PERFORMANCE TUNING AND OPTIMIZATION OF J2EE APPLICATIONS ON THE JBOSS PLATFORM

Samuel Kounev, Björn Weis and Alejandro Buchmann

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

Over the past couple of years the JBoss application server has established itself as a competitive open-source alternative to commercial J2EE platforms. Although it has been criticized for having poor scalability and performance, it keeps gaining in popularity and market share. In this paper, we present an experience report with a deployment of the industry-standard SPECjAppServer2004 benchmark on JBoss. Our goal is to study how the performance of JBoss applications can be improved by exploiting different deployment options offered by the JBoss platform. We consider a number of deployment alternatives including different JVMs, different Web containers, deployment descriptor settings and data access optimizations. We measure and analyze their effect on the overall system performance in both single-node and clustered environments. Finally, we discuss some general problems we encountered when running the benchmark under load.

1 Introduction

The JBoss Application Server is the world's most popular open-source J2EE application server. Combining a robust, yet flexible architecture with a free open-source license and extensive technical support from the JBoss Group, JBoss has quickly established itself as a competitive platform for e-business applications. However, like other open-source products, JBoss has often been criticized for having poor performance and scalability, failing to meet the requirements for mission-critical enterprise-level services.

In this paper, we study how the performance of J2EE applications running on JBoss can be improved by exploiting different deployment options offered by the platform. We use SPECjAppServer2004 [1] - the new industry standard benchmark for measuring the performance and scalability of J2EE application servers. However, our goal is not to measure the performance of JBoss or make any comparisons with other application servers. We rather use SPECjAppServer2004 as an example of a realistic application, in order to evaluate the effect of some typical JBoss deployment and configuration options on the overall system performance. The exact version of JBoss Server considered is 3.2.3, released on November 30, 2003. In addition to studying how JBoss performance can be improved, we report on problems we encountered during deployment of the benchmark as well as some scalability issues we noticed when testing under load.

The rest of the paper is organized as follows. We start with an overview of SPECjAppServer2004, concentrating on the business domain and workload it models. We then describe the deployment environment in which we deployed the benchmark and the experimental setting for our performance analysis. After this we study a number of different configuration and deployment options and evaluate them in terms of the performance gains they bring. We start by comparing several alternative Web containers (servlet/JSP engines) that are typically used in JBoss applications, i.e. Tomcat 4, Tomcat 5 and Jetty. Following this, we evaluate the performance difference when using local interfaces, as opposed to remote interfaces, for communication between the presentation layer (Servlets/JSPs) and the business layer (EJBs) of the application. We measure the performance gains from several typical data access optimiza-

[1]SPECjAppServer is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2004 results or findings in this publication have not been reviewed by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjAppServer2004 is located at http://www.spec.org/jAppServer2004.

tions often used in JBoss applications and demonstrate that the choice of Java Virtual Machine (JVM) has very significant impact on the overall system performance. Finally, we report on some scalability and reliability issues we noticed when doing stress testing.

2 The SPECjAppServer2004 Benchmark

SPECjAppServer2004 is a new industry standard benchmark for measuring the performance and scalability of Java 2 Enterprise Edition (J2EE) technology-based application servers. SPECjAppServer2004 was developed by SPEC's Java subcommittee, which includes BEA, Borland, Darmstadt University of Technology, Hewlett-Packard, IBM, Intel, Oracle, Pramati, Sun Microsystems and Sybase. It is important to note that even though some parts of SPECjAppServer2004 look similar to SPECjAppServer2002, SPECjAppServer2004 is much more complex and substantially different from previous versions of SPECjAppServer. It implements a new enhanced workload that exercises all major services of the J2EE platform in a complete end-to-end application scenario.

2.1 SPECjAppServer2004 Business Model

The SPECjAppServer2004 workload is based on a distributed application claimed to be large enough and complex enough to represent a real-world e-business system [Sta04]. The benchmark designers have chosen manufacturing, supply chain management, and order/inventory as the "storyline" of the business problem to be modeled. This is an industrial-strength distributed problem, that is heavyweight, mission-critical and requires the use of a powerful and scalable infrastructure. The SPECjAppServer2004 workload has been specifically modeled after an automobile manufacturer whose main customers are automobile dealers. Dealers use a Web based user interface to browse the automobile catalogue, purchase automobiles, sell automobiles and track dealership inventory.

As depicted in Figure 1, SPECjAppServer2004's business model comprises five domains: *customer domain* dealing with customer orders and interactions, *dealer domain* offering Web based interface to the services in the customer domain, *manufacturing domain* performing "just in time" manufacturing operations, *supplier domain* handling interactions with external suppliers, and *corporate domain* managing all customer, product, and supplier information. Figure 2 shows some examples of typical transactions run in these domains (the dealer domain is omitted, since it does not provide any new services on itself).

The customer domain hosts an *order entry application* that provides some typical online ordering functionality. The latter includes placing new orders, retrieving the status of a particular order or all orders of a given

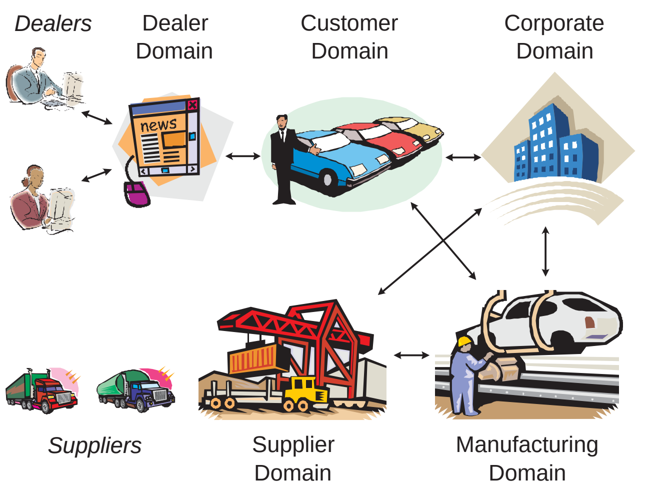

Figure 1: SPECjAppServer2004 Business Model

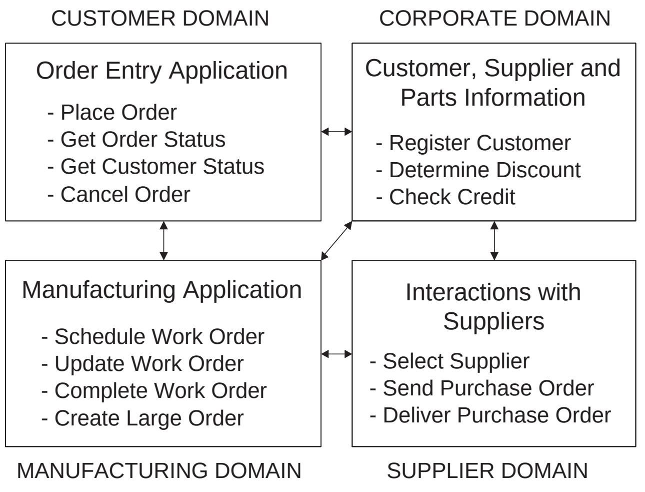

Figure 2: SPECjAppServer2004 Business Domains

customer, canceling orders and so on. Orders for more than 100 automobiles are called *large orders*.

The dealer domain hosts a Web application (called *dealer application*) that provides a Web based interface to the services in the customer domain. It allows customers, in our case automobile dealers, to keep track of their accounts, keep track of dealership inventory, manage a shopping cart, and purchase and sell automobiles.

The manufacturing domain hosts a *manufacturing application* that models the activity of production lines in an automobile manufacturing plant. There are two types of production lines, namely *planned lines* and *large order lines*. Planned lines run on schedule and produce a predefined number of automobiles. Large order lines run only when a large order is received in the customer domain. The unit of work in the manufacturing domain is a *work order*. Each work order is for a specific number of automobiles of a certain model. When a work order is created, the bill of materials for the corresponding type of automobile is retrieved and the required parts are taken out of inventory. As automobiles move through the assembly line, the work order status is updated to reflect progress. Once the work order is complete, it is marked as completed and inventory is updated. When 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, which is responsible for interactions with external suppliers.

2.2 SPECjAppServer2004 Application Design

All the activities and processes in the five domains described above are implemented using J2EE components (Enterprise Java Beans, Servlets and Java Server Pages) assembled into a single J2EE application that is deployed in an application server running on the *System Under Test (SUT)*. The only exception is for the interactions with suppliers which are implemented using a separate Web application called *Supplier Emulator*. The latter is deployed in a Java-enabled Web server on a dedicated machine. The supplier emulator provides the supplier domain with a way to emulate the process of sending and receiving purchase orders to/from suppliers.

The workload generator is implemented using a multi-threaded Java application called *SPECjAppServer 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. The driver is made of two components - *Manufacturing Driver* and *DealerEntry Driver*. The manufacturing driver drives the production lines (planned lines and large order lines) in the manufacturing domain and exercises the manufacturing application. It communicates with the SUT through the RMI (Remote Method Invocation) interface. The DealerEntry driver emulates automobile dealers that use the dealer application in the dealer domain to access the services of the order entry application in the customer domain. It communicates with the SUT through HTTP and exercises the dealer and order entry applications using three operations referred to as *business transactions*:

1. Browse - browses through the vehicle catalogue
2. Purchase - places orders for new vehicles
3. Manage - manages the customer inventory (sells vehicles and/or cancels open orders)

Each business transaction emulates a specific type of client session comprising multiple round-trips to the server. For example, the browse transaction navigates to the vehicle catalogue Web page and then pages a total of thirteen times, ten forward and three backwards.

A relational database management system (DBMS) is used for data persistence and all data access operations use entity beans which are mapped to tables in the SPECjAppServer database. Data access components follow the guidelines in [KB02] to provide maximum scalability and performance.

Communication across domains in SPECjAppServer2004 is implemented using asynchronous messaging exploiting the Java Messaging Service (JMS) and Message Driven Beans (MDBs). In particular, the placement and fulfillment of large orders (LOs), requiring communication between the customer domain and the manufacturing domain, is implemented asynchronously. Another example is the placement and delivery of supplier purchase orders, which requires communication between the manufacturing domain and the supplier domain. The latter is implemented according to the proposal in [KB02] to address performance and reliability issues.

The throughput of the benchmark is driven by the activity of the dealer and manufacturing applications. The throughput of both applications is directly related to the chosen *Transaction Injection Rate*. The latter determines the number of business transactions generated by the DealerEntry driver, and the number of work orders scheduled by the manufacturing driver per unit of time. The summarized performance metric provided after running the benchmark is called **JOPS** and it denotes the average number of successful **J**AppServer **O**perations **P**er **S**econd completed during the measurement interval.

3 Experimental Setting

In our experimental analysis, we use two different deployment environments for SPECjAppServer2004, depicted on Figures 3 and 4, respectively. The first one is a single-node deployment, while the second one is a clustered deployment with four JBoss servers. Table 1 provides some details on the configuration of the machines used in the two deployment environments. Since JBoss exhibits different behavior in clustered environment, the same deployment option (or tuning parameter) might have different effect on performance when used in the clustered deployment, as opposed to the single-node deployment. Therefore, we consider both deployment environments in our analysis.

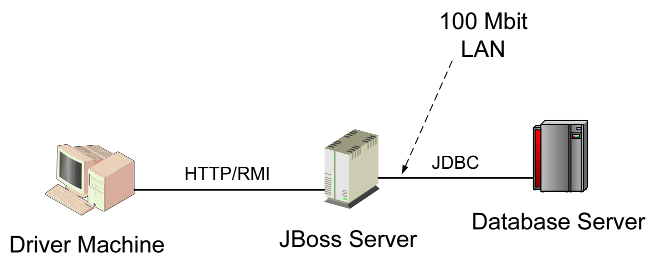

Figure 3: Single-Node Deployment

JBoss is shipped with three standard server configurations: *'minimal'*, *'default'* and *'all'*. The 'default' configuration is typically used in single-server environments,

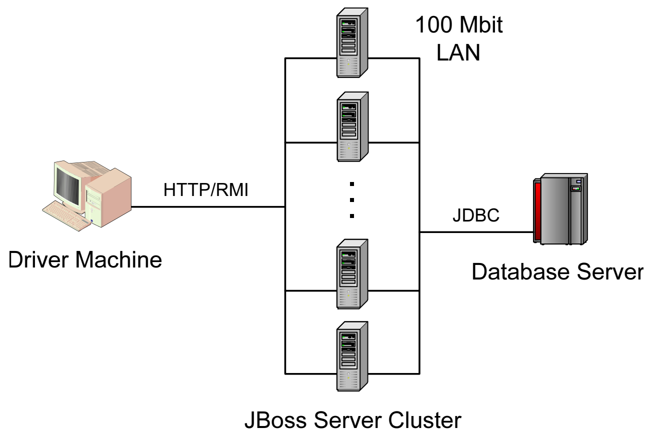

Figure 4: Clustered Deployment

Table 1: Deployment Environment Details

Node	Description
Driver Machine	SPECjAppServer Driver & Supplier Emulator 2 x AMD XP2000+ CPU 2 GB, SuSE Linux 8
Single JBoss Server	JBoss 3.2.3 Server 2 x AMD XP2000+ CPU 2 GB, SuSE Linux 8
JBoss Cluster Nodes	JBoss 3.2.3 Server 1 x AMD XP2000+ CPU 1 GB, SuSE Linux 8
Database Server	Popular commercial DBMS 2 x AMD XP2000+ CPU 2 GB, SuSE Linux 8

while the 'all' configuration is meant for clustered environments. We use the 'default' configuration as a basis for the single JBoss server in our single-node deployment, and the 'all' configuration as a basis for the N JBoss servers in our clustered deployment. For details on the changes made to the standard server configurations for deploying SPECjAppServer2004, the reader is referred to [Wei04].

The driver machine hosts the SPECjAppServer2004 driver and the supplier emulator. All entity beans are persisted in the database. The DBMS we use runs under SQL isolation level of READ_COMMITTED by default. For entity beans required to run under REPEATABLE_READ isolation level, pessimistic SELECT_FOR_UPDATE locking is used. This is achieved by setting the *row-locking* option in the `jbosscmp-jdbc.xml` configuration file.

We adhere to the *SPECjAppServer2004 Run Rules* for most of the experiments in our study. However, since not all deployment options that we consider are allowed by the Run Rules, in some cases we have to slightly deviate from the latter. For example, when evaluating the performance of different entity bean commit options, in some cases we assume that the JBoss server has exclusive access to the underlying persistent store (storing entity bean data), which is disallowed by the Run Rules. This is acceptable, since our aim is to evaluate the impact of the respective deployment options on performance, rather than to produce standard benchmark results to be published and compared with other results.

In both the single-node and the clustered deployment, all SPECjAppServer2004 components (EJBs, servlets, JSPs) are deployed on all JBoss servers. In the clustered deployment, client requests are evenly distributed over the JBoss servers in the cluster. For RMI requests (from the manufacturing driver), load-balancing is done automatically by JBoss. Unfortunately, this is not the case for HTTP requests, since JBoss is shipped without a load balancer for HTTP traffic. Therefore, we had to modify the DealerEntry driver to evenly distribute HTTP requests over the cluster nodes. Although we could have alternatively used a third-party load balancer, we preferred not to do this, since its performance would have affected our analysis whose main focus is on JBoss.

Another problem we encountered, was that access to the *ItemEnt* entity bean was causing numerous deadlock exceptions. The ItemEnt bean represents items in the vehicle catalogue and is accessed very frequently. However, it was strange that it was causing deadlocks, since the bean is only read and never updated by the benchmark. Declaring the bean as *read-only* alleviated the problem. After additionally configuring it to use JBoss' Instance-Per-Transaction Policy, the problem was completely resolved. The Instance-Per-Transaction Policy allows multiple instances of an entity bean to be active at the same time [Sco03]. For each transaction a separate instance is allocated and therefore there is no need for transaction based locking.

4 Performance Analysis

We now present the results from our experimental analysis. We look at a number of different JBoss deployment and configuration options and evaluate their impact on the overall system performance. As a basis for comparison a standard out-of-the-box configuration is used with all deployment parameters set to their default values. Hereafter, we refer to this configuration as *Standard* (shortened *"Std"*). For each deployment/configuration setting considered, its performance is compared against the performance of the standard configuration. Performance is measured in terms of the following metrics:

- CPU utilization of the JBoss server(s) and the database server
- Throughput of business transactions
- Mean response times of business transactions

By business transactions, here, we mean the three dealer operations, Purchase, Manage and Browse (as defined in section 2.2) and the WorkOrder transaction running in the manufacturing domain.

It is important to note that the injection rate at which experiments in the single-node environment are conducted, is different from the injection rate for experiments in the clustered environment. A higher injection rate is used for cluster experiments, so that the four JBoss servers are utilized to a reasonable level. Disclosing the exact injection rates at which experiments are run, is not allowed by the SPECjAppServer2004 license agreement.

4.1 Use of Different Web Containers

JBoss allows a third-party Web container to be integrated into the application server framework. The most popular Web containers typically used are Tomcat [Apa] and Jetty [Mor]. By default Tomcat 4.1 is used. As of the time of writing, the integration of Tomcat 5 in JBoss is still in its beta stage. Therefore when using it, numerous debug messages are output to the console and logged to files. This accounts for significant overhead that would not be incurred in production deployments. For this reason, we consider two Tomcat 5 configurations, the first one out-of-the-box and the second one with debugging turned off. It is the latter that is more representative and the former is only included to show the overhead of debugging.

Since the manufacturing application does not exercise the Web container, it is not run in the experiments of this section. Only the dealer application and the order-entry application are run, so that the stress is put on the benchmark components that exercise the Web container.

We consider four different configurations:

1. Tomcat 4.1 (shortened *Tom4)*
2. Tomcat 5 out-of-the-box (shortened *Tom5*)
3. Tomcat 5 without debugging (shortened *Tom5WD*)
4. Jetty

4.1.1 Analysis in Single-node Environment

Comparing the four Web containers in the single-node deployment, revealed no significant difference with respect to achieved transaction throughput and average CPU utilization. With exception of Tom5WD, in all configurations, the measured CPU utilization was about 90% for the JBoss server and 45% for the database server. The Tom5WD configuration exhibited 2% lower CPU utilization both for the JBoss server and the database server. As we can see from Figure 6, the lower CPU utilization resulted in Tom5WD achieving the best response times, followed by Jetty. It stands out that the response time improvement was most significant for the Browse transaction. The reason for this is that, while Purchase and Manage comprise only 5 round-trips to the server, Browse comprises a total of 17 round-trips each going through the Web container. As mentioned, the effect on transaction throughput was negligible. This was expected since, for a given injection rate, SPECjAppServer2004 has a target throughput that is normally achieved unless there are some system bottlenecks.

Figure 5: Legend for diagrams on Figures 6 and 7

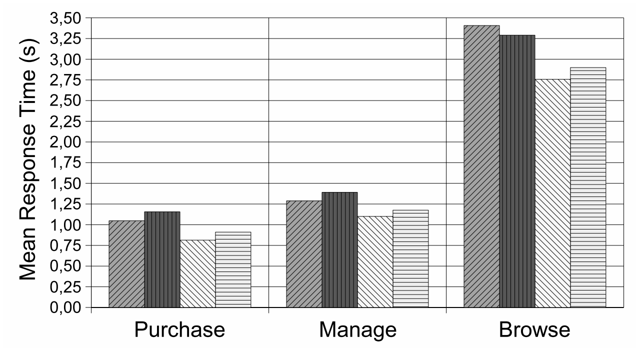

Figure 6: Mean response times with different Web containers in the single-node environment

4.1.2 Analysis in Clustered Environment

The four Web containers exhibited similar behavior in the clustered deployment. The only exception was for the Tom5 configuration, which in this case was performing much worse compared to the other configurations. The reason for this was that, all four servers in the clustered deployment were logging their debug messages to the same network drive. Since, having four servers, means four times more debug information to be logged, the shared logging drive turned into a bottleneck. Figure 7 shows the response times of the three business transactions. Note that this diagram uses a different scale.

4.2 Use of Local vs. Remote Interfaces

In SPECjAppServer2004, by default, remote interfaces are used to access business logic components (EJBs) from the presentation layer (Servlets/JSPs) of the application. However, since in both the single-node and clustered environments, presentation components are co-located with business logic components, one can alternatively use local interfaces. This eliminates the

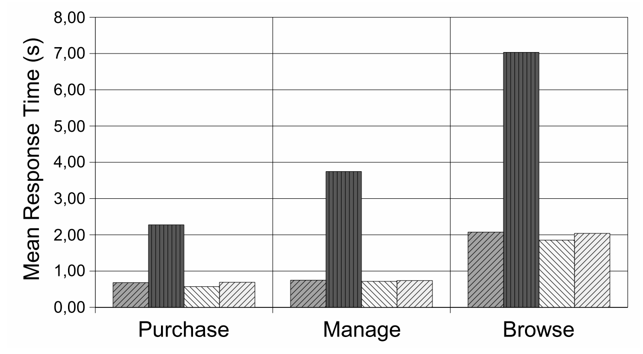

Figure 7: Mean response times with different Web containers in the clustered environment

overhead of remote network communication and is expected to improve performance. In this section, we evaluate the performance gains from using local interfaces to access EJB components from Servlets and JSPs in SPECjAppServer2004. Note that our standard configuration (i.e. Std) uses remote interfaces.

4.2.1 Analysis in Single-node Environment

Figure 9 shows the transaction response times with remote vs. local interfaces in the single-node deployment. As we can see, using local interfaces led to response times dropping by up to 35%. Again, most affected was the Browse transaction. In addition to this, the use of local interfaces led to lower CPU utilization of the JBoss server. It dropped from 82% to 73%, when switching from remote to local interfaces. Again, differences in transaction throughputs were negligible.

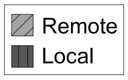

Figure 8: Legend for diagrams on Figures 9 and 10

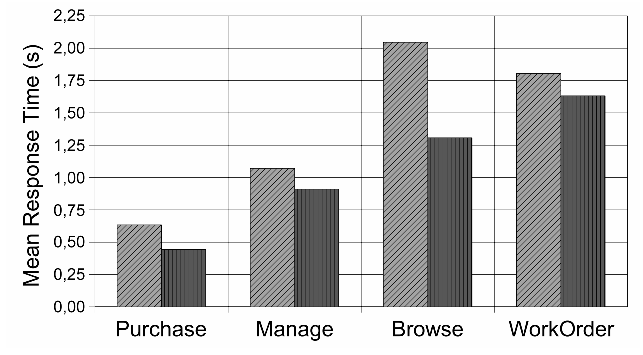

Figure 9: Mean response times with remote vs. local interfaces in the single-node environment

4.2.2 Analysis in Clustered Environment

As expected, switching to local interfaces brought performance gains also in the clustered deployment. However, in this case, the delays resulting from calls to the EJB layer were small compared to the overall response times. This is because in clustered environment, there is additional load balancing and synchronization overhead which contributes to the total response times. As a result, delays from calls to the EJB layer constitute smaller portion of the overall response times than in the single-node case. Therefore, the performance improvement from using local interfaces was also smaller than in the single-node case. Figure 10 shows the measured response times of business transactions. The effect on transaction throughput and CPU utilization was negligible.

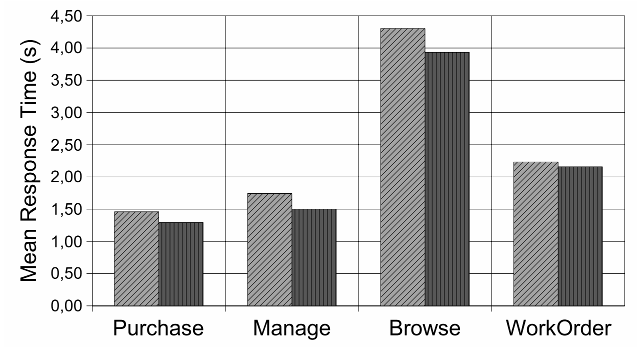

Figure 10: Mean response times with remote vs. local interfaces in the clustered environment

4.3 Data Access Optimizations

In this section, we measure the effect of several data access configuration options on the overall system performance. The latter are often exploited in JBoss applications to tune and optimize the way entity beans are persisted. We first discuss these options and then present the results from our analysis.

4.3.1 Description of Considered Optimizations

Entity Bean Commit Options: JBoss offers four entity bean persistent storage commit options, i.e. A, B, C and D [Sco03, BR01]. While the first three are defined in the EJB specification [Sun02], the last one is a JBoss-specific feature. Below we quickly summarize the four commit options:

- Commit Option A - the container caches entity bean state between transactions. This option assumes that the container has exclusive access to the persistent store and therefore it doesn't need to synchronize the in-memory bean state from the persistent store at the beginning of each transaction.
- Commit Option B - the container caches entity bean state between transactions, however unlike option A, the container is not assumed to have exclusive access to the persistent store. Therefore, the container has to synchronize the in-memory entity

bean state at the beginning of each transaction. Thus, business methods executing in a transaction context don't see much benefit from the container caching the bean, whereas business methods executing outside a transaction context can take advantage of cached bean data.

- Commit Option C - the container does not cache bean instances.
- Commit Option D - bean state is cached between transactions as with option A, but the state is periodically synchronized with the persistent store.

Note that the standard configuration (i.e. Std) uses commit option B for all entity beans. We consider two modified configurations exploiting commit options A and C, respectively. In the first configuration (called *CoOpA*), commit option A is used. While in the single-node deployment commit option A can be configured for all SPECjAppServer2004 entity beans, doing so in the clustered deployment, would introduce potential data inconsistency. The problem is that changes to entity data in different nodes of the cluster are normally not synchronized. Therefore, in the clustered deployment, commit option A is only used for the beans which are never modified by the benchmark (the read-only beans), i.e. AssemblyEnt, BomEnt, ComponentEnt, PartEnt, SupplierEnt, SupplierCompEnt and POEnt.

The second configuration that we consider (called *CoOpC*), uses commit option C for all SPECjAppServer2004 entity beans.

Entity-Bean-With-Cache-Invalidation Option: We mentioned that using commit option A in clustered environment may introduce potential data inconsistency. This is because each server in the cluster would assume that it has exclusive access to the persistent store and cache entity bean state between transactions. Thus, when two servers update an entity at the same time, the changes of one of them could be lost. To address this problem, JBoss provides the so-called *cache invalidation framework* [Sac03]. The latter allows one to link the entity caches of servers in the cluster, so that when an entity is modified, all servers who have a cached copy of the entity are forced to invalidate it and reload it at the beginning of next transaction. JBoss provides the so-called "Standard CMP 2.x EntityBean with cache invalidation" option for entities that should use this cache invalidation mechanism [Sac03]. In our analysis, we consider a configuration (called *EnBeCaIn*), which exploits this option for SPECjAppServer2004' entity beans. Unfortunately, in the clustered deployment, it was not possible to configure all entity beans with cache invalidation, since doing so led to numerous rollback exceptions being thrown when running the benchmark. The latter appears to be due to a bug in the cache invalidation mechanism. Therefore, we could only apply the cache invalidation mechanism to the read-only beans, i.e. AssemblyEnt, BomEnt, ComponentEnt, PartEnt, SupplierEnt, SupplierCompEnt and POEnt. Since, read-only beans are never modified, this should be equivalent to simply using commit option A without cache invalidation. However, as we will see later, results showed that there was a slight performance difference.

Instance-Per-Transaction Policy: JBoss' default locking policy allows only one instance of an entity bean to be active at a time. Unfortunately, the latter often leads to deadlock and throughput problems. To address this, JBoss provides the so-called *Instance Per Transaction Policy*, which eliminates the above requirement and allows multiple instances of an entity bean to be active at the same time [Sco03]. To achieve this, a new instance is allocated for each transaction and it is dropped when the transaction finishes. Since each transaction has its own copy of the bean, there is no need for transaction based locking.

In our analysis, we consider a configuration (called *InPeTr*), which uses the instance per transaction policy for all SPECjAppServer2004 entity beans.

No-Select-Before-Insert Optimization: JBoss provides the so-called *No-Select-Before-Insert* entity command, which aims to optimize entity bean create operations [Sco03]. Normally, when an entity bean is created, JBoss first checks to make sure that no entity bean with the same primary key exists. When using the No-Select-Before-Insert option, this check is skipped. Since, in SPECjAppServer2004 all primary keys issued are guaranteed to be unique, there is no need to perform the check for duplicate keys. To evaluate the performance gains from this optimization, we consider a configuration (called *NoSeBeIn*), which uses the No-Select-Before-Insert option for all SPECjAppServer2004 entity beans.

Sync-On-Commit-Only Optimization: Another optimization typically used is the so-called *Sync-On-Commit-Only* container configuration option. It causes JBoss to synchronize changes to entity beans with the persistent store, *only* at commit time. Normally, dirty entities are synchronized whenever a finder is called. When using Sync-On-Commit-Only, synchronization is not done when finders are called, however it is still done after deletes/removes, to ensure that cascade deletes work correctly. We consider a configuration called *SyCoOnly*, in which Sync-On-Commit-Only is used for all SPECjAppServer2004 entity beans.

Prepared Statement Cache: In JBoss, by default, prepared statements are not cached. To improve performance one can configure a prepared statement cache of an arbitrary size [Sco03]. We consider a configuration called *PrStCa*, in which a prepared statement cache of size 100 is used.

In summary, we are going to compare the following configurations against the standard (Std) configuration:

1. Commit Option A (CoOpA)
2. Commit Option C (CoOpC)
3. Entity Beans With Cache Invalidation (EnBeCaIn)
4. Instance Per Transaction Policy (InPeTr)
5. No-Select-Before-Insert (NoSeBeIn)
6. Sync-On-Commit-Only (SyCoOnly)
7. Prepared Statement Cache (PrStCa)

4.3.2 Analysis in Single-node Environment

Figure 12 shows the average CPU utilization of the JBoss server and the database server under the different configurations in the single-node environment. Figure 13 shows the response times of business transactions. All configurations achieved pretty much the same transaction throughputs with negligible differences.

As we can see, apart from the three configurations CoOpA, EnBeCaIn and PrStCa, all other configurations had similar performance. As expected, configurations CoOpA and EnBeCaIn had identical performance, since in a single-node environment there is practically no difference between them. Caching entity bean state with commit option A, resulted in 30 to 60 percent faster response times. Moreover, the CPU utilization dropped by 20% both for the JBoss server and the database server. The performance gains from using a prepared statement cache were even greater, i.e. the database utilization dropped from 47% to only 18%!

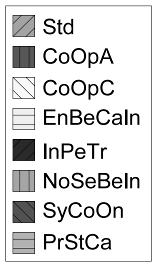

Figure 11: Legend for diagrams on Figures 12, 13, 14 and 15

4.3.3 Analysis in Clustered Environment

Figure 14 shows the average CPU utilization of the JBoss server and the database server under the different configurations in clustered environment. Figure 15 shows the response times of business transactions. As

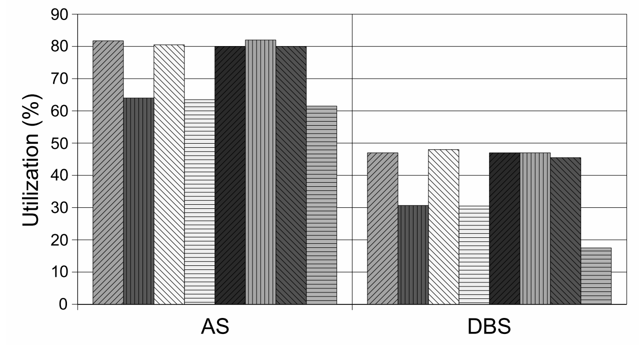

Figure 12: CPU utilization under different configurations in the single-node environment

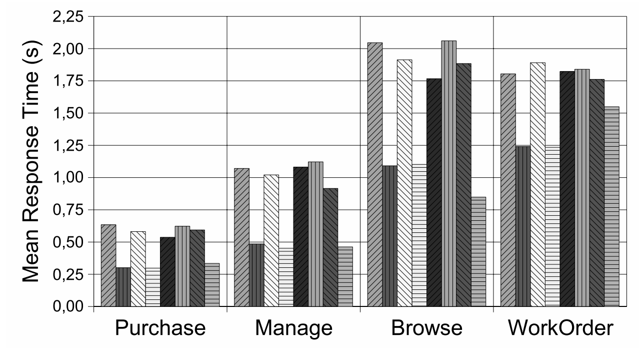

Figure 13: Mean response times under different configurations in the clustered environment

usual, all configurations achieved pretty much the same transaction throughputs with negligible differences.

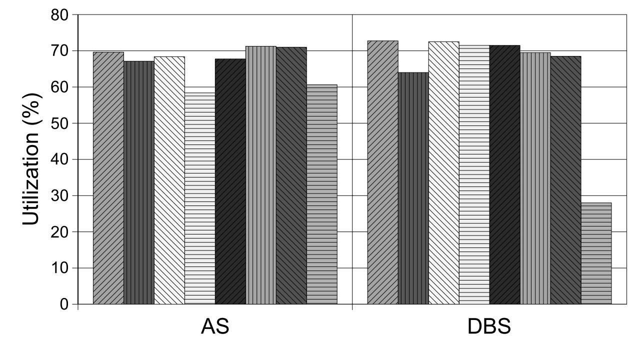

Figure 14: CPU utilization under different configurations in the clustered environment

As we can see, the performance gains from caching entity bean state with commit option A (configurations CoOpA and EnBeCaIn), were not as big as in the single-node deployment. This was expected since, as discussed earlier, in this case only the read-only beans could be cached. It came as a surprise, however, that there was a difference between simply using commit option A for read-only beans (CoOpA), as opposed to using it with the cache invalidation option (EnBeCaIn). We didn't expect to see a difference here, since cached beans, being read-only, were never invalidated. There

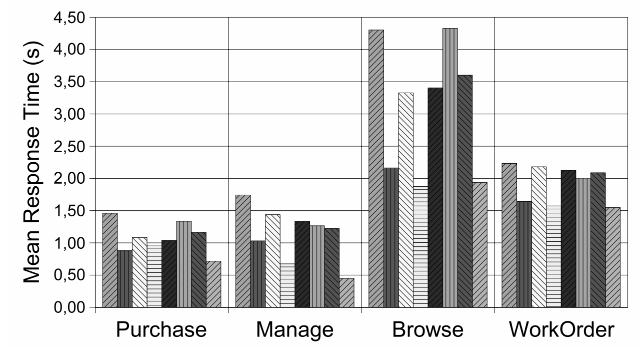

Figure 15: Mean response times under different configurations in the clustered environment

was a slight difference, however, not just in CPU utilization, but also in transaction response times, which in most cases were slightly better for EnBeCaIn. Again, the performance gains from having a prepared statement cache were considerable, i.e. the database utilization dropped from 78% to only 27%!

4.4 Use of Different JVMs

In this section, we demonstrate that the underlying JVM has a very significant impact on the performance of JBoss applications. We compare the overall performance of our SPECjAppServer2004 deployments under two different popular JVMs. Unfortunately, we cannot disclose the names of these JVMs, as this is prohibited by their respective license agreements. We will instead refer to them as "JVM A" and "JVM B". The goal is to demonstrate that changing the JVM on which JBoss is running, may bring huge performance gains and therefore it is worth considering this in real-life deployments.

Figures 17 and 19 show the average CPU utilization of the JBoss server and the database server under the two JVMs in the single-node and clustered environment, respectively. Response times are shown in Figures 18 and 20. As evident, in both environments, JBoss runs much faster with the second JVM: response times are up to 55% faster and the CPU utilization of the JBoss server is much lower.

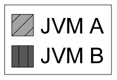

Figure 16: Legend for diagrams on Figures 17, 18, 19 and 20

5 JBoss Behavior under Heavy Load

While JBoss performance can be boosted significantly by tuning configuration parameters and optimizing the deployment environment, our experiments under heavy load, showed that there is still a lot to be desired as far as reliability and scalability are concerned. Trying to

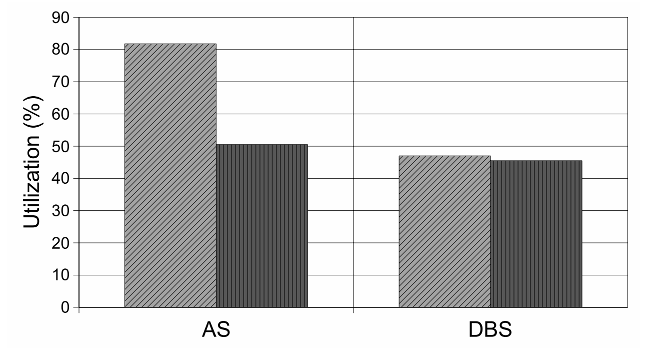

Figure 17: CPU utilization under the two JVMs in the single-node environment

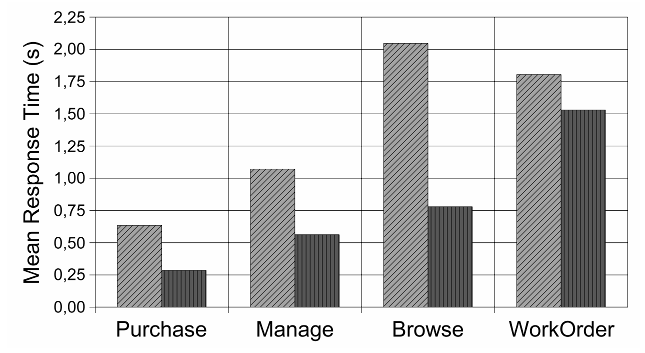

Figure 18: Mean response times under the two JVMs in the single-node environment

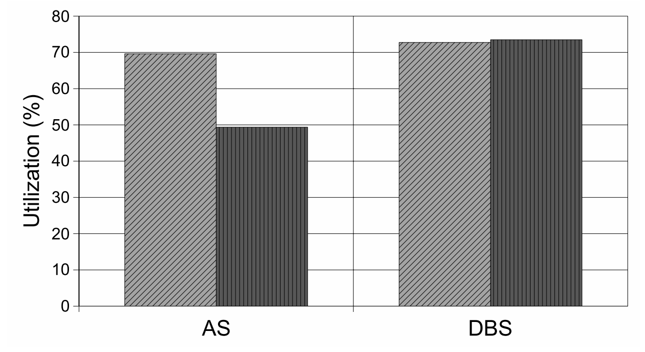

Figure 19: CPU utilization under the two JVMs in the clustered environment

raise the injection rate, so that JBoss server is close to saturated, led to numerous transaction aborts and system exceptions (JBoss-specific), being thrown by the container. This was happening both in the single-node and clustered environments. Moreover, in the clustered environment, adding additional servers would not resolve the problem. For these reasons, we were not able to scale out our deployment. Note that testing with a popular commercial application server in the same environment did not lead to any scalability problems.

Another problem was the Web container which proved to be a serious bottleneck. This was observed when steadily raising the injection rate and monitor-

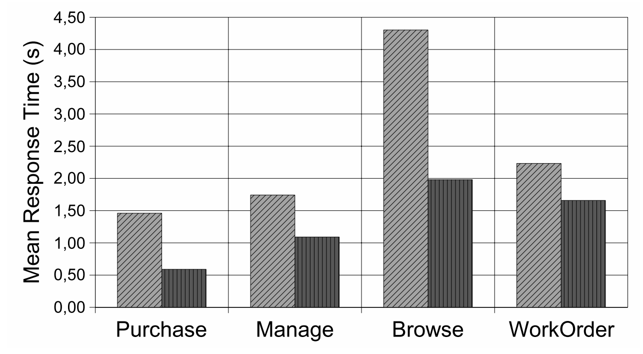

Figure 20: Mean response times under the two JVMs in the clustered environment

ing the throughput of the dealer and manufacturing domains. While the throughput of the manufacturing domain was increasing with the injection rate, the throughput of the dealer domain quickly reached its peak point and started to drop.

6 Summary and Conclusions

In this paper, we studied how the performance of JBoss applications can be improved by exploiting different deployment and configuration options offered by the JBoss platform. We used the industry standard SPECjAppServer2004 benchmark as a basis for our experimental analysis.

Comparing the three alternative Web containers Tomcat 4.1, Tomcat 5 and Jetty, revealed no significant performance differences. Tomcat 5 (with debugging turned off) and Jetty, turned out to be slightly faster than Tomcat 4.1. On the other hand, using local interfaces to access business logic components from the presentation layer, brought significant performance gains. Indeed, response times dropped by up to 35% and the utilization of JBoss server dropped by 10%. Caching entity beans with commit option A (with and without cache invalidation), resulted in 30 to 60 percent faster response times and 20% lower CPU utilization, both for the JBoss server and the database server. The performance gains from caching prepared statements were even greater. The other data access optimizations considered, led to only minor performance improvements. However, we must note here that, being an application server benchmark, SPECjAppServer2004 places the emphasis on the application server (transaction processing) rather than the database. We expect the performance gains from data access optimizations to be more significant, in applications where the database is the bottleneck. Furthermore, in our analysis we considered optimizations one at a time. If all of them were to be exploited at the same time, their cumulative effect would obviously lead to more significant performance improvements. Finally, we demonstrated that switching to a different JVM may improve JBoss performance by up to 60%.

On the negative side, our experience with JBoss showed that some important services often used in J2EE applications, were either not supported at all (for e.g. load balancing for HTTP requests) or they were supported, but trying to use them with SPECjAppServer2004 led to problems most likely due to bugs in JBoss (for e.g. the ItemEnt caching, the cache invalidation mechanism, optimistic locking with "modified" locking strategy, etc). We hope that these issues, as well as the scalability issues mentioned earlier, will be addressed in the next version of JBoss.

Acknowledgments

We acknowledge the many fruitful discussions with our colleague Christofer Dutz from Darmstadt University of Technology.

References

[Apa] Apache Software Foundation. Jakarta Tomcat. `http://jakarta.apache.org/tomcat/`.

[BR01] P. Brebner and S. Ran. Entity Bean A, B, C's: Enterprise Java Beans Commit Options and Caching. In *Proc. of IFIP/ACM Intl Conf. on Distr. Systems Platforms - Middleware*, 2001.

[KB02] S. Kounev and A. Buchmann. Improving Data Access of J2EE Applications by Exploiting Asynchronous Processing and Caching Services. In *Proc. of the 28th Intl Conference on Very Large Data Bases - VLDB2002*, 2002.

[Mor] Mort Bay Consulting. Jetty Java HTTP Servlet Server. `http://jetty.mortbay.org`.

[Sac03] Sacha Labourey and Bill Burke. *JBoss Clustering*. The JBoss Group, 2520 Sharondale Dr., Atlanta, GA 30305 USA, 6th edition, 2003.

[Sco03] Scott Stark. *JBoss Administration and Development*. The JBoss Group, 2520 Sharondale Dr., Atlanta, GA 30305 USA, 3th edition, 2003.

[Sta04] Standard Performance Evaluation Corporation (SPEC). SPECjAppServer2004 Documentation. Specification, Apr 2004. `http://www.spec.org/jAppServer2004`.

[Sun02] Sun Microsystems, Inc. Enterprise JavaBeans 2.0. Specification, 2002. `http://java.sun.com/products/ejb/`.

[Wei04] Bjoern Weis. Performance Optimierung von J2EE Anwendungen auf der JBoss Plattform. Master thesis, Darmstadt University of Technology, March 2004. In German.