represented as Java objects. The actual data being modelled is stored in attributes of the objects. However, in order to make the entity bean data persistent the container needs an underlying persistence mechanism. Examples of persistence mechanisms that can be used are relational databases, object databases or file systems. In the rest of this paper we will be assuming that a relational DBMS is used as persistence mechanism, since this is the most typical case.

Before the entity bean data can be made persistent, it must be mapped to some data structures in the underlying storage - the database. Data access code (typically SQL) must be provided for storing data to the database and retrieving it. The EJB specification [12] offers two alternatives for defining the data access code of entity beans. In the first case, code is written by the component developer and the bean is said to use *Bean-Managed Persistence (BMP)*. In the second case, code is automatically generated by the container and the bean is said to use *Container-Managed Persistence (CMP)*. As discussed in [10] both BMP and CMP have their virtues and drawbacks. We will later discuss in detail the issues that drive the choice between BMP and CMP and study the way they compare in terms of performance.

There are some fundamental benefits that entity beans bring to the table. First, they allow a clear separation between the business logic and the persistence logic of the application. When using entity beans the developer doesn't need to know about the underlying persistence mechanism. The data access logic is decoupled from the application logic and application code is much easier to understand and maintain. Second, as will be seen later, entity beans allow the container to cache and reuse data in the middle tier and in this way

reduce the load on the database. Finally, entity beans can enforce control on the way data is accessed and modified. For example, when updating an attribute on an entity bean, the entity bean may need to perform validation logic on its changes and possibly institute updates on other entity beans in the application.

However, there are situations when it is not worth to go through the entity bean layer. In particular, when reading large amounts of read-only data for listing purposes it is recommended to consider bypassing entity beans and read data directly through JDBC in session beans. This is sometimes termed *Session Bean-Managed Persistence (SMP)* and in the above situations may lead to a significant performance speedup. For lack of space, we do not go into further detail on SMP but concentrate on BMP and CMP, since they are more typical for modern J2EE applications. For more information on SMP we refer the reader to the "JDBC for Reading" pattern in [7].

To summarize, the J2EE platform provides 3 major approaches for managing persistent business data: BMP, CMP and SMP. In the next section we take a look at some general techniques that could be exploited to optimize entity bean performance.

3 Improving Entity Bean Performance

Ever since their introduction the use of entity beans has been the subject of heated discussions in the Enterprise Java Community. Practice has shown that for many e-business applications entity beans can achieve a reasonable performance level at a very low cost in terms of development time and effort. However, if not configured and optimized properly, entity beans can lead to serious performance degradation. In this section we will discuss the most important performance issues regarding the use of entity beans and their tuning and optimization. Before we start lets take a quick look at the lifecycle of an entity bean.

3.1 Entity Bean Lifecycle

According to the EJB specification [12] entity beans can be in one of three possible states: "Does Not Exist", "Pooled" or "Ready". These states together with the methods that are invoked upon state transitions are illustrated in Figure 1.

An entity bean instance's life starts when the container creates an instance of the entity bean's class by calling *newInstance*. The instance enters a pool of available instances, which are normal in-memory Java objects. While an instance is in the pool, it is not associated with any particular entity data. In this sense, all instances in the pool are considered equivalent and the container may use them to execute the entity bean's finder methods - *ejbFinds*. Finder methods are used to locate particular entity bean data in the underlying datastore and load the data in an entity bean object

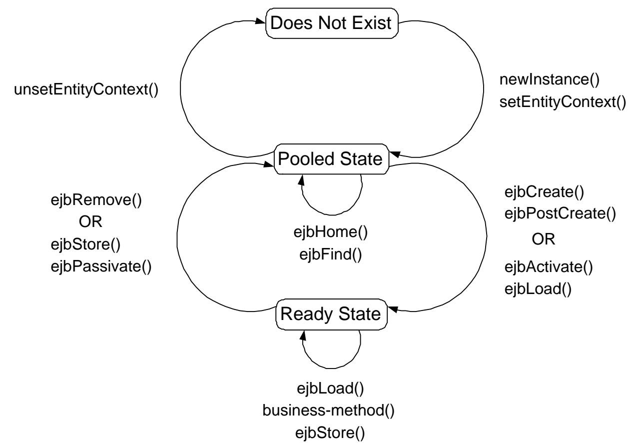

Figure 1: Lifecycle of an Entity Bean instance.

from the pool. The container loads the data by calling the ejbLoad method on the selected instance. This transitions the bean instance from the Pooled state to the Ready state. The instance is now associated with particular entity bean data and business methods can be invoked on it to manipulate (e.g. update) this data. When the client has finished working with the data, the container calls the *ejbStore* method of the instance to update the entity bean data in the underlying persistent storage. Hereafter we will use the term *entity bean* to refer both to the in-memory object instance and to the underlying persistent data that it represents.

The purpose of the *ejbLoad* and *ejbStore* methods is to synchronize the state of the instance with the state of the entity in the underlying persistent storage. Typically, the container calls ejbLoad at transaction begin in order to load the entity's data. At transaction commit the container calls ejbStore to write back updated data to persistent storage.

3.2 Configuring the Container's Caching Behavior

Having control over the times when entity beans transition from one state to another and the times when ejbLoad and ejbStore methods are called, allows containers to cache both entity bean object instances (with and without identity) as well as entity bean data. For example, once data is loaded from the underlying database into an entity object, the container can use the object to service multiple client requests, without further accessing the database. This reduces the load on the DBMS and is one of the greatest benefits that entity beans provide in general. However, in order to take advantage of this the container's caching behavior needs to be configured properly.

The EJB specification defines three *commit options* for entity beans - A, B and C. The latter can be used to configure the container's behavior with respect to when transitions between states are triggered

and when ejbLoad and ejbStore methods are invoked. This in turn determines what is cached across transactions: objects without identity (commit option C), objects with identity (commit option B) or objects with data (commit option A). However, the specification does not mandate support for all three options and most containers currently on the market do not support them explicitly. Nevertheless, all containers usually provide some means for configuring their caching behavior. In the following, we will discuss the most crucial issues related to the configuration of caching behavior and will use BEA WebLogic Server to illustrate the points we make. For a detailed discussion and performance analysis of the three different commit options we refer the reader to [4].

As we already pointed out, most containers invoke ejbLoad at the point when an entity bean is first accessed from the context of a transaction. When the transaction commits, ejbStore is called to store updates in the underlying store. This ensures that new transactions always use the latest version of the entity's persistent data, and always write this data back to persistent storage upon committing. In certain circumstances, however, this default behavior may lead to excessive database calls and performance degradation. Our aim here is to configure the container in such a way that calls to the database (ejbLoad and ejbStore) are minimized. For example, for beans that are never modified calls to ejbStore can be spared. In WebLogic Server, this can be done by declaring the bean as `read-only` in the "weblogic-ejb-jar.xml" deployment descriptor by use of the so-called "read-only concurrency strategy" [1]. While declaring a bean as read-only completely eliminates calls to ejbStore, the same does not apply to ejbLoad calls. This is because even though the bean is never modified through the EJB layer, this does not prevent direct updates of its underlying data external to the J2EE application. Therefore, periodic calls to ejbLoad are still needed to keep cached data up-to-date.

Access to entity beans that are only occasionally updated can also be optimized by using the so-called *Read-Mostly Pattern* [1]. The idea is to implement a read-only entity bean and a separate read-write entity bean, mapping both of them to the same underlying data. To read the data, you use the read-only entity bean. To update the data, you use the read-write entity bean.

For situations where only a single server instance ever accesses the data of a particular entity bean, calling ejbLoad at the start of each transaction is unnecessary and can be eliminated. Only one initial call is needed to load the data from persistent storage. Afterwards this data can be cached and accessed by many transactions without further calls to ejbLoad. Because no other clients or systems update the underlying storage, the cached version of the entity data is always up-to-date. In WebLogic Server this can be achieved by setting the so-called `db-is-shared` deployment parameter to "false" in the bean's deployment descriptor.

Before we finish with this section lets say a couple of words about concurrency control for entity beans. There are basically two options for enforcing concurrency control when using entity beans. The application server can choose to use it's own algorithm to enforce serializability (for example by employing some form of a locking protocol). Alternatively, the application server can delegate concurrency control to the underlying data store. Although having control of concurrent access to entity beans provides the container with more possibilities to cache data, practical experience shows that delegating concurrency control to the DBMS usually achieves better concurrency and results in higher throughput. Therefore we recommend the second alternative for most applications.

We are now going to examine some of the techniques described above and study the performance gains that they provide. Before doing this we introduce the ECperf benchmark, which we use as a basis to conduct our performance studies.

4 The ECperf Benchmark

ECperf is a newly released J2EE benchmark application prototyped and built by Sun in conjunction with application server vendors including BEA Systems, IBM, iPlanet, Oracle, Borland, Macromedia, Hewlett Packard and IONA. Server vendors can use ECperf to measure, optimize and showcase their product's performance and scalability. Users, on the other hand, can use it to gain a better understanding and insight into the tuning and optimization issues surrounding the development of modern J2EE applications. This is exactly what we tried to achieve by means of the ECperf benchmark. We deployed ECperf on a BEA Web Logic Server and conducted a number of experiments with it. Our aim was to evaluate the different persistence techniques in J2EE and study how their performance could be improved.

ECperf is composed of a specification and a toolkit. The specification [14] describes the benchmark as a whole, the modeled workload, the running and scaling rules, and finally the operation and reporting requirements. The toolkit provides the necessary code to run the benchmark and measure performance.

The ECperf workload is based on a large distributed application claimed to be big and complex enough to represent a real-world e-business system [14]. The ECperf designers have chosen manufacturing, supply chain management, and order/inventory as the "storyline" of the business problem modeled. As the designers themselves describe it [14], this is a meaty, industrial-strength distributed problem, that is heavyweight, mission-critical and requires the use of a powerful and scalable infrastructure. Most importantly,

it requires the use of interesting middleware services, including distributed transactions, clustering, load-balancing, fault-tolerance, caching, object persistence, and resource pooling among others. It is those services of application servers that are stressed and measured by the ECperf benchmark.

ECperf models businesses by using 4 domains [14]:

1. The *Customer Domain* which handles customer orders and interactions.
2. The *Manufacturing Domain* which performs "Just In Time" manufacturing operations.
3. The *Supplier Domain* which handles interactions with external suppliers.
4. The *Corporate Domain* which is the master keeper of customer, product, and supplier information.

Figure 2 illustrates the 4 ECperf business domains and gives some examples of typical transactions run in each domain.

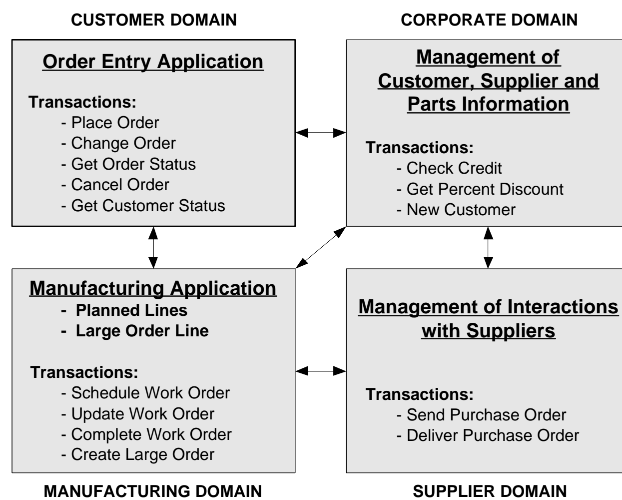

Figure 2: The ECperf Business Model

We include a brief overview of these four domains as they are described in the ECperf specification itself [14]:

4.1 The Customer Domain

Work in the customer domain is OLTP in nature. An order entry application runs in this domain whose functionality includes adding new orders, changing an existing order and retrieving the status of a particular order or all orders of a particular customer. Orders are placed by individual customers as well as by distributors. Orders placed by distributors are called *large orders*.

4.2 The Manufacturing Domain

This domain models the activity of production lines in a manufacturing plant. Products manufactured by the plant are called *widgets*. Manufactured widgets are also called *assemblies*, since they are comprised of *components*. The Bill of Materials (BOM) for an assembly indicates the components needed for producing it. Both assemblies and components are commonly referred to as *parts*. There are two types of production lines, namely *planned lines* and *large order lines*. Planned lines run on schedule and produce a pre-defined number of widgets. Large order lines run only when a large order is received from a customer such as a distributor. Manufacturing begins when a *work order* enters the system. Each work order is for a specific quantity of a particular type of widget. When a work order is created, the Bill of Materials for the corresponding type of widget is retrieved and the required parts are taken out of inventory. As inventory of parts gets depleted, suppliers need to be located and *purchase orders (POs)* need to be sent out. This is done by contacting the supplier domain.

4.3 The Supplier Domain

This domain is responsible for interactions with suppliers. The supplier domain decides which supplier to choose based on the parts that need to be ordered, the time in which they are required and the price quoted by suppliers. The company sends a purchase order (PO) to the selected supplier. When parts are received from the supplier, the supplier domain sends a message to the manufacturing domain to update inventory.

4.4 The Corporate Domain

This domain manages the global list of customers, parts and suppliers. Credit information, including credit limits, about all customers is kept solely in a database in the corporate domain. This is to provide maximal security and privacy.

4.5 The ECperf Application Design

All the activities and processes in the four domains described above are implemented using EJB components (adhering to the EJB 1.1 specification [12]) assembled into a single J2EE application which is deployed on the *System Under Test (SUT)*. The only exception is for the interactions with suppliers which are implemented using a special Java servlet called *Supplier Emulator* that runs on a separate machine. The latter is assembled into a separate application which is deployed in a Java-enabled web server. The supplier emulator provides the supplier domain with a way to emulate the process of sending and receiving purchase orders to/from suppliers. The supplier emulator accepts a purchase order from the BuyerSes session bean in the supplier domain, processes the purchase order, and then delivers the items requested to the ReceiverSes session bean after sleeping for an amount of time based

on the lead time of the component. This interaction is depicted in Figure 3:

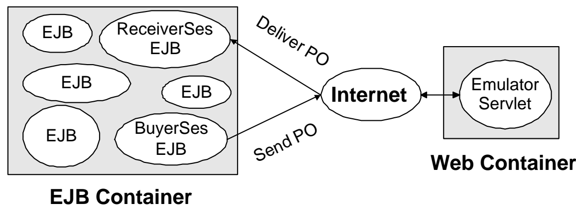

Figure 3: Interaction with the Supplier Emulator

The workload generator is implemented using a multithreaded Java application called *ECperf Driver*. The latter is designed to run on multiple client machines, using an arbitrary number of Java Virtual Machines to ensure that it has no inherent scalability limitations. A relational DBMS is used for data persistence and all data access operations use entity beans which are mapped to tables in the ECperf database. Both container (CMP) and bean-managed (BMP) persistence is supported.

The throughput of the ECperf benchmark is driven by the activity of the order entry and manufacturing applications. The throughput of both applications is directly related to the chosen *Transaction Injection Rate (Ir)*. The latter determines the number of order entry requests generated and the number of work orders scheduled per second. Note, that the relationship between the injection rate and the total number of transaction requests (order entry and work order transactions) that are generated per second is not straightforward. We refer the reader to the ECperf specification for further information [12]. In any case, to increase throughput, the injection rate needs to be increased. The summarized performance metric provided after running the benchmark is called *BBops/min* and it denotes the average number of successful Benchmark Business OPerationS per minute completed during the measurement interval. BBops/min is composed of the total number of business transactions completed in the customer domain, added to the total number of work orders completed in the manufacturing domain, normalized per minute. The benchmark can be run in two modes. In the first mode only the order entry application in the customer domain is run, while in the second mode both the order entry and the manufacturing applications are run. Because of lack of space, we will not go into further details on the ECperf EJBs, the database model and transactions implemented. Readers interested in more details are referred to [14].

4.6 Our ECperf Deployment Environment

We deployed ECperf on the environment depicted in Figure 4.

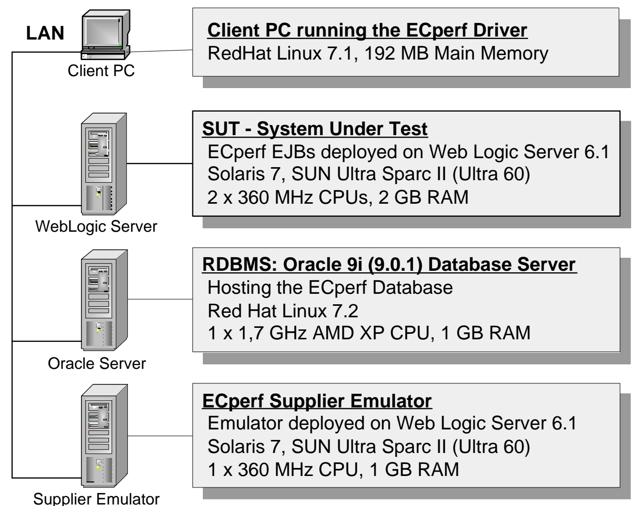

Figure 4: ECperf Deployment Environment

Now that we have introduced the ECperf application, lets return to entity beans and see how they are used in ECperf.

5 Evaluating Entity Bean Performance using ECperf

In this section we compare Bean-Managed Persistence (BMP) with Container-Manager Persistence (CMP) in terms of performance and discuss the issues that drive the choice between them. In addition, we provide some guidelines for improving BMP performance.

ECperf offers both BMP and CMP versions of all entity beans used. We conducted experiments, first with BMP and then with CMP, in order to gain an understanding of how big the performance difference was. We were quite surprised that ECperf performed much worse with BMP than with CMP. Monitoring the database server, we noticed that in the BMP version of ECperf, entity bean data was being written to the database at every transaction commit, even if no changes had been made. We modified the BMP code to check if data had been modified and only in this case update the database [11]. As a result throughput soared by a factor of 2, but performance was still worse than with CMP.

Figure 5 shows the ECperf results that we obtained with our optimized BMP code compared to the results that we obtained with CMP. In these experiments we run the order entry application under different transaction injection rates. The first graph compares average throughput (order entry transactions per min) relative to the average throughput achieved when running with an injection rate of 10. The second graph shows the average transaction commit time (in ms) of the order entry transactions. As we can see CMP performs substantially better as we raise the injection rate beyond 30.

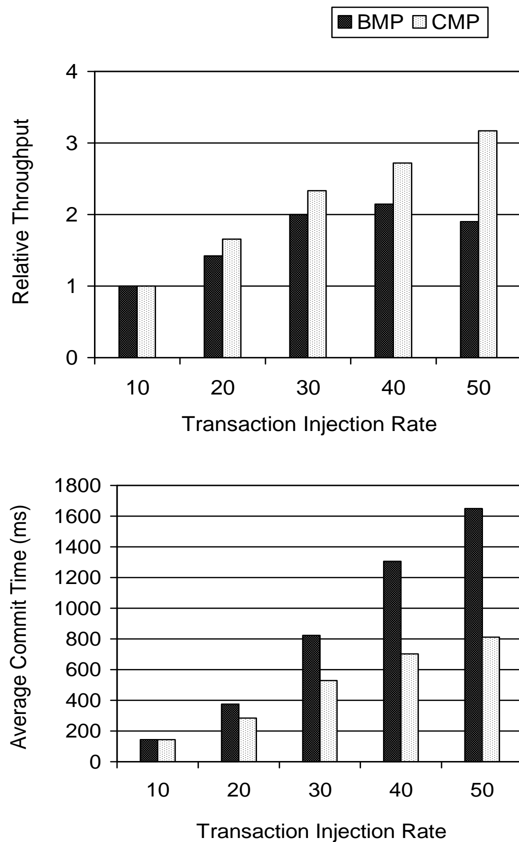

Figure 5: ECperf Results with BMP vs. CMP

As argued in [11] there are some important reasons for this performance difference. Most importantly, giving the container control over the data access logic, allows for some automatic optimizations usually not implemented in BMP code. For example, the container can monitor which fields of an entity bean are modified during a transaction and make sure that only these fields are written to the database at commit time. This minimizes database access calls and avoids doing unnecessary work. Another optimization that is usually provided is related to the loading of entity beans. With BMP loading an entity bean usually requires 2 database calls:

1. ejbFind to find the respective database record and retrieve its primary key.

2. ejbLoad to read the entity bean's data from the database.

With CMP these steps are usually transparently combined into a single database access retrieving both the primary key and the data. Similar optimization can also be applied when loading a collection of N entity beans. With BMP this would require N+1 database calls - 1 ejbFind and N ejbLoads. With CMP the container can be configured to automatically combine the N+1 calls into a single call. We should note here that not all containers currently available implement all of the above optimizations. However, the major ones do, and it should be expected that as containers mature, more and more optimizations will be automatically provided by CMP.

Coming back to our results with ECperf (Figure 5), we see that the extra database calls when using BMP lead not only to higher response times but also to the system getting saturated much more quickly. As a result, throughput starts to drop as we go beyond an injection rate of 40. So, generally speaking, if configured properly CMP usually performs much better than BMP. Therefore, our recommendation is to use CMP instead of BMP whenever it is possible. However, there are situations where one may not be able to use CMP. For example, in cases where the container does not directly support the persistence mechanism used or it supports it but some complex mappings need to be defined that are not supported. In such cases BMP holds an advantage over CMP, because it not only allows an arbitrary persistence mechanism to be used, but provides complete control on how beans are mapped to storage structures. We will now briefly discuss some common techniques that could be applied to improve BMP performance [6]. First of all, the N+1 database calls problem described earlier, can be eliminated by using the so-called *Fat Key Pattern*. For lack of space, we are not going to present this pattern here and refer interested readers to [7]. Another thing that could be done is to make sure BMP code only uses parameterized prepared SQL statements [8]. This reduces the load on the DBMS by allowing it to reuse cached execution plans for statements that were already prepared. Most application servers provide a *PreparedStatement Cache* as part of the connection pool manager. The application server keeps a list of prepared statements and when an application calls prepareStatement on a connection, the server checks if the statement has already been prepared. If that is the case, the PreparedStatement object is found in the cache and is directly returned to the application. This reduces the number of calls to the JDBC Driver and improves response times. Most containers allow the size of the prepared statement cache to be tuned for optimal performance.

6 Elimination of Persistence Bottlenecks

In this section we present some general guidelines on how persistence bottlenecks of J2EE applications could be approached and eliminated. We stick to our running example application - the ECperf benchmark and use it as a basis to illustrate the points we make.

6.1 The ECperf Persistence Bottleneck

Apart from running ECperf with Oracle we also conducted some experiments with Informix. However, surprisingly enough, the benchmark was exhibiting quite a different behavior when run with Informix. More specifically, the persistence layer was turning into a system bottleneck and preventing the benchmark to stress the application server and measure its performance. We now take a closer look at the sources of this problem and then proceed to offer a concrete solution to eliminate the persistence bottleneck.

Figure 6 depicts our new deployment environment with Informix as a database server.

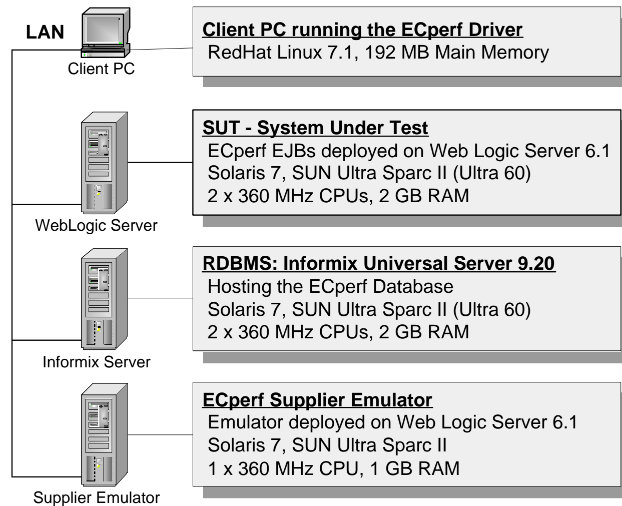

Figure 6: Deployment Environment with Informix

When we ran ECperf out-of-the-box in this environment we monitored the database and observed very high data contention levels. We should note here that unlike Oracle, Informix employs a pessimistic scheduler which uses a locking-based concurrency control technique based on the popular 2PL (2-Phase-Locking) Protocol [16]. Under 2PL data items are locked before being accessed. Concurrent transactions trying to access locked data items in conflicting mode are either aborted or blocked waiting for the locks to be released. When running ECperf we noticed that large amounts of data access operations were resulting in lock conflicts which were blocking the respective transactions. A high proportion of the latter were eventually being aborted because of either timing out or causing a deadlock. As a result very poor throughput levels were achievable and raising the injection rate beyond 2 caused a sudden drop in throughput - a phenomenon known as *Data Thrashing* [15].

The first thing that comes to mind when trying to reduce data contention is to decrease the locking granularity [16]. After setting up Informix to use row-level locks (instead of page-level locks) we observed a significant increase in throughput. To further optimize the data layer we tried decreasing the isolation level [9]. We configured all entity beans to use the SQL `TRANSACTION_COMMITTED_READ` isolation level although this could compromise data consistency and integrity. However, the ECperf specification [14] doesn't place a restriction with respect to this.

While these obvious optimizations could help to alleviate the identified bottleneck, they could not eliminate it and make ECperf behave as originally intended.

6.2 Getting to the Core of the Problem

After conducting a number of experiments and carefully monitoring the database we noticed the following: the crucial transaction **scheduleWorkOrder** of the WorkOrderSes bean was taking relatively long to complete, while holding exclusive locks on some highly demanded database tables. The transaction first creates a new work order entity, then identifies the components that make up the requested assembly in the Bill of Materials and assigns the required parts from inventory. The transaction can also cause calls into the Supplier Domain in case some parts get depleted and new amounts need to be ordered. Figure 7 depicts the execution step-by-step.

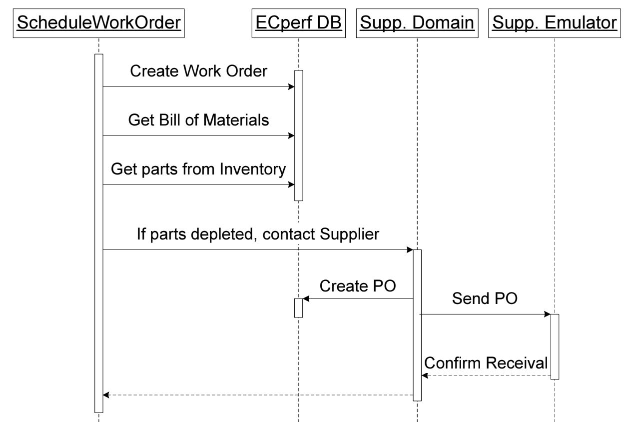

Figure 7: The scheduleWorkOrder Transaction

The scheduleWorkOrder transaction proceeds as follows:

1. Create a work order
   - insert a row in the `M_WORKORDER` table
2. Start processing the work order (stage 1 processing)
   - get the Bill of Materials needed
   - assign required parts from inventory
3. If parts need to be ordered send a Purchase Order
   - insert rows in the `S_PURCHASE_ORDER` and `S_PURCHASE_ORDERLINE` tables

- send the purchase order to the supplier emulator - order is sent in XML format through HTTP

We identified two problems with this design of the scheduleWorkOrder transaction. First, sending the purchase order (the last step) delays the transaction while holding locks on the previously inserted table and index entries. We monitored the database lock tables and observed that indeed most of the lock conflicts were occurring when trying to access the `M_WORKORDER`, `S_PURCHASE_ORDER` and `S_PURCHASE_ORDERLINE` tables or their indices. This supported our initial suspicion that it was the access to these tables that was causing the bottleneck.

The second problem is that the sending step is not implemented to be undoable. In other words once a purchase order is sent to the supplier emulator, its processing begins and this processing is not cancelled even if the scheduleWorkOrder transaction that sent the order is eventually aborted. Indeed, if this occurs, all actions of the scheduleWorkOrder transaction are rolled back except for the sending step. As a result, the respective purchase order is removed from the `S_PURCHASE_ORDER` table, but the emulator is not notified that the order has been cancelled and continues processing it. Later when the order is delivered no information about it will be found in the database and an exception will triggered. So while the first problem has to do with data contention and performance, the second one concerns the scheduleWorkOrder transaction's atomicity and semantics.

In the following we are going to suggest some minor modifications of the benchmark that aim at circumventing the identified bottleneck. Our approach is to break up the scheduleWorkOrder transaction into smaller separate transactions. This is known as *Transaction Chopping* in the literature [16]. The main goal is to commit update operations as early as possible, so that respective locks are released. We strive to isolate time-consuming operations in separate transactions that do not require exclusive locks. At the same time we ensure that transaction semantics are correct.

We have identified and proposed two different solutions to the identified problems. In the first one we keep adhering to the EJB 1.1 specification, while in the second we utilize some services defined in the EJB 2.0 specification. In this paper we will only consider the second solution since it has some significant advantages over the first one. Readers interested in the EJB 1.1 solution are referred to [5].

6.3 Utilizing Messaging and Asynchronous Processing

The problem with the scheduleWorkOrder transaction was that the sending of the purchase order may delay the transaction while holding highly demanded locks. On the one hand, we want to move this step into a separate transaction to make scheduleWorkOrder finish faster. On the other hand, we need to guarantee that the sending operation is executed atomically with the rest of the scheduleWorkOrder transaction. In other words, if the transaction commits, the purchase order should be sent, if the transaction aborts the purchase order should be destroyed and never sent. In fact, as already discussed above, this atomicity is not guaranteed by the given design of ECperf. However, we believe that this is what the correct behavior of the scheduleWorkOrder transaction should be. We would like to note that in the given situation we do not have the requirement that the sending of the purchase order must be executed to its end before scheduleWorkOrder commits. We only need to make sure that, provided that it commits, the sending operation is eventually executed. This situation lends itself naturally to *Asynchronous Processing and Messaging.*

Messaging is an alternative to traditional *Request-Reply Processing.* Request-Reply is usually based on Remote Method Invocation or Remote Procedure Call (RPC) mechanisms. Under these mechanisms, a client sends a request (usually by calling a method on a remote object) and is then blocked, waiting until the request is processed to its end. This is exactly our situation above with the sending of the purchase order. This blocking element prevents the client from performing any processing in parallel while waiting for the server - a problem that has long been a thorn for software developers and whose solution has led to the emergence of *Messaging* and *Message-Oriented Middleware (MOM)* as an alternative to Request-Reply. In fact, it is a solution defined long time ago with the advent of the so-called *Queued Transaction Processing Models* [3].

In a nutshell, the idea behind messaging is that a middleman is introduced, sitting between the client and the server [10]. The middleman receives messages from one or more message producers and broadcasts those messages to possibly multiple message consumers. This allows a producer to send a message and then continue processing while the message is being delivered and processed. The message producer can optionally be later notified when the message is completely processed. A special type of messaging is the so-called *Point-To-Point Messaging* in which each message is delivered to a single consumer. Messages are sent to a centralized *message queue* where they are processed usually on a first-in-first-out (FIFO) basis. Multiple consumers can grab messages off the queue, but any given message is consumed exactly once.

The Java Messaging Service (JMS) [13] is a standard Java API for accessing MOM infrastructures and the EJB 2.0 specification [12] integrates JMS with EJB by introducing the so-called *Message-Driven Beans.* The latter are components that act as message consumers in that they receive and process messages de-

livered by the MOM infrastructure. For example in a point-to-point messaging scenario message-driven beans can be used to process messages arriving on a message queue.

6.4 Eliminating the Persistence Bottleneck

We wanted to allow the transaction to commit before the purchase order is sent, but with the guarantee that it will eventually be sent out. By utilizing messaging we can simply send a message to the supplier domain notifying that a new order has been created and must be sent. After this we can commit our transaction and release all locks. A dedicated message-driven bean can be deployed in the Supplier Domain to handle incoming messages by sending orders out to the emulator. We only need to make sure that the operation that sends the notification message to the supplier domain is transactional and is executed as part of the scheduleWorkOrder transaction. This would ensure that the sending of the message is executed atomically with the rest of the transaction. Furthermore, modern messaging infrastructures provide a *Guaranteed Message Delivery* option which ensures that once a transaction is committed all messages it has sent will be delivered even if respective consumers were down at the time of sending. Taking advantage of this property, we can claim that if the scheduleWorkOrder transaction creates a purchase order and then successfully commits, we have the guarantee that the supplier domain will eventually be notified and the purchase order will be sent out to the emulator. The latter enables us to safely move our sending step into a separate transaction and execute it asynchronously. Figure 8 illustrates this solution.

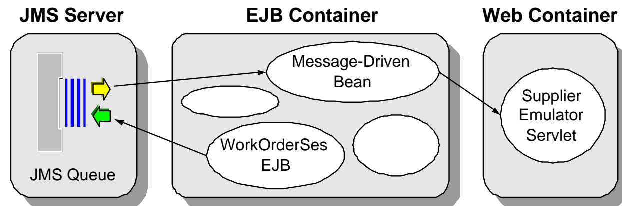

Figure 8: Sending purchase orders asynchronously.

The new design of the scheduleWorkOrder transaction is depicted in Figure 9.

Now lets examine how the new design of the scheduleWorkOrder transaction affected the behavior of the benchmark. Figure 10 compares throughput (BBops/min) achieved with Informix when running ECperf out-of-the-box against throughput achieved after implementing our messaging-based redesign. All data is relative to the throughput (Bbops/min) that we obtained when running ECperf out-of-the-box with injection rate of 1. As we can see throughput increases linearly up to injection rate of 6 and then gradually starts to drop as we go beyond injection rate of 10.

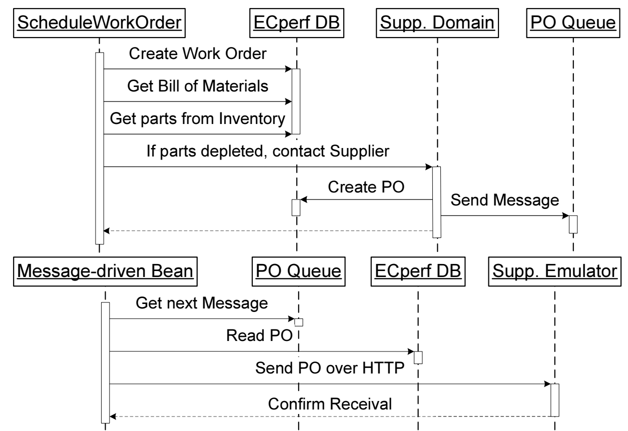

Figure 9: New design of the scheduleWorkOrder transaction.

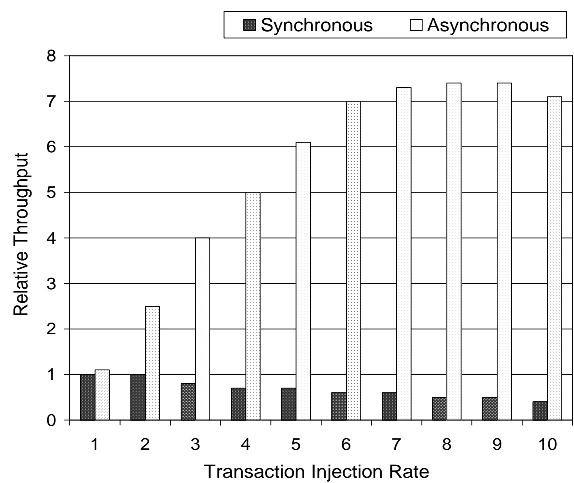

Figure 10: Synchronous vs. Asynchronous variant of ECperf with Informix

This is because at this point the application server is saturated, i.e. the CPU utilization of the WebLogic Server process approaches 100%. So, in this case the application server becomes the bottleneck. However, this is exactly the intended behavior of the benchmark, since its aim is to stress the application server and measure the maximum throughput that it can achieve. We should note here that all tests were run using a single instance of WebLogic Server. The latter was not able to utilize both CPUs of the machine and we expect much better results if a multi-instance WebLogic Cluster is run on the same hardware. However, what is more important is that the new design of the scheduleWorkOrder transaction eliminated the persistence bottleneck and made the benchmark behave as intended.

Not only does the asynchronous design bring a big performance advantage, but it also eliminates the second problem that we mentioned regarding the atomicity of the scheduleWorkOrder transaction. There is no

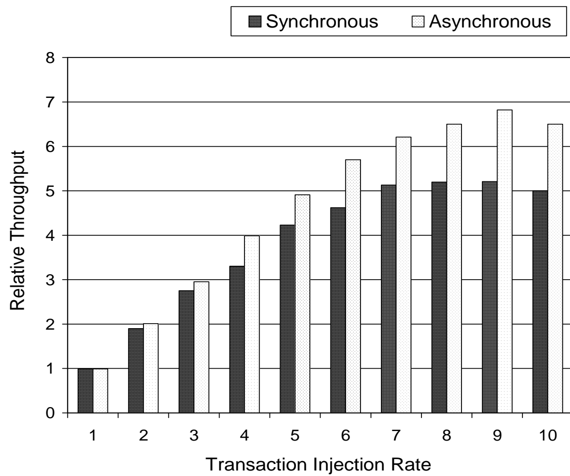

Figure 11: Synchronous vs. Asynchronous variant of ECperf with Oracle 9i

way for a purchase order to be cancelled after it has been sent to the supplier emulator. A further benefit that we get is related to the application's reliability. Under the original design if the supplier emulator is down at the time a new purchase order is being created, the scheduleWorkOrder transaction will be aborted after timing out and all its work will be lost. With the new design the notification message will be sent successfully and although the sending of the order will be delayed until the emulator comes up, the scheduleWorkOrder transaction will be able to commit successfully.

ECperf has been planned to be a DBMS-independent benchmark and therefore some changes are required to make this a reality. We submitted our optimization proposals to the ECperf Expert Group at Sun, where they have been discussed and addressed. Although it is too late to make modifications to the 1.0 version of the benchmark, the ECperf Expert Group vowed to eliminate the problems that we raised with the next version of the benchmark.

One might wonder why the bottleneck was not noticed earlier. The answer is that everyone had only been testing with Oracle where an optimistic multi-version concurrency control protocol is used. The important advantage of this protocol is that it never blocks read operations even when concurrent updates are taking place. For this particular workload Oracle's protocol obviously proved to perform much better. However, we will now show that even with an Oracle DBMS our redesign brings some significant performance and reliability benefits.

Figure 11 compares the synchronous and asynchronous variants of ECperf when run with Oracle. We can see that the performance gain of the asynchronous variant is considerably lower in this case, but it increases steadily as we raise the injection rate. The average CPU utilizations of the database server and the

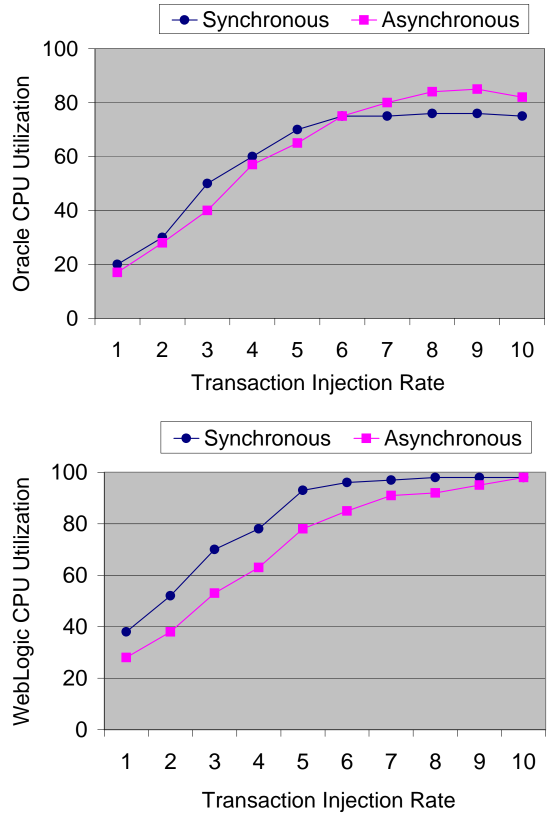

Figure 12: CPU Utilization of the Database Server and the Application Server

application server during the experiments are shown on Figure 12. These are very rough approximations, but should serve to give us a picture of the behavior of the benchmark under load. We can see that under the synchronous variant of ECperf the CPU utilization of the application server approaches 100% at injection rate of 6. This explains why there is hardly any increase in throughput at higher injection rates. Under the asynchronous variant, the CPU utilization of the application server is constantly lower because purchase orders are not sent immediately. As as result, the application server is saturated much later, namely at injection rate of 10 instead of 6. This enables us to achieve higher throughput levels at injection rates beyond 6 and explains why the database server's CPU utilization is higher at these rates. We can see that as far as the database server is concerned, there is almost no difference in CPU utilization. This is because whether we send purchase orders synchronously or asynchronously does not affect the operation of the Oracle database very significantly.

However, we must note that the scenario under which we carried out our experiments is not realistic in the sense that the supplier emulator is contacted over a Local Area Network (LAN). In real life contacting the

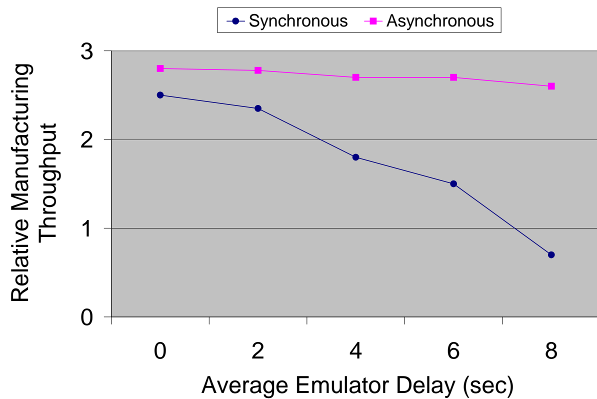

Figure 13: Manufacturing Throughput as we increase the supplier emulator's delay (network delay)

supplier emulator might take much longer if we have to go over a Wide Area Network (WAN). This would result in much higher network delays and would further degrade the performance of the synchronous ECperf design. In order to illustrate this, we programmed the emulator to impose an artificial delay before confirming receipt of the purchase order. We then carried out some experiments under the same transaction injection rate, but with different length of the simulated network delay. Figure 13 shows the impact of the emulator delays on the throughput of the manufacturing application. Obviously, there was hardly any impact on the asynchronous variant of ECperf because the delays were not blocking the transactions in the manufacturing domain. However, this was not the case for the synchronous variant of ECperf where the delays were making the scheduleWorkOrder transaction finish slower and in this way were directly affecting the throughput of the manufacturing application. We can see how quickly throughput drops as we increase the length of the network delay.

7 Summary and Conclusions

In this paper we examined the major persistence techniques provided in the J2EE platform and evaluated their performance in the context of a realistic application. We showed how easily the data access layer of J2EE applications can become a system bottleneck and provided some guidelines on how such bottlenecks could be approached and eliminated. While doing this we presented a number of methods for improving the way persistent data is managed in J2EE applications. Besides techniques well-known in the database community (for example exploiting locking granularity and isolation levels) we discussed how caching services and asynchronous processing could be exploited in order to reduce the load on the database and improve system performance and reliability. We studied the performance difference between Container-Managed and Bean-Managed Persistence and showed that in general CMP utilizes the container's caching services much better than BMP. In addition, we demonstrated through a practical example the performance gains and reliability benefits that asynchronous message-based processing could provide over traditional request-reply processing.

Acknowledgments

We gratefully acknowledge the many fruitful discussions with Shanti Subramanyam and Akara Sucharitakul from Sun Microsystems Inc, Dan Fishman and Steve Realmuto from BEA Systems and Chris Beer from SPEC.

References

[1] Bea Systems, Inc. WebLogic Server Documentation. Technical report. `http://e-docs.bea.com/wls/`.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[3] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, Inc., 1997.

[4] P. Brebner and S. Ran. Entity Bean A, B, C's: Enterprise Java Beans Commit Options and Caching. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms - Middleware*, 2001.

[5] S. Kounev. Eliminating ECperf Persistence Bottlenecks when using RDBMS with Pessimistic Concurrency Control. Technical Report `http://www.dvs1.informatik.tu-darmstadt.de/~skounev`, Technical University of Darmstadt, Germany, September 2001.

[6] S. Kounev and A. Buchmann. Performance Issues in E-Business Systems. In *Proc. of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet - SSGRR-2002w*, 2002.

[7] F. Marinescu. *Enterprise Java Beans Design Patterns*. John-Wiley & Sons, Inc., 2002.

[8] B. Newport. Why prepared statements are important and how to use them properly. *TheServerSide.com J2EE Community - http://www.theserverside.com/*, 2001.

[9] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2nd edition, 2000.

[10] Ed Roman, S. Ambler, and T. Jewell. *Mastering Enterprise Java Beans II and the Java 2 Platform, Enterprise Edition*. John-Wiley & Sons, Inc., 2002.

[11] A. Sucharitakul. Seven Rules for Optimizing Entity Beans. *Java Developer Connection - http://www.java.com/*, 2001.

[12] Sun Microsystems, Inc. Enterprise JavaBeans 1.1 and 2.0. Specifications. `http://java.sun.com/products/ejb/`.

[13] Sun Microsystems, Inc. Java Message Service API 1.0.2. Specification. `http://java.sun.com/products/jms/`.

[14] Sun Microsystems, Inc. The ECperf 1.0 Benchmark. Specification, June 2001. `http://java.sun.com/j2ee/ecperf/`.

[15] Y. Tay, N. Goodman, and R. Suri. Locking Performance in Centralized Databases. *ACM Transactions on Database Systems*, 10/4, 1985.

[16] G. Weikum and G. Vossen. *Transactional Information Systems - Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2002.