Performance Evaluation of J2EE(2)

Presented by:
Henry Xu (20198718)
Huahao Zhang (20256751)
July 26, 2007
Outline

- Introduction
  - Problems, motivations and approach
  - Environment, methods and architecture
- Performance evaluation
  - Web performance evaluation
  - Java performance evaluation
- Summary
  - Achieved results
  - Contributions
Problems and Motivations

- Problems
  - Does J2EE5 improve application performance?
  - What are the performance bottlenecks in J2EE5?

- Motivations
  - Performance evaluation
  - Try to find out bottlenecks
Our Approach

1. Setup J2EE5 and deploy PetStore2.0
2. Analyze the MVC and other patterns
3. Study the new Web2.0 techniques
   *Step 2 & 3 have been done by tracing the source code manually*
4. Set up load testing environment
5. Create test script and run load test
6. Evaluate the performance and find out the bottlenecks in J2EE5
   *Step 5 & 6 have been done by using the LoadRunner and Java Profiler*
# Testing Environments

## Hardware

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Centrino II 1.4G</td>
<td>AMD Athlon 64 3200+</td>
</tr>
<tr>
<td>Memory</td>
<td>1G</td>
<td>2G</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>40G, 5400RPM</td>
<td>250G, 7200RPM</td>
</tr>
<tr>
<td>Network</td>
<td>100M LAN</td>
<td>100M LAN</td>
</tr>
</tbody>
</table>

## Software

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>WIN XP + SP2</td>
<td>WIN XP + SP2</td>
</tr>
<tr>
<td>Testing tools</td>
<td>LoadRunner 8.1</td>
<td>NetBeans Java Profiler 5.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SUN J2EE App Server 9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NetBeans 5.5 IDE+JDK 1.6</td>
</tr>
</tbody>
</table>
Testing Methods

- Client
  - Black box testing
  - Testers know nothing about source code
  - Focusing on web performance

- Server
  - White box testing
  - Tester know the source code well
  - Focusing on Java class and method level performance

- J2EE performance evaluation
  - Based on both web performance and Java performance
Testing Architecture (Ctrl. Flow)
Testing Architecture (Data Flow)

**Client Side**
- Tester
- Virtual User Generator
- LoadRunner Controller
- LoadRunner Analyser
  - Web performance results
  - Input
  - Generate scripts

**Server Side**
- PetStore 2.0
- SUN App Server
- NetBean IDE 5.5
  - Java Performance Results
  - Attach
  - Output
- Netbean Profiler 5.5
  - Output
  - Java Performance Reports

- Generate HTTP requests and receive HTTP responses
- Attach

**Flow**
- Tester → Virtual User Generator
- Virtual User Generator → LoadRunner Controller
- LoadRunner Controller → LoadRunner Analyser
- LoadRunner Analyser → Web Performance Reports
- Web Performance Reports → Server Side
  - SUN App Server
  - PetStore 2.0
  - NetBean IDE 5.5
  - Netbean Profiler 5.5
Web Performance Comparison

- **Compare the performance of Ajax web pages to non-Ajax web pages**
  - Create a new JSP file that implements the same autocomplete feature using synchronous (non-Ajax) architecture: Copy the FileUpload.jsp to FileUpload_noajax.jsp and then add the following code to FileUpload_noajax.jsp

```jsp
<%@page import="com.sun.javaee.blueprints.petstore.controller.*"%>

<% //Call the methods directly to simulate synchronous calls
    AutocompleteBean clsAutocompleteBean = new AutocompleteBean();
    //Search cities from Java database where the first character is ‘w’
    String decodedStr = clsAutocompleteBean.initCities("w", 10);
    out.println("Debug: decodedStr="+ decodedStr);
%>
```
Web Performance Comp. (Packet Size)

- Non-Ajax (first time): `web_url("fileupload_noAjax.jsp")`,
  - **14,146** header bytes, **647,602** body bytes,
- Non-Ajax (caching): `web_url("fileupload_noAjax.jsp")`
  - **3,138** header bytes, **220,853** body bytes,
- Ajax: `web_url("fileupload.jsp")`
  - **324** header bytes, **340** body bytes,
Web Performance Comp. (Http Response)

<table>
<thead>
<tr>
<th>Scale</th>
<th>Measurement</th>
<th>Min</th>
<th>Ave</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non Ajax</td>
<td>1</td>
<td>HTTP_200</td>
<td>20.5</td>
<td>38.6</td>
</tr>
<tr>
<td>Ajax</td>
<td>1</td>
<td>HTTP_200</td>
<td>24</td>
<td>45.4</td>
</tr>
</tbody>
</table>

- **Reasons**
  - Very difficult to fully convert the Ajax applications to non-Ajax version
  - The autocomplete feature is IO-bound
Java Profiling Results

- The results include the percentage of CPU execution time, execution time and the number of invocations of each method
- Call Tree: showing the top-down method call chain
Hot Spots

- What are hot spots?
  The methods that consume the most of CPU time.

- TOP 5 hot spots
  The first 5 methods in the hot spots list which consume around 60% of CPU execution time in our load test
Top 1 hot spot: `sun.nio.ch.WindowsSelectorImpl$SubSelector.poll()`

Track back previous callers in the call tree. The first caller is `CatalogFacade: GetTagsInChunk()`
Top1 Hot Spot Analysis (Cont.)

- **Source Code**
  
  ```java
  public List<Tag> getTagsInChunk(int start, int chunkSize) {
    1. EntityManager em = emf.createEntityManager();
    2. Query query = em.createQuery("SELECT t FROM Tag t ORDER BY t.refCount DESC, t.tag");
    3. List<Tag> tags = query.setFirstResult(start).setMaxResults(chunkSize).getResultList();
    4. em.close();
    5. return tags;
  }
  ```

- **Reason analysis**
  1. The purpose of this function is to select all tag records from Java database.
  2. Database operation (SQL execution + network communication)
  3. Since the EntityManager class belongs to the JPA component, we believe the bottleneck is caused by the lack of support for caching in the JPA component

- **Solutions**
  1. Cache the tag records after they have been selected from the Java database
  2. By our estimation, performance is expected to be improved by 25%
Top2-Top5 Hot Spots Analysis

- Top2 to Top5 hot spots are located under the same call tree
- The first caller is `EntityFilter::doFilter()`
Top2-Top5 Hot Spots Analysis (Cont.)

- **Source code**
  
  ```java
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
    ...
    chain.doFilter(request, response); // This line is the bottle neck
    ...
  }
  ```

- **Reason analysis**
  1. `com.sun.javaee.blueprints.petstore.controller.EntryFilter.doFilter()` is provided by the J2EE5 framework based on the apache core library(`org.apache.catalina.core.ApplicationFilterChain`)
  2. The Controller in the M-V-C model

- **Suggestion**
  1. Use other MVC frameworks (Spring, Struts) instead of the one provided by SUN and Apache
Conclusions

Performance evaluation of J2EE 5
- Performance is acceptable but not very impressive
- Provides re-usable components that simplify the development of web applications

JPA (Java Persistence API)
- **Advantage**
  - Simplifies the development of database interfaces
- **Disadvantage**
  - The bottleneck (40% execution time) in the system due to lack of support for data caching
- **Solution**
  - Caching

JSF (Java Server Faces) and AJAX (Asynchronous JavaScript and XML)
- **Advantage**
  - Simplify the development of Ajax-Enabled components.
  - Reduce network traffic significantly (depends on web page size)
  - Improve performance by 20% in our testing
- **Disadvantage**
  - Project length is extended
- **Suggestion**
  - Evaluate the application type before we make the decision. If the application is IO-bound, Ajax should be used. If it is CPU-bound, Ajax won’t offer much performance benefit.
Contributions

- Our report will be useful for the following people
  1. Java programmers who are interested in developing and deploying Java EE 5 applications on the Sun Application Server 9.
  2. Web developers who want to quickly develop and deploy Ajax-enabled JSF applications
  3. Architects who evaluate different types of frameworks before deciding which framework to be used in their systems
Thank you!