Performance Issues in E-Business Systems

Samuel D. Kounev and Alejandro P. Buchmann
Databases and Distributed Systems Group
Department of Computer Science
Darmstadt University of Technology
D-64283 Darmstadt, Germany
{*skounev,buchmann*}*@informatik.tu-darmstadt.de*

Abstract— **Performance and scalability issues in e-business systems are gaining in importance as we move from hype and prototypes to real operational systems. Typical for this development is also the emergence of standard benchmarks of which TPC-W for transactional B2C systems and ECperf for performance and scalability measurement of application servers are two of the better known examples. In this paper we present an experience report with the ECperf benchmark defined by Sun and discuss performance issues that we observed in our implementation of the benchmark. Some of these issues are related to the specification of the benchmark, for which we made suggestions how to correct them and others are related to database connectivity, locking patterns, and the need for asynchronous processing.**

Keywords— **benchmarking, ECperf, J2EE, scalability, performance, middleware, messaging**

I. Introduction

OVER the past couple of years, the Java 2 Enterprise Edition Platform (J2EE) has established itself as the technology of choice for developing modern e-business solutions. A large part of this success is due to the fact that J2EE is not a proprietary product such as Microsoft .NET, but rather an industry standard, developed as the result of a large industry initiative led by Sun Microsystems. The goal of this initiative was to establish a standard middleware framework for developing enterprise-class distributed applications in Java. Over 30 enterprise software vendors have participated in this effort and have come up with their own implementations of J2EE. The fact that these products are all based on a common standard and that this standard is itself based on Java, ensures that J2EE applications are not only operating-system independent, but also portable across a wide range of middleware platforms - J2EE implementations. This enables customers to develop their applications in a platform-independent manner and gives them a wide selection of products where they can later deploy them. Giving customers this freedom of choice encourages best-of-breed products to compete and establish themselves in the market. However, once product functionality is standardized, the focus is placed on the performance and scalability of the underlying platforms. It is exactly here that companies strive to distinguish their products in the market and gain competitive advantage over the competition. This development has led to the emergence of standard benchmarks for measuring performance and scalability of middleware products. A typical example is the newly released *ECperf benchmark* which was prototyped and built by Sun in conjunction with J2EE application server vendors under the Java Community Process. 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-based 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 and tests with it. In this paper we present our experience from these experiments and the lessons that we learned. We focus on the problems that we encountered when running the benchmark out-of-the-box and attempting to scale the workload as described in the documentation. In essence, our goal is to use the ECperf benchmark as an example of a realistic application, in order to identify and discuss the factors that have the greatest performance impact on J2EE applications. At the same time we examine the areas that are crucial for scalability and that could often turn into system bottlenecks. In doing this, we consider both issues related to the application's design, as well as issues related to the configuration of the deployment environment. In the end we summarize the lessons that we learned while experimenting with ECperf and present a list of tuning and optimization techniques that could be applied to boost the performance of an arbitrary J2EE application. Before we start lets take a closer look at the J2EE Platform and the role of the ECperf benchmark.

II. The J2EE Platform and ECperf

Fundamentally, the J2EE platform provides an infrastructure that enables the rapid development of large-scale distributed applications in Java. The Enterprise JavaBeans specification [16] is at the heart of this infrastructure and it supplies the component model for developing *EJB components* which are the building blocks of *J2EE applications*. 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 caching, resource-pooling, clustering, transparent-failover, load-balancing and back-end integration to name just a few. The latter are crucial for today's e-commerce systems that need to cope with ever increasing load levels and service demands. It is the *J2EE application server* that takes the role of providing these services. Developers

can concentrate on the business and application logic and rely on the application server to provide the infrastructure needed for scalability and performance.

While the above is also provided by traditional middleware platforms, the J2EE platform takes this paradigm one step further by not only allowing developers to leverage prewritten middleware services provided by the industry, but also allowing them to do that without needing to code to specific middleware APIs. That is, the only thing that is required of developers is to declare the services they need (using so-called deployment descriptors in the EJB terminology). They are not required to write to an API in order to obtain the services desired. This is often termed *implicit middleware* [14] as opposed to *explicit middleware* provided by traditional (for e.g. CORBA-based) middleware infrastructures and Transaction Processing Monitors (TPMs). Combined with the J2EE platform's promise of complete application portability and reusability across any vendor's middleware infrastructure, this brings the "write-once run-anywhere" paradigm of Java into the world of server-side programming and appears to fulfill what until recently could only be dreamed of.

But is it really so simple and easy as it sounds? Are all these promises proven and can developers trust that the server they have chosen is robust enough to bring scalability, reliability and performance to their systems? Obviously, there are many factors affecting performance among them the applicationŠs design, the deployment platform and hardware used. However, a very significant factor is the application server itself that is selected as a middleware platform. This is where benchmarking and performance measurement come into play. The only way one could gain a picture of how good an application server performs is to have it tested by running a realistic application with a realistic workload. However, we have all been able to witness how in the recent months different vendors have been running proprietary benchmarks and coming up with contradictory and biased results in their attempts to push their products and beat the competition. One has to be extremely naive to trust any such results and claims. Needed is a publicly available industry-standard benchmark, built and maintained by the industry itself with no predominance of any particular vendor. Needed also is a committee to monitor and control the testing process in order to prevent speculations and misuse of results, similar to the Transaction Processing Performance Council (TPC) in the area of DBMS benchmarking. This is exactly what the ECperf benchmark and the ECperf review committee claim to bring to the table. ECperf is composed of a specification and a toolkit. The specification [18] 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. ECperf was prototyped and built in conjunction with leading J2EE application server vendors including BEA Systems, IBM, iPlanet, Oracle, Sun Microsystems, Borland and IONA.

Now that we have gotten a better picture of J2EE and the role and goals of ECperf, lets take a look at the workload used in ECperf as a basis for benchmarking performance and scalability. The presentation below is based on the ECperf workload description provided in [18] and [5].

III. The ECperf Workload

The ECperf workload is based on a huge distributed application claimed to be big and complex enough to represent a real-world e-business system [18]. 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 [5], 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 [18]:

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

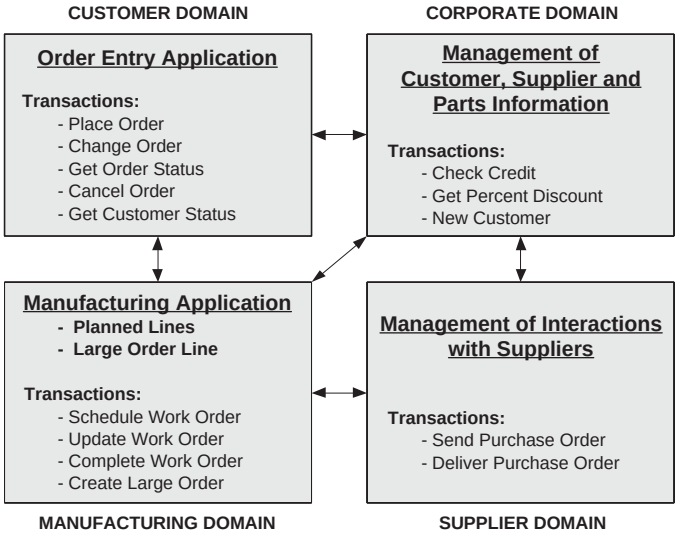

Fig. 1. The ECperf Business Model

Below we walk you through a brief overview of these four domains as they are described in the ECperf specification itself [18]:

A. 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*.

Orders arrive from existing customers as well as new customers. In either case, a credit check is performed on the customer by sending a request to the corporate domain. Various discounts are applied to the order depending on whether the customer is a distributor, repeat or first-time customer. Existing orders may be changed. A customer or salesperson can view the status of a particular order.

B. 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. While work orders in the planned line are typically created as a result of a forecasting application, work orders in the large order line are generated as a result of customer orders. 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 the widgets move through the assembly line, the work order status is updated to reflect progress. Once a work order is complete, it is marked as complete and inventory is updated. 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.

C. 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. The purchase order will include the quantity of various parts being purchased, the site it must be delivered to and the date by which delivery must happen. When parts are received from the supplier, the supplier domain sends a message to the manufacturing domain to update inventory.

D. 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. For each new order, the customer domain requests a credit worthiness check to the corporate domain. Customer discounts are also computed in the corporate domain for each new order or whenever an order is changed.

IV. The ECperf Application Design

All the activities and processes in the four domains described above are implemented using Enterprise Java Bean components (adhering to the EJB 1.1 specification [16]) 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 2:

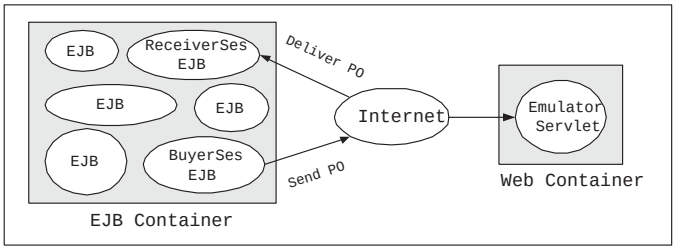

Fig. 2. Interaction with the Supplier Emulator

The workload generator is implemented using a multi-threaded Java application called the *ECperf Driver*. The latter is designed to run on multiple client machines, using an arbitrary number of Java Virtual Machines (JVMs) 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 [16]. 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. Because of the lack of space, we will not go into any further details on the ECperf EJBs, the database model and transactions implemented. Readers interested in more details are referred to [5] for more information.

V. The ECperf Persistence Bottleneck

Now that we have gained a better picture of the ECperf benchmark itself, lets move on to the problems we encountered when putting it to use. We deployed ECperf on the environment depicted in Figure 3.

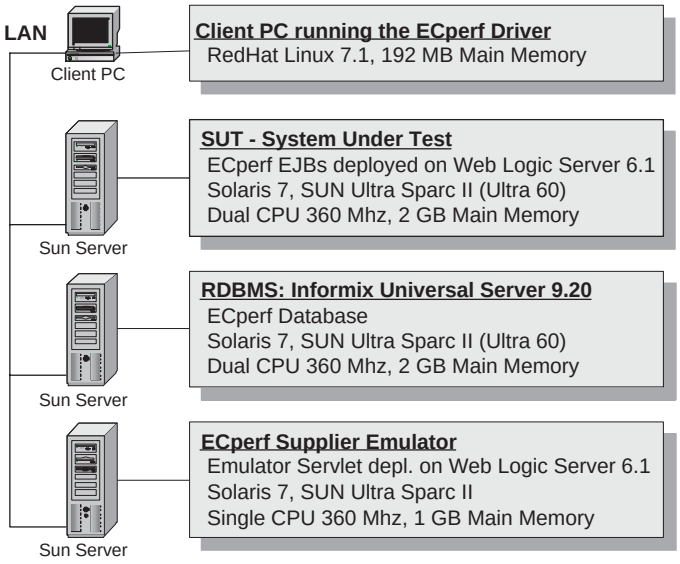

Fig. 3. ECperf Deployment Environment

After running some experiments, we noticed that the benchmark exhibits quite a different behavior depending on the type of database management system (DBMS) that is used for persistence. When deployed with Oracle8i which uses optimistic multi-version concurrency control techniques, the benchmark behaves as intended. However, when deployed with a DBMS utilizing pessimistic locking-based concurrency control techniques (such as Informix Universal Server), the persistence layer seems to turn into a bottleneck preventing one to stress the application server and benchmark 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.

Pessimistic concurrency control schedulers usually employ locking-based protocols such as the popular 2PL in its many variants [21]. 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 we run ECperf out-of-the-box in such an environment we monitored the database and observed very high data contention levels. 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 [19].

The first thing that comes to mind when trying to reduce data contention is to decrease the locking granularity [21]. 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 [13]. We configured all entity beans to use the SQL `TRANSACTION_COMMITTED_READ` isolation level although this could compromise data consistency and integrity at high injection rates. However, the ECperf specification [18] doesn't place a restriction with respect to this.

While the above optimizations could help to reduce the identified bottleneck, they could not completely eliminate it and make ECperf

behave as originally intended. We now take an inside look at the way ECperf is designed in order to gain a better picture of the reasons for the bottleneck.

A. Getting to the Bottom 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. Now, lets follow its execution in a step by step fashion as depicted in Figure 4.

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

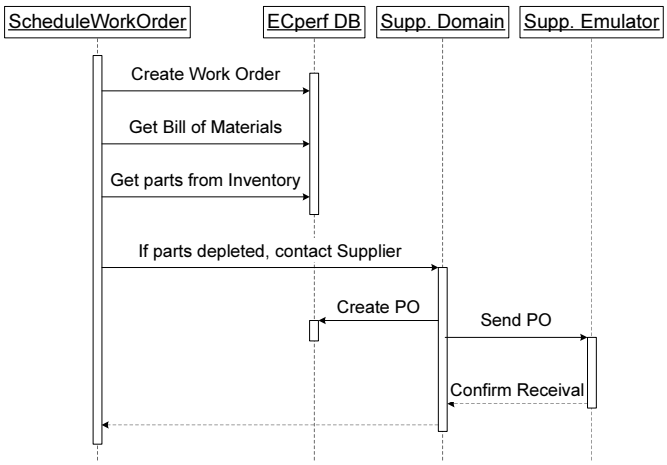

Fig. 4. The scheduleWorkOrder Transaction

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 with the given design of ECperf is that the sending step is not undoable. This means that once a new purchase order is sent to the Supplier Emulator, its processing begins and this processing cannot be cancelled even if the scheduleWorkOrder transaction eventually aborts. Indeed, if that occurs all actions of the transaction will be rolled back except for the sending step. As a result, the respective purchase order will be removed from the `S_PURCHASE_ORDER` table, but the Emulator won't be aware that the order has been cancelled and will continue processing it. Later when the order is delivered no information about it will be found in the database and an exception will be triggered. So while the first problem is 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 [21]. 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 [9].

B. 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 operations. Notice here that in the given situation we do not have the requirement that the sending of the purchase order is 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 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 [14]. 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. The Java Messaging Service (JMS) [17] is a standard Java API for accessing MOM infrastructures and the EJB 2.0 specification [16] 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 delivered by the MOM infrastructure.

C. Eliminating the Persistence Bottleneck

We are now ready to present our solution to the problem with the scheduleWorkOrder transaction. 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 sent a

message to the supplier domain notifying that a new order has been created and must be sent. After this we can safely commit our transaction and release all locks. A dedicated message-driven bean can run in the Supplier Domain to handle incoming messages by sending orders out to the Supplier Emulator. Modern Messaging infrastructures also 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. This property enables us to safely move our sending step into a separate transaction and allow it to execute asynchronously. Figure 5 illustrates this solution.

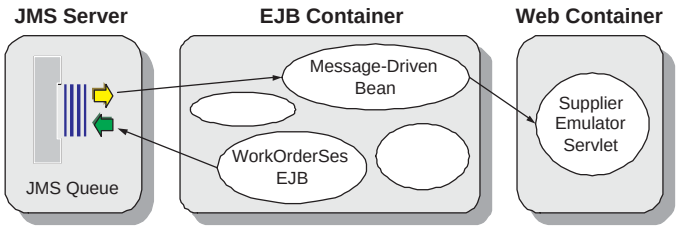

Fig. 5. Sending purchase orders asynchronously.

The new design of the scheduleWorkOrder transaction is depicted in Figure 6.

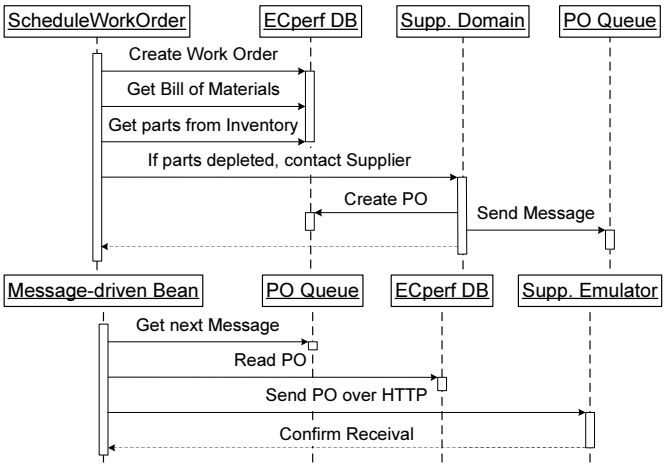

Fig. 6. New design of the scheduleWorkOrder transaction.

Figure 7 compares throughput (BBops/min) achieved when running ECperf out-of-the-box with throughput achieved after implementing our Messaging-based optimization. All data is relative to the throughput (Bbops/min) that we obtained when running ECperf out-of-the-box with injection rate of 1. Note that all tests were run using a single instance of WebLogic Server. We expect even better results if a multi-instance WebLogic Cluster is run on the same hardware.

Not only does this 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 way for a purchase order to be cancelled after it has been sent to the Supplier Emulator. A

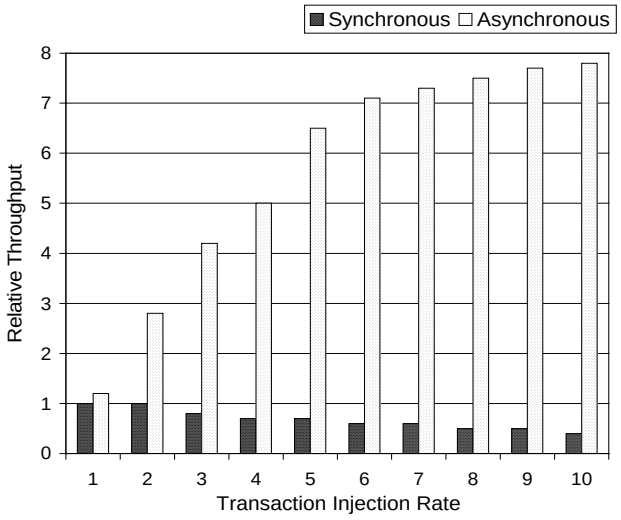

Fig. 7. Synchronous vs. Asynchronous variant of ECperf

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 finish successfully.

Before we conclude this section we want to stress that the elimination of the persistence bottleneck was the crucial step in getting the benchmark up and running in our deployment environment. We could hardly achieve any sensible results before implementing our optimizations. One might wonder why these problems were not noticed earlier. The answer is that everyone had only been testing with Oracle where a completely different concurrency control protocol is used. For this particular workload Oracle's protocol proved to perform much better. However, even with an Oracle DBMS our optimizations bring some significant performance and reliability benefits. In any case, 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 Microsystems Inc., 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.

In the rest of this paper we are going to discuss some general techniques for tuning and optimizing J2EE applications. Some of these techniques we are going to apply to the ECperf benchmark in order to try to give a rough idea of the performance speedup that they can achieve. There are numerous design patterns for writing scalable J2EE applications available in the literature [10]. Since we cannot cover every possible technique in this paper, we are going to focus on the issues that, in our opinion, are most signif-

icant and that are applicable for most J2EE applications and application servers. These issues revolve around the techniques provided by J2EE for data persistence, caching and resource pooling. The latter have proven to be crucial for J2EE performance and practical experience shows that it is exactly in these areas that processing inefficiencies and scalability bottlenecks are usually discovered.

VI. Optimizing Entity Beans

Entity Beans are the natural method provided by J2EE for modeling persistent data. Ever since their introduction, they have been the subject of hot discussions and disputes in the Enterprise Java Community. Some argue that they are so inefficient that one should not even consider using them. Others go to the opposite extreme and take their use to excess. In any case, 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 turn into a performance killer. In this section we will discuss the most important performance issues regarding the use of entity beans and their tuning and optimization. We will be assuming that a relational DBMS is used as an underlying persistence mechanism.

Before we start lets take a quick look at the lifecycle of an entity bean.

A. Entity Bean Lifecycle

According to the EJB specification [16] at each point in time 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 8.

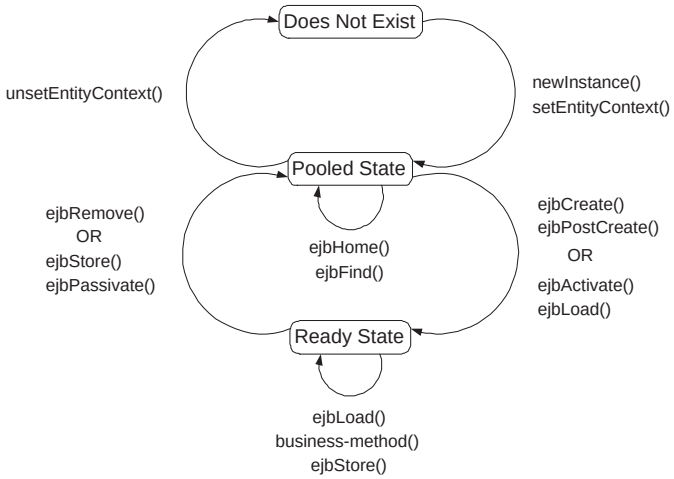

Fig. 8. Lifecycle of an Entity Bean instance.

An entity bean instance's life starts when the container creates the instance by calling *newInstance*. The instance enters a pool of available instances. Each entity bean has its own pool. While the instance is in the pool, it is not associated with any particular entity object identity. All instances in the pool are considered equivalent and the container may use them to execute any of the entity bean's finder methods - *ejbFind*. An instance transitions from the pooled state to the ready state when the container selects that instance to service a client call to an entity bean. When an entity bean instance is in the ready state, the instance is associated with a specific entity object identity and business methods can be invoked on it zero or more times. 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 (when an entity bean is first accessed) in order to load the entity's data. At transaction commit the container calls ejbStore to write back updated data to persistent storage.

B. Configuring 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. 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 (which according to a recent report issued by Meta Group is currently the most popular J2EE application 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 (including WebLogic) 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". For more information on how to do

this refer to [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 EJB application. Therefore, periodic calls to ejbLoad are still needed to keep cached data up-to-data.

Access to entity beans that are only occasionally updated can also be optimized by using the so-called *Read-Mostly Pattern*. 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

more information and examples refer to [1].

For situations where only a single server instance ever accesses 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 data, 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 weblogic-ejb-jar.xml 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 (which is the typical persistence mechanism) usually achieves better concurrency and results in higher throughput. Therefore we recommend the second alternative for most applications.

C. Bean-Managed vs. Container-Managed Persistence

The EJB specification [16] 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 [14] both BMP and CMP have their virtues and drawbacks. In this section we will discuss the issues that drive the choice between BMP and CMP and will provide some guidelines for optimizing entity bean persistence logic.

Lets get back to ECperf. The benchmark offers both BMP and CMP versions of all entity beans. After eliminating the persistence bottleneck we conducted some experiments, first with BMP and then with CMP, in order to gain a picture 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 [15]. As a result throughput soared by a factor of 2, but performance was still worse than with CMP. Figure 9 shows the ECperf results that we obtained with our optimized BMP code compared to the results that we obtained with CMP. The first graph compares average throughput - transactions per min. The second graph compares average response time of the four order entry transactions - NewOrder, ChangeOrder, OrderStatus and CustomerStatus. All data is normalized relative to the respective results obtained for BMP code with injection rate of 5.

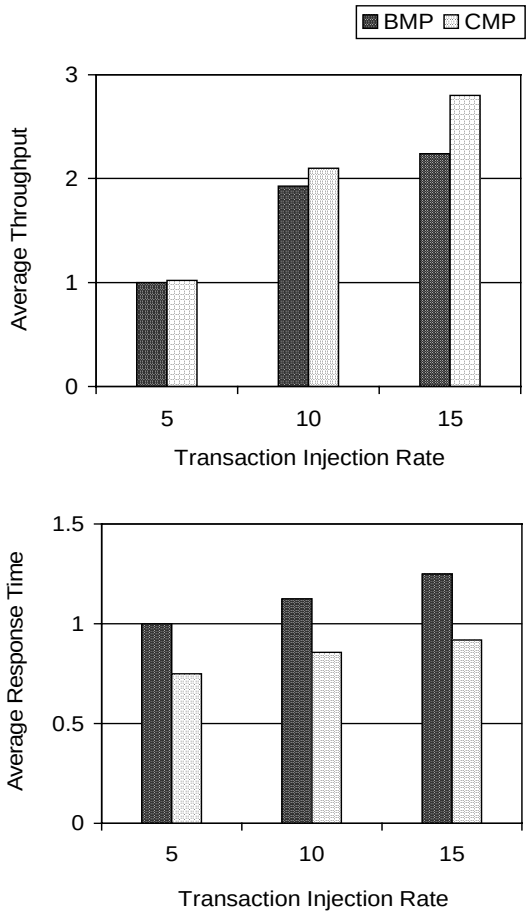

Fig. 9. ECperf Results with BMP vs. CMP

As argued in [15] 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 those fields are written to the database at commit time. This minimizes database access calls and avoids doing unnecessary work. Another optimization which 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 one single call. We should note here that not all containers currently available implement all of the above optimizations. However, the major ones do and we can expect that as containers mature, more and more optimizations will be automatically provided by CMP. Therefore our recommendation is to use CMP instead of BMP whenever that is possible. However, if for some reason you need to use BMP, there are some common techniques that you could apply to increase performance. First of all, you can workaround the N+1 database calls problem described above, by applying the so-called *Fat Key Pattern*. For lack of space, we are not going to discuss this pattern here but refer interested readers to [10] for more information.

Another thing one should remember when using BMP is to always use parameterized prepared SQL statements [11]. This reduces the load on the DBMS by allowing it to reuse cached execution plans for statements that were already prepared. The correct use of prepared statements also lets you take advantage of the so-called *Prepared-Statement Cache* which is usually provided by application servers 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 that statement was previously 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 you to tune the size of the prepared statement cache for optimal performance. Finally, remember to make sure that all SQL statements are closed properly when they are no longer needed. This closes respective database cursors which cost resources in the database.

D. Eliminating Deadlocks

As discussed in [15], when multiple entity beans are involved in a transaction, the sequence in which their ejbStore methods get called at the end of the transaction is undefined. However, although the EJB specification does not mandate this, it is usually assumed that the container invokes ejbStore methods in the sequence in which the respective beans have initially joined the transaction (by having a transactional method invoked on them). Therefore accessing entity beans in different orders from within different transcations may lead to deadlocks in databases which employ locking-based schedulers.

For example, with ECperf, deadlocks were detected when reading order items in different orders in the ChangeOrder and OrderStatus transactions. This is because the order items are not sorted. If a transaction tries to access items X and Y, while another transaction tries to do the same but in the opposite order, a deadlock will result. This problem was identified by the ECperf developers and was fixed in update 1 of the benchmark by making sure that order items are always accessed in sorted order. The take away point here is that in order to minimize deadlocks it is recommended that you access entity beans always in the same order in different transactions throughout your application.

E. Avoid overusing Entity Beans

Even though entity beans are the natural method for managing persistent data in J2EE applications, there are situations when the benefit of using them may not be worth the additional overhead of going through the entity bean layer. A typical example would be an application that needs to present some static server-side data to a client in tabular form - for example information on all employees of a company, the line items of a large order or the characteristics of all products a company produces. In such cases going through the entity bean layer may lead to unacceptable performance degradation. Therefore, 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 a detailed discussion on how to decide when to use SMP instead of entity beans see the "JDBC for Reading" pattern in [10].

VII. Summary of Lessons Learned

We summarize the lessons that we learned while experimenting with ECperf and present a list of tuning and optimization techniques that could be applied to boost the performance of an arbitrary J2EE application. Following is a list of take-away points from the discussions presented in this paper:

1. Use CMP instead of BMP whenever possible.
2. When reading large amounts of data for listing purposes, use SMP.
3. Use the lowest isolation level that does not compromise data integrity.
4. Configure the database's locking granularity properly.
5. Exploit container's caching services to their full extent.
6. Always access entity beans in the same order in all transactions throughout the application.

7. Make transactions as short as possible.
   - Don't allow a transaction to span user interactions, network communications or any other activities that might potentially take a long time. Execute long operations asynchronously.
   - Make sure that transactions are demarcated correctly - check the transaction attribute settings.
8. Use Asynchronous Processing and Messaging instead of traditional Request-Reply Processing whenever possible - Messaging brings significant performance, scalability and reliability benefits.
9. Use your container's facilities for monitoring connection pools, bean pools, threads, transactions and other resources used during operation. At the same time use available tools and facilities to monitor all database servers in use. This might help you discover subtle processing inefficiencies and scalability bottlenecks.
10. If you are using a DBMS for persistence try testing your application with different database servers.

VIII. Conclusions

ECperf specifies a heavy duty problem and the necessary settings of application servers to guarantee an unbiased performance comparison. However, a few problems with the specification of ECperf were identified and solutions proposed to the ECperf Expert Group. In particular, the redefinition of the interaction with suppliers was proposed and the benefits of asynchronous message-based processing were illustrated leading to a performance boost of at least a factor of 5. It was shown that J2EE entity beans are a powerful platform but entity beans must be used carefully. Apparently minor differences in the use of entity beans may result in significant performance gains or losses.

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, Chris Beer from SPEC and our colleagues Christian Haul and Mariano Cilia from Darmstadt University of Technology.

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] TheServerSide.com J2EE Community. The ECPerf homepage. `http://ecperf.theserverside.com/ecperf/`.

[6] S. Deshpande, B. Martin, and S. Subramanyam. Eight Reasons ECperf is the Right Way to Evaluate J2EE Performance. *TheServerSide.com J2EE Community*, 2001. `http://www.theserverside.com/`.

[7] J. Gray and A. Reuter. *Transaction Processing - Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[8] 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.

[9] S. Kounev and A. Buchmann. Optimizing Sun's ECperf Benchmark for Measuring Performance and Scalability of J2EE-Application Servers. In *To appear in Proc. of the 22nd International Conference on Distributed Computing Systems - ICDCS*, 2002.

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

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

[12] C. Pancake and C. Lengauer. High-Performance Java. *Communications of the ACM*, 44, 2001.

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

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

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

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

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

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

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

[20] C. Vawter and Ed Roman. J2EE vs. Microsoft .NET - A comparison of building XML-based web services. *TheServerSide.com J2EE Community - `http://www.theserverside.com/`*, 2001.

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