# Java Application Server Optimization for Multi-core Systems

Intel Corporation: Khun Ban, Kingsum Chow, Yong-Fong Lee  
Oracle Corporation: Robert Butera, Staffan Friberg, Evan Peers

## Table of Contents

1. **INTRODUCTION** .................................................................................................................. 2  
2. **PERFORMANCE ANALYSIS METHODOLOGY** ................................................................. 3  
   2.1 **ANALYSIS TOOLS** ........................................................................................................ 3  
   2.2 **TEST CONFIGURATION** ................................................................................................ 4  
3. **SYSTEM TUNINGS** ............................................................................................................. 6  
   3.1 **USING MULTIPLE JVMs** ............................................................................................... 7  
   3.1.1 **USING 64-bit** ............................................................................................................ 10  
   3.2 **Binding JVMs to a NUMA node** .................................................................................... 11  
   3.3 **Use Solid-State Drives (SSD) to Reduce IOWait Time** .................................................. 11  
   3.4 **Large Pages** .................................................................................................................. 12  
4. **JVM TUNING** ..................................................................................................................... 12  
   4.1 **64-bit JVM** .................................................................................................................... 12  
   4.2 **Aggressive optimization** .............................................................................................. 12  
5. **WEBLOGIC SERVER TUNING** .......................................................................................... 13  
   5.1 **JDBC tuning** .................................................................................................................. 13  
   5.1.1 **Using Logging Last Resource Optimization** ................................................................. 13  
   5.1.2 **Using the Prepared Statement Cache** ....................................................................... 13  
   5.1.3 **Setting Initial and Max Capacity** .............................................................................. 14  
   5.1.4 **Using Pinned-To-Thread** ......................................................................................... 14  
   5.2 **EJB tuning** ..................................................................................................................... 14  
   5.2.1 **Using Read-only Concurrency Strategy** .............................................................. 14  
   5.2.2 **Using Optimistic Concurrency Strategy** ............................................................. 14  
   5.2.3 **Using eager-relationship-caching** ........................................................................... 15  
   5.2.4 **EJB caching** .............................................................................................................. 15  
   5.3 **SocketMuxer tuning** ..................................................................................................... 15  
   5.3.1 **Changing the Socket Muxer** .................................................................................... 15  
   5.3.2 **Change # of socket reader threads** ......................................................................... 16  
6. **SUMMARY** ......................................................................................................................... 16  
7. **REFERENCES** .................................................................................................................... 16
ABSTRACT
Recent advances in computer architectures have taken multiple paths. One of them is the move towards using more cores on a microprocessor. However, Java application server performance does not automatically benefit from the extra processing power from the added cores. This paper examines the performance characteristics of Java application servers running on 32-bit and 64-bit Java Virtual Machines (JVM) and operating systems on the latest architectures and platforms available today. It exposes the need to address platform performance for Java applications running on the latest multi-core servers. The performance characteristics covered at the Java application, the JVM and the operating system layers provide helpful insights for software developers need to deploy application servers and tune for better performance to close the gap between out-of-the-box performance and the best performance achievable on modern multi-core microprocessors.

1. INTRODUCTION
Enterprise applications are widely popular and their performance and scalability are becoming even more critical as the number of online transactions and CPU cores continue to increase over time. Many different frameworks and application servers are available for implementing these enterprise applications and one of the most common ones are Java application servers. Enterprise applications are often multi-tier and usually consist of a middle tier where the applications are deployed and a database backend tier where data are stored and retrieved to perform business functions. While every tier plays an important role in performance, the performance of middle tier is particularly important as it is often responsible for performing complex business logic. In this paper, we examine the performance characteristics of some common configurations available today.

We used the popular SPEC application server benchmark called SPECjAppServer2004 as the workload for our performance measurements. As described on the SPEC web site, “SPECjAppServer2004 is a J2EE™ benchmark to measure the scalability and performance of J2EE servers and containers and is based on the J2EE 1.3 Specification. SPECjAppServer2004 strives to stress the middle-tier rather than the client tier or the database server tier. The SPECjAppServer2004 workload emulates an automobile manufacturing company and its associated dealerships. Dealers interact with the system using web browsers (simulated by the driver) while the actual manufacturing process is accomplished via RMI (also driven by the driver).” Here are some key characteristics of this workload:

- SPECjAppServer2004 is a J2EE application. Figure 1 depicts the software/hardware stack of the configurations we used.
- SPECjAppServer2004 emulates a just-in-time manufacturing process, very similar to many deployments used for business to business arrangements.
- The connections from the drivers to the application server are mainly HTTP, with a small number of RMI connections.
- The connections from the application server to the database use standard JDBC connections.

For the JVM and J2EE containers, we chose to use Oracle JRockit and Oracle WebLogic Server. The driver software for SPECjAppServer2004 is written in Java and is part of the benchmark kit distributed by SPEC. As for the hardware platform, we used the latest dual processors system with Quad-Core Intel® Xeon® processor X5570, code named Nehalem.

At the start of our study, we used the best known configuration settings that we believe were widely in use in real world J2EE deployments, and took a baseline performance measurement. From the performance data we collected for the baseline, we
discovered that the mid tier system was greatly underutilized. The CPUs were less than 50% used. After we applied all the tunings discussed in this white paper, the performance increased more than 100% from the baseline.

2. PERFORMANCE ANALYSIS METHODOLOGY

Performance analysis is a difficult task to master, as it is most often an afterthought. Many quality assurance engineers only test for functionality at low load and miss the performance issues at higher load. During higher load, performance issues can also become functionality issues, such as keeping users from accessing valuable information due to excessive response time. Once it happens, we must identify the real cause of the problem instead of the symptom. Only then can we identify possible solutions.

Figure 2 illustrates the closed loop process that we used to tune our configuration. The whole process starts with collecting good performance data for the baseline. This could be with lower load as we are concentrating on two things, the functionality of the setup and the system behavior. After we have the baseline data, we are ready to begin our performance tests, such as doubling the load level. The tests are most useful when we have good performance data to correlate with the test. From the performance data collected, we can identify the bottlenecks that may have caused issues such as slow response times or failed transactions by comparing the performance data of the current test against the baseline.

Figure 2. General performance analysis methodology

If we are able to correctly identify the root cause of the bottlenecks, we can then identify some possible solutions. We apply one of our proposed solutions and then we restart the tuning cycle by rerunning the test and collecting data again.

2.1 ANALYSIS TOOLS

No optimization or tuning should be done blindly! The simplest check that can be done is that some performance metric of your application shows improvement after applying a proposed solution. However this does not give any insights in to how your application, the JVM, the operating system, or the platform itself is behaving internally. Oracle JRockit Mission Control (JRMC) has been extensively used throughout the processes of improving the performance of both Oracle WebLogic and JRockit on Nehalem. JRMC is able to provide in-depth information about the application showing both which methods are the most executed and from where they are called, to fine grained latency information about which locks are contended and how long network transactions take. On the JVM level it provides information about garbage collection, heap fragmentation and even information about how contended internal JVM data structures are.
Information gathered with JRMC has enabled us to find scalability issues as the number of cores increase and allowed us focus on the actual bottlenecks and minimize effort wasted on wrong areas due to blind guessing. JRMC is available for download here, http://www.oracle.com/technology/software/products/jrockit/index.html, with the documentation available here, http://download.oracle.com/docs/cd/E13188_01/jrockit/tools/index.html.

For the operation system level information, we used the SAR utility that came with Linux. With SAR, we can collect information on CPU usage, network saturation for the individual network interface card, disk IO, context switches, and many more counters. Please refer to the SAR help pages for more information.

JRMC and SAR are great tools to help us gain better understanding at the JVM and the operating system level. For the platform and the CPU level information, we used the Intel VTune™ Performance Analyzer. With VTune, we can find out CPU cache misses and which module caused the cache misses, system bus and memory latency, and which memory controller are most heavily used. For more information, please refer to http://www.intel.com/software/products/vtune.

2.2 TEST CONFIGURATION

Enterprise applications, including Java application server workloads, are often multi-layered. In other words, many layers of software are involved in executing Java application server workloads such as the operating system, JVM, libraries, various J2EE components, and the application code itself, as shown in Figure 3. This implies that in order to characterize the performance of Java application server workloads, it is necessary to understand the performance characteristics of all these layers.

While using the same hardware, we obtained performance data for the following four configurations for comparing and contrasting the characteristics of different software stacks:

- **32-bit 1-J2EE**: this configuration uses a 32-bit OS and a 32-bit JVM to run the workload
- **32-bit 2-J2EE**: this configuration uses a 32-bit OS and two 32-bit JVMs to run the workload. Each JVM instance is running identical applications but listening to different network addresses.
- **64-bit 1-J2EE**: this configuration uses a 64-bit OS and a 64-bit JVM to run the workload.
- **64-bit 2-J2EE**: this configuration uses a 64-bit OS and two 64-bit JVMs to run the workload. Each JVM instance is running identical applications but listening to different network addresses.

Table 1 shows the major hardware and software components for our four configurations. This configuration used the latest Quad-Core Intel Xeon processor X5570 and Oracle WebLogic Server version 10.3.1; both are commercially available and widely used. We chose to use Enterprise Linux* 5 Update 3, as many of the SPECjAppServer2004 publication results used Linux. As for the JVM, we chose to use Oracle* JRockit for all configurations. Thus our studies have good coverage of the latest multi-core processors, the latest J2EE application server and JVM from Oracle, and the latest Linux kernel release as of this writing.

<table>
<thead>
<tr>
<th>SPECjAppServer2004</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware Configuration</strong></td>
</tr>
<tr>
<td>Quad-Core Intel Xeon processor X5570, 2.93GHz with 8MB shared L3 cache, 24 GB Memory (12 x 2 GB DDR3-1333), 2x Quad Port Intel PRO/1000 PT</td>
</tr>
<tr>
<td><strong>Software Configuration</strong></td>
</tr>
<tr>
<td>Enterprise Linux* 5 Update 3</td>
</tr>
<tr>
<td>Oracle JRockit, Oracle WebLogic Server version 10.3.1</td>
</tr>
</tbody>
</table>

We tuned the configuration for the 32-bit platform and applied the same parameters to the 64-bit platform, with the exception of increasing the heap size for the 64-bit platform. This maybe sub-optimal for the 64-bit setup, but we want to limit the impact from other factors outside the change of the software stack.
3. SYSTEM TUNINGS

Today computer systems are many times faster and more energy efficient when compared to the computer systems that are just a few years old. For example in Figure 4 we show the system diagram of the Quad-Core Intel Xeon processor X5500 series system that performed about twice as fast as the previous generation system as measured by SPECjAppServer2004, and consumed much less power. This system came with many new features, like a new memory subsystem, Intel® QuickPath Interconnect, Intel® Intelligent Power Technology, a new I/O Subsystem, Intel Turbo Boost Technology, and Hyper-Threading. When paired with software that can take advantage of these features, they yield big improvements in performance. For example, when running SPECjAppServer2004 with Oracle WebLogic Server version 10.3.1, we measured a 30-40% performance improvement with Hyper-Threading when compared to without Hyper-Threading.
Figure 5. Cores and cache layout for **Quad-Core Intel Xeon Processor X5500** Series system

Figure 5 shows this same system but focuses on the processors. Each CPU has four cores, with 2 threads per core with Hyper-Threading enabled, for a total of 16 hardware threads for a system with 2 CPU. Each CPU has up to 8 MB of shared last level cache and an integrated memory controller. From Figure 4 and Figure 5, it becomes obvious that a CPU core can access memory or cache faster if they are on the same node (socket). Therefore core1 of CPU1 will be able to perform work faster if the memory it requires resides in the shared cache of CPU1 or on the memory connected to CPU1. Because accessing memory from other nodes incurs a longer delay on this system, it is called a Non-Uniform Memory Architecture (NUMA) system. For a good description of NUMA systems, please refer to http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access.

On Linux, we can find out the NUMA status of the system using numactl with the --show option. Here is the output on the upcoming Intel Xeon processor 7500 series system code named Nehalem EX:

```
> numactl --show
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
cpubind: 0 1 2 3
nodebind: 0 1 2 3
membind: 0 1 2 3
```

### 3.1 USING MULTIPLE JVMs

In the previous section we talked about how having worker threads execute on the same node as the resource they require can improve overall performance. Here in Table 2 we compared the performance when running SPECjAppServer2004 using 1 JVM vs. 2 JVMs bound to specific cores using the Linux utility taskset, as NUMA is not available on 32-bit Linux. We started out by tuning the system for the 1 JVM configuration and ran the benchmark to as high as we could. We then reran the benchmark, this time using 2-JVM, with each JVM using half the threads as when running 1-JVM. Network interrupts were bound to individual cores for both experiments. Since we ran these experiments on 32-bit Enterprise Linux, the largest Java heap we were able to use was 2.7 GB per JVM. So for the 2 JVMs case, we are in effect doubling the heap size by having two 2.7 GB heap.

Table 2 shows the OS level system counters such as CPU usage, context switches, and run queue length for the two configurations. It also shows inbound and outbound network traffic data for the application server both in terms of packets
and bytes per second. The network traffic for this workload tracked the changes in performance well for both packets and bytes per second. We monitored the context switches to make sure that they are within the capability of the system. Different systems will have different upper limits for these counters; the levels in Table 2 were within the capability of the system based on our experience with the system running other experiments not shown here.

One of the most important functions of the JVM is garbage collection (GC), the process of reclaiming unused memory. We show GC information in Table 2 for the two configurations. GC data was taken for 30 minutes during the steady state part of the workload. Percentage of time spent in GC is based on wall clock time. By using 2 JVMs, we observed the percentage of time spent in GC dropped from 14% to 4%.

For this test we did not fully saturate the CPU. This was due to the workload’s response time requirements. When we attempted to push system to a higher load level, we were not able to meet the response time requirements. Overall we observed a 24% improvement in performance when running with 2 JVMs instead of 1 JVM. This improvement came from the reduction in the time spent in GC, along with other factors such as more efficient use of the CPU shared cache and memory that were mentioned in the previous section.

Table 2. System performance counters comparing 1 instance vs. 2 instances on 32-bit Linux

<table>
<thead>
<tr>
<th>System Counters</th>
<th>1-JVM</th>
<th>2-JVM</th>
<th>2-JVM / 1-JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Metric (Normalized to 1 JVM)</td>
<td>1.00</td>
<td>1.24</td>
<td>1.24</td>
</tr>
<tr>
<td>cswch/s</td>
<td>86,770</td>
<td>97,534</td>
<td>1.12</td>
</tr>
<tr>
<td>CPU Usage (Normalized to 1 JVM)</td>
<td>1.00</td>
<td>1.06</td>
<td>1.06</td>
</tr>
<tr>
<td>eth_rxpk/s</td>
<td>124,738</td>
<td>154,835</td>
<td>1.24</td>
</tr>
<tr>
<td>eth_txpk/s</td>
<td>234,443</td>
<td>288,903</td>
<td>1.23</td>
</tr>
<tr>
<td>eth_rxbyt/s</td>
<td>20,258,974</td>
<td>25,141,876</td>
<td>1.24</td>
</tr>
<tr>
<td>eth_txbyt/s</td>
<td>236,708,036</td>
<td>293,526,634</td>
<td>1.24</td>
</tr>
<tr>
<td>runq-sz</td>
<td>24</td>
<td>27</td>
<td>1.15</td>
</tr>
<tr>
<td>plist-sz</td>
<td>472</td>
<td>552</td>
<td>1.17</td>
</tr>
<tr>
<td>ldavg-1</td>
<td>26</td>
<td>34</td>
<td>1.28</td>
</tr>
<tr>
<td>ldavg-5</td>
<td>26</td>
<td>33</td>
<td>1.25</td>
</tr>
<tr>
<td>ldavg-15</td>
<td>24</td>
<td>29</td>
<td>1.22</td>
</tr>
<tr>
<td>Average GC Pause Time (ms)</td>
<td>361</td>
<td>247</td>
<td>0.69</td>
</tr>
<tr>
<td>Average Resident Memory Size (MB)</td>
<td>1,169</td>
<td>680</td>
<td>0.58</td>
</tr>
<tr>
<td>Average Time between GC (sec)</td>
<td>2.58</td>
<td>5.54</td>
<td>2.15</td>
</tr>
<tr>
<td>Percentage of Time Spent on GC</td>
<td>14.01%</td>
<td>4.47%</td>
<td>0.32</td>
</tr>
</tbody>
</table>

Even though the total thread counts are the same for both configurations, the 2-JVM configuration has more threads ready to run. To better understand this behavior, we captured locking data using JRMC. Table 3 shows the locks used by Java code for the 1 JVM vs. 2 JVMs, and Table 4 shows the locks used by the JVM for garbage collection activities. With this lock data, the reason for the performance improvement going from 1 JVM to 2 JVM becomes clear. By using 2 JVMs, with the same total combined thread count as the 1 JVM case, we observed a big reduction in the number of “Fat Contended” locks used by the Java code.
### Table 3. Comparing 1 instance vs. 2 instances Java locks on 32-bit Linux

<table>
<thead>
<tr>
<th>Class</th>
<th>Fat Cont.</th>
<th>Fat Uncont.</th>
<th>Fat Sleep</th>
<th>Thin Cont.</th>
<th>Thin Uncont.</th>
</tr>
</thead>
<tbody>
<tr>
<td>weblogic.aej.container.cache.NRUCache</td>
<td>873,056</td>
<td>2,962,712</td>
<td>843,340</td>
<td>2,572,838</td>
<td>48,340,737</td>
</tr>
<tr>
<td>weblogic.transaction.internal.ResourceDescriptor51</td>
<td>4,429</td>
<td>4,760</td>
<td>4,395</td>
<td>28,749</td>
<td>4,218,688</td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td>4,310</td>
<td>106,203</td>
<td>3,384</td>
<td>181,944</td>
<td>59,264,995</td>
</tr>
<tr>
<td>java.util.HashSet</td>
<td>2,182</td>
<td>96,124</td>
<td>693</td>
<td>289,630</td>
<td>21,095,935</td>
</tr>
<tr>
<td>java.util.LinkedList</td>
<td>2,106</td>
<td>6,349</td>
<td>1,928</td>
<td>1,191,666</td>
<td>22,812,985</td>
</tr>
<tr>
<td>weblogic.util.collections.StackPool</td>
<td>2,063</td>
<td>1,085</td>
<td>2,052</td>
<td>1,114,030</td>
<td>95,441,687</td>
</tr>
<tr>
<td>weblogic.jvm.MsgAbbrevVMConnectionSwritingState</td>
<td>1,110</td>
<td>10,877</td>
<td>565</td>
<td>49,590</td>
<td>6,104,240</td>
</tr>
<tr>
<td>weblogic.kernel.ServerExecuteThread</td>
<td>910</td>
<td>5,070,149</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>661</td>
<td>1,153</td>
<td>634</td>
<td>188,127</td>
<td>10,759,863</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>438</td>
<td>6,292</td>
<td>54</td>
<td>1,256,837</td>
<td>565,758,153</td>
</tr>
<tr>
<td>weblogic.messaging.kernl.internal.QueueImpl</td>
<td>284</td>
<td>1,451</td>
<td>213</td>
<td>41,063</td>
<td>750,402</td>
</tr>
<tr>
<td>weblogic.timers.internal.TimerThread</td>
<td>39</td>
<td>4,754</td>
<td>0</td>
<td>0</td>
<td>71</td>
</tr>
</tbody>
</table>

1-JVM

<table>
<thead>
<tr>
<th>Class</th>
<th>Fat Cont.</th>
<th>Fat Uncont.</th>
<th>Fat Sleep</th>
<th>Thin Cont.</th>
<th>Thin Uncont.</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.HashMap</td>
<td>991</td>
<td>57,528</td>
<td>118</td>
<td>111,006</td>
<td>14,458,878</td>
</tr>
<tr>
<td>java.util.Map</td>
<td>734</td>
<td>96,749</td>
<td>512</td>
<td>88,579</td>
<td>37,515,072</td>
</tr>
<tr>
<td>weblogic.kernel.ServerExecuteThread</td>
<td>144</td>
<td>1,620,761</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>weblogic.timers.internal.TimerThread</td>
<td>48</td>
<td>5,456</td>
<td>3</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>java.util.List</td>
<td>30</td>
<td>185</td>
<td>29</td>
<td>403,033</td>
<td>16,424,278</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>27</td>
<td>541</td>
<td>13</td>
<td>91,012</td>
<td>6,637,283</td>
</tr>
<tr>
<td>weblogic.messaging.kernl.internal.Queue...</td>
<td>6</td>
<td>97</td>
<td>0</td>
<td>11,907</td>
<td>478,099</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>377,173</td>
<td>364,839,805</td>
</tr>
<tr>
<td>weblogic.aej.container.cache.EntityCache...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6,075</td>
<td>322,935,710</td>
</tr>
<tr>
<td>weblogic.transaction.internal.ServerTrans...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>313,708,221</td>
<td></td>
</tr>
<tr>
<td>oracle.jdbc.driver.T4CConnection</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>246,621,815</td>
<td></td>
</tr>
</tbody>
</table>

2-JVM

At the same time, the number of locks used by the JVM for GC related activities also saw a big drop by moving to 2 JVMs.
Therefore on 32-bit Enterprise Linux, we will benefit from both reduction in heap pressure (twice the heap) and improvement in lock contention by moving to 2-JVMs, as each JVM has to service only half the number of threads as the 1-JVM configuration.

3.1.1 USING 64-bit

The previous section compared 1-JVM vs. 2-JVMs on 32-bit Linux, which has a limitation on heap size. But with a 64-bit JVM on 64-bit Linux, the limitation on heap size goes away. In this section, we did the same experiments but using a 64-bit JVM on 64-bit Linux and found similar improvement in performance using multiple instances. We used a 20 GB heap for the 1-JVM setup and a 10 GB heap for each of the 2-JVM setups with each JVM bound to a node using the numactl utility. As with 32-bit Linux, we saw a great reduction in lock contention. Unlike with 32-bit Linux, the performance improvement was not from a reduction in GC related activity.

### Table 5. System performance counters comparing 1 instance vs. 2 instances on 64-bit Linux

<table>
<thead>
<tr>
<th>Performance Metric (Normalized to 1 JVM)</th>
<th>1-JVM</th>
<th>2-JVM</th>
<th>2-JVM / 1-JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Usage</td>
<td>1.00</td>
<td>1.29</td>
<td>1.29</td>
</tr>
<tr>
<td>Percentage of Time Spent on GC</td>
<td>6.3%</td>
<td>4.7%</td>
<td>0.74</td>
</tr>
</tbody>
</table>

Instead the performance gains were mostly from improvement in cache and memory access locality. By using 2 bound JVMs, the percentage of time the CPU got the memory from the faster local node jumped from 49% to 98% of the time. This
translates into about a 15% reduction in the path length, or the number of CPU instructions required in performing one Java operation.

Table 6. Micro-Architecture counters comparing 1 instance vs. 2 instances on 64-bit Linux

<table>
<thead>
<tr>
<th>Micro-Architecture Counters</th>
<th>Overall User Kernel</th>
<th>Overall User Kernel</th>
<th>Overall User Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Length</td>
<td>5,880,612</td>
<td>4,992,349</td>
<td>888,264</td>
</tr>
<tr>
<td>Local DRAM Demand &amp; Prefetch Requests</td>
<td>0.489</td>
<td>0.975</td>
<td>1.99</td>
</tr>
</tbody>
</table>

3.2 Binding JVMs to a NUMA node

When running multiple JVMs on a NUMA system there is a good chance the JVM will be scheduled on multiple nodes and use memory from different nodes at the same time. How execution is scheduled and on which node memory allocations are done is controlled by the OS. If a thread is running on one node and referencing memory on a different node, that is slower than accessing memory on the node where the thread is running. If multiple processes are doing this, the interconnect, the connection between the NUMA-nodes, will become the bottleneck for all memory accesses. By using the Linux utility, `numactl`, it is possible to tell the OS how the program should be allowed to execute. Through `numactl` it is possible to lock a process to a specific node both for execution and memory allocation. If a JVM instance is locked to a single node then the inter node traffic is removed and all memory access will happen on the fast local memory. This is particularly useful when running multiple instances, since binding each instance to its own node will limit interference from both CPU and memory resources. The net result is the most optimal CPU cache and memory locality; thus best response time.

Once we find out which processor we want to bind the JVM to, we can issue the following command to start our application and bind to the node. Here we bind `myapp` to the first CPU node and restrict the memory to the memory connected to the first CPU node.

```
numactl --cpunodebind=0 --membind=0 myapp
```

The JVM has no knowledge about other JVMs or processes running on the same machine, further it does not notice if it has been bound to a limited set of the cores in the machine. In general this does not affect the tuning of the JVM; however tuning the number of GC threads can improve performance of both the JVM and the other processes running on the machine. The number of GC threads can be set by using the command line option, `-XX:GCThreads`. By default the number of parallel Garbage Collection threads will be the same as the number of cores in the machine. If the machine is shared with multiple JVMs or other CPU intensive processes, the number of GC threads should be the number of cores divided by the number of competing processes. If numactl is used to bind the JVM processes to a set of cores, the number of GC-threads should be equal to the number of cores. This will guarantee that a JVM does not use up all the cores during a GC and potentially affect the performance of the other running processes.

3.3 Use Solid-State Drives (SSD) to Reduce IOWait Time

Enterprise J2EE applications are generally deployed with many threads. Some of these threads will require reading and writing to files on disk, which is much slower than memory. As the number of disk accesses increase, more and more threads will become blocked waiting to access the disk. This behavior will appear as an increase in IOWait time. The threads that are blocked on IOWait will not be able to perform other useful tasks while waiting for the disk. Worst yet, they may be blocking other threads from doing useful work as they may also be holding other valuable resources. This increased IOWait time also leads to increased lock contention.

Many innovative solutions have been created to address this issue, including the use of many very fast spinning disks arranged in a RAID configuration. This solved the high IOWait time problem to a certain extent, but created its own problem. With many very fast spinning disks, many disk failures also become the norm. Many disks also require much space and power. Therefore once we add up the cost of constantly fixing disk errors, and the added space and power requirement, this solution become very expensive to maintain.
Today there is a new solution; using a small number of very fast Solid-State Drives (SSD) to store frequently accessed files and temporary files that must persist even in the event of a power failure. With SSD, such as the Intel® X25-E Extreme SATA Solid-State Drive (http://www.intel.com/design/flash/nand/extreme/index.htm), disk accesses become almost as fast as accessing memory, but still meet the persistence requirement even during power failure. Using SSD greatly reduces IOWait time for J2EE applications and improves the overall scalability of the system.

3.4 Large Pages

Since a Java applications memory is managed by the JVM the application itself has no control over how the objects it uses are located on the heap. The JVM tries to place objects on the heap depending on how they normally are accessed, but memory accesses tend to be spread out on the heap with different memory areas used for newly allocated objects compared to old objects. On all x86 CPUs, the Operating System manages memory using pages. By default the page size is 4 KB. This means when running with a heap of 2 GB the OS has to keep track of 524288 pages for that process alone. Since the CPU cannot keep track of all these pages internally the page tables are stored in memory. Memory is very slow compared to the CPU, so a cache is kept in the CPU called the Translation Look-aside Buffer (TLB) which keeps the most frequently used pages in the CPU for quick access. With more than half a million pages and memory access spread out over a large portion of them the TLB will unable to keep all used pages in the cache.

To remedy this situation Oracle JRockit provides the ability to make use of large pages. Modern x86 CPUs have the ability to use a large page size of 2 MB per page. If 2 MB pages are used for a 2 GB heap the CPU only needs to keep track of 1024 pages. By enabling large pages, we observed more than a 10% gain in performance when measured using SPECjAppServer2004. Enabling large pages requires some OS configuration. For more information on how to configure and use large pages see the Oracle JRockit documentation for large pages, http://download.oracle.com/docs/cd/E13188_01/jrockit/jrdocs/refman/optionX.html#wp999525.

4. JVM TUNING

4.1 64-bit JVM

All recent X86 CPUs have 64-bit address support. Enabling 64-bit on the x86 CPU gives the JVM two benefits, more registers and the support for very large heaps. The increased number of registers allows more data to be handled in the CPU concurrently without the need to spill data to the stack due to register pressure. While the 64-bit address space has the benefit of enabling heaps larger than 4 GB, which is impossible with 32-bit, it also adds the overhead of 64-bit wide addresses. With 64-bit all references on the heap will be double the size compared to 32-bit. This has two implications, a same sized heap will fill up faster and there will be increased pressure on the memory bus. The memory usage increase will not be two fold; the width of primitives is specified in the JVM specification, so only references will double in size. Running with 64-bit will therefore not only enable larger heaps it will also require larger heaps to fit the same number of objects. Following the same reasoning it is easy to see why the pressure on the memory bus will increase; to read the same number of objects with 64-bit, the application will now have to read up to two times the amount of memory compared to the same application running in a 32-bit JVM.

The applications that require some extra innovation from the JVM are smaller applications that don’t require a heap larger than 4 GB, but want to make use of the performance improvement of the extra registers. To make sure this type of application is able to make full use of the extra registers without the overhead of 64-bit references, Oracle JRockit provides a feature called Compressed References. The optimization does exactly what the name suggests; it compresses the references from 64-bit to 32-bit. This will of course limit the maximum heap size to 4 GB, but it will give an application running on Oracle JRockit 64bit JVM all the benefit and none of the overhead associated with 64-bit. Compressed references is automatically enabled when running Java application with heap smaller than 4 GB on Oracle JRockit.

4.2 Aggressive optimization

By default JRockit uses only a single optimization thread. The reason for this is to minimize the impact of the CPU intensive processes of doing optimization on the running Java application. However with the increased number of cores on modern x86 CPUs, such the Quad-Core Intel Xeon processor X5500 series, there might be room to increase the number of optimization threads. This will enable JRockit to optimize frequently executed methods in parallel using multiple threads, without impacting the running application and reach a fully optimized state faster. The number of optimization threads can be set

For our SPECjAppServer2004 runs we also made use of the –XXaggressive flag, http://download-llnw.oracle.com/docs/cd/E13188_01/jrockit/jrdocs/refman/optionXX.html#wp999550. This will further increase both the extent and priority of optimizations done by Oracle JRockit to quickly reach a stable optimized state.

Both these configuration options will help to push performance and reduce the time it takes to reach an optimized state. However with this aggressive optimization behavior there will be fewer CPU cycles for the Java application while the application is being optimized. This might not be the preferred case for latency sensitive applications that need to have low and stable response times response times the whole time. For these applications a longer but more controlled optimization phase may be the preferred solution.

This white-paper serves as good introduction on some of the tuning that can be done with Oracle JRockit, however for a more in-depth and thorough documentation, the Oracle JRockit Diagnostics Guide (http://download.oracle.com/docs/cd/E13188_01/jrockit/geninfo/diagnos/index.html) is highly recommended.

5. WEBLOGIC SERVER TUNING

The Oracle WebLogic Server 10.3 Performance and Tuning guide provides a general guide to tuning WebLogic Server for best performance. It can be found here: http://download.oracle.com/docs/cd/E12840_01/wls/docs103/perform/index.html. This section will highlight some key tunings that can be beneficial for applications like SPECjAppServer2004.

5.1 JDBC tuning

5.1.1 Using Logging Last Resource Optimization

When using a transactional database application, consider using the Logging Last Resource (LLR) transaction policy instead of XA (TwoPhaseCommit). The LLR optimization can significantly improve transaction performance by safely eliminating some of the 2PC XA overhead for database processing, especially for two-phase commit database insert, update, and delete operations. Two phase transactions occur when two different resources participate in the same global transaction (global transactions are often referred to as “XA” or “JTA” transactions). Consider the following:

- Typical two-phase transactions in JMS applications usually involve both a JMS server and a database server. The LLR option can as much as double performance compared to XA.
- The safety of the JDBC LLR option contrasts with well known but less-safe XA optimizations such as “last-agent”, “last-participant”, and “emulate-two-phase-commit” that are available from other vendors as well as WebLogic.
- JDBC LLR works by storing two-phase transaction records in a database table rather than in the transaction manager log (the TLOG).

5.1.2 Using the Prepared Statement Cache

When you use a prepared statement in an application or EJB, there is considerable processing overhead for the communication between the application server and the database server and on the database server itself. To minimize the processing costs, WebLogic Server can cache the prepared statements used in your applications. When an application or EJB calls any of the statements stored in the cache, WebLogic Server reuses the statement stored in the cache. Reusing prepared statements reduces CPU usage on the database server, improving performance for the current statement and leaving CPU cycles for other tasks.

5.1.2.1 Statement Cache Size

The Statement Cache Size attribute determines the total number of prepared statements to cache for each connection in each instance of the data source. By caching statements, you can increase your system performance. However, you must consider how your DBMS handles open prepared statements. In many cases, the DBMS will maintain a cursor for each open statement. This applies to prepared statements in the statement cache. If you cache too many statements, you may exceed the limit of open cursors on your database server.
5.1.3 Setting Initial and Max Capacity

Creating a database connection is a relatively expensive process. Typically, a connection pool starts off with a small number of connections. As the demand for more connections grows, there may not be enough connections in the pool to satisfy the requests. WebLogic Server creates additional connections and adds them to the pool until the maximum pool size is reached.

One way to avoid connection creation delays is to initialize all connections at server startup, rather than on-demand as clients need them. Set the initial number of connections (initial-capacity) equal to the maximum number of connections (max-capacity) in the Connection Pool tab of your data source configuration.

5.1.4 Using Pinned-To-Thread

Pinned-To-Thread can be used to minimize the time it takes for an application to reserve a database connection from a data source and to eliminate contention between threads for a database connection.

When Pinned-To-Thread is enabled, WebLogic Server pins a database connection from a data source to an execute thread the first time an application uses the thread to reserve a connection. When the application finishes using the connection and calls connection.close(), instead of returning the connection to the data source, WebLogic Server keeps the connection with the execute thread. When an application subsequently requests a connection using the same execute thread, WebLogic Server provides the connection already reserved by the thread. There is no locking overhead and no lock contention on the data source when multiple threads attempt to reserve/return a connection from/to the connection pool at the same time.

5.2 EJB tuning

Using read-only concurrency with query-caching or optimistic concurrency with cache-between-transactions for CMP EJBs can greatly help improve performance. Both of these options leverage the Entity Bean cache provided by the EJB container.

5.2.1 Using Read-only Concurrency Strategy

- ReadOnly concurrency strategy is the most performing. When selected, the container assumes the EJB is non-transactional and automatically turns on cache-between-transactions. Bean states are updated from the database at periodic, configurable intervals or when the bean has been programmatically invalidated. The interval between updates can cause the persistent state of the bean to become stale. This is the only concurrency-strategy for which query-caching can be used.

- Query-caching is a WebLogic Server 9.0 feature that allows the EJB container to cache results for arbitrary non-primary-key finders defined on read-only EJBs. All of these parameters can be set in the application/module deployment descriptors. See Concurrency Strategy http://download.oracle.com/docs/cd/E12840_01/wls/docs103/perform/EJBTuning.html#wp1144711.

5.2.2 Using Optimistic Concurrency Strategy

- Optimistic-concurrency with cache-between-transactions=true work best with read-mostly beans. Using verify-rows=Read in combination with these provides high data consistency guarantees with the performance gain of caching. See Tuning WebLogic Server EJBs http://download.oracle.com/docs/cd/E12840_01/wls/docs103/perform/EJBTuning.html.

- The goal of the optimistic concurrency strategy is to minimize locking at the Database while continuing to provide data consistency. The basic assumption is that the persistent state of the EJB is changed very rarely. The container loads the bean without causing locks to be acquired at the data base. At commit-time, if the bean has been modified, a predicated update is used to ensure its persistent state has not been changed by some other transaction. If it has changed, an OptimisticConcurrencyException is thrown and must be handled by the application.

Since EJBs that can use this concurrency strategy are rarely modified, using cache-between-transactions=true can boost performance significantly. This strategy also allows commit-time verification of beans that have been read, but not changed. This is done by setting the verify-rows parameter to Read in the cmp-rdbms descriptor. This provides very
high data-consistency while at the same time minimizing locks at the database. It is also recommended that the optimistic verification be performed using a version column, by setting `verify-columns=Version`.

### 5.2.3 Using eager-relationship-caching

This feature allows the EJB container to load related beans using a single SQL join statement. It improves performance by reducing the number of database calls to load related beans in transactions when a bean and its related beans are expected to be used in the same transaction. See Using Container-Managed Relationships (CMRs)

http://download.oracle.com/docs/cd/E12840_01/wls/docs103/entity.html%23wp1188444.

### 5.2.4 EJB caching

The WebLogic Server Administration Console reports a wide variety of EJB runtime monitoring statistics, many of which are useful for tuning your EJBs. To display the statistics in the Administration Console, see “Monitoring EJBs”

http://download.oracle.com/docs/cd/E12840_01/wls/docs103/ConsoleHelp/taskhelp/ebp/monitorEJBs.html

in Administration Console Online Help. If you prefer to write a custom monitoring application, you can access the monitoring statistics using JMX or WLST by accessing the relevant runtime MBeans. See Runtime MBeans

http://download.oracle.com/docs/cd/E12840_01/wls/docs103/wls_mbeanref/core/index.html

reference in the WebLogic Server MBean

#### 5.2.4.1 Cache Miss Ratio

The cache miss ratio is a ratio of the number of times a container cannot find a bean in the cache (cache miss) to the number of times it attempts to find a bean in the cache (cache access):

\[
\text{Cache Miss Ratio} = \left( \frac{\text{Cache Total Miss Count}}{\text{Cache Total Access Count}} \right) \times 100
\]

#### 5.2.4.2 Setting max-beans-in-cache

A high cache miss ratio could be indicative of an improperly sized cache. If your application uses a certain subset of beans (read primary keys) more frequently than others, it would be ideal to size your cache large enough so that the commonly used beans can remain in the cache as less commonly used beans are cycled in and out upon demand. If this is the nature of your application, you may be able to decrease your cache miss ratio significantly by increasing the maximum size of your cache.

If your application doesn’t necessarily use a subset of beans more frequently than others, increasing your maximum cache size may not affect your cache miss ratio. We recommend testing your application with different maximum cache sizes to determine which give the lowest cache miss ratio. It is also important to keep in mind that your server has a finite amount of memory and therefore there is always a trade-off to increasing your cache size.

#### 5.2.4.3 Using disable-ready-instances

The use of caches in the WebLogic EJB container often provides a significant performance benefit. The WebLogic EJB 2.1 container caches beans in two distinct states – active and ready. An active bean is currently enlisted in a transaction. After completing the transaction, the instance becomes ready and remains in the cache, in least-recently-used (LRU) order, until space is needed for other beans.

This ready cache helps for read-only concurrency or optimistic concurrency with cache-between-transactions. But for very active beans with database concurrency and high cache miss ratios the overhead of maintaining the LRU ready cache can outweigh any benefits. In this case setting disable-ready-instance will avoid the overhead of maintaining the LRU ready cache.

### 5.3 SocketMuxer tuning

#### 5.3.1 Changing the Socket Muxer

WebLogic Server use a piece of native code called the SocketMuxer to improve performance while reading from sockets. On Linux, the default Socket is the EPoll Socket Muxer. On most platforms the NIO Socket Muxer is also available. You can
specify which socket muxer you want to use by setting a WLS command line argument. For example, you can specify the NIO Socket Muxer with:
-Dweblogic.MuxerClass=weblogic.socket.NIOSocketMuxer

5.3.2 Change # of socket reader threads

By default WebLogic Server sets the number of socket reader threads to the number of CPUs on the machine + 1. In some cases the application may not require that many socket reader threads to handle its peak load. Since only one thread can enter the OS to wait for socket activity, the rest of the idle threads repeatedly needlessly contest for a lock and then go to sleep.

If the application can handle the load with fewer socket muxer threads, you can reduce number of socket reader threads and eliminate the lock contention by setting a WLS command line argument. For example you can set the number of socket reader threads to 1 with:
-Dweblogic.SocketReaders=1

6. SUMMARY

This paper describes advances in hardware and software can bring better performance for Java application servers. To get the best performance, the system must be tuned at multiple layers of hardware, operating systems, Java virtual machines and application servers. It outlines a step-by-step, iterative approach to increase the performance of Java application servers. It also lists common configurations that may help get out the last bit of performance. By characterizing Java application server workloads, it provides the rationale behind the tuning methodology. The case study described in this paper demonstrates significant performance improvement can be obtained by following the tuning guidance.

7. REFERENCES
