Using the Middle Tier to Understand Cross-Tier Delay in a Multi-tier Application

Haichuan Wang(1), Qiming Teng(1), Xiao Zhong(1), Peter F. Sweneey(2)

(1) IBM Research – China, {wanghaic, tengqim, zhongx}@cn.ibm.com
(2) IBM Watson Research Center, pfs@cn.ibm.com
Motivation

- Enterprise applications have multi-tier architectures
- A performance bottleneck on any tier may cause the whole system to under perform

Observed:
- Low throughput
- Low CPU utilization

Which tier causes the bottleneck?
Previous Approaches

1. Collect system metrics on all tiers
   – Statistics on each machine
2. Aggregate resource consumption
   – Interaction between machines
3. Build a whole system interaction model

- Some Limitations
  – Hard to collect system metrics on all tiers in some production systems
    • e.g. Thousands of clients; out-bound servers
  – Complex
  – Identify performance bottlenecks based on a large number of metrics
Proposed Approach – Focus on the Middle-tier

- Focus on the middle-tier
  - Application server (Java based)

- Track cross-tier method invocations in Java level
  - Identify method invocations that handle cross-tier interactions
  - Extract “contextual information” associated with these method invocations

- Identify the blocking in native level
  - Trace thread interruptible (blocking) state
  - Map back to the cross-tier method invocations

- Refer to the blocking source tier
  - Based on the contextual information

Example:

- SocketRead(…)
  - Call Stack: LDAP Connector IP Address: xxx → LDAP Server
  - Thread Blocking State: SocketRead(…): Blocked
  - Tier of Blocking Source: LDAP server (xxx)
Solution Architecture Overview

- Constructing Cross-Tier Delay data from the following data
  - Method invocation by dynamically byte code instrumentation
  - Context information by dynamically instrumentation
  - Thread States by JVMTI agent and a kernel module
Tracing Method Invocations – Class Instrumentation

- Tracing rules driven Java byte code instrumentation

Example Tracing Rules

```
Method:
java.io.InputStream java.net.PlainSocketImpl.getInputStream();
Parameters Mask:
Record Return: False
Fields:
java.net.InetAddress address;
java.io.FileDescriptor fd;
```

```
Method:
int java.net.SocketInputStream.socketRead0(java.io.FileDescriptor para0, byte[] para1, int para2, int para3, int para4);
Parameters Mask: 10000
Record Return: True
Fields:
```

- Method Signature
- Parameters Masks
- Return Value Indicator
- Field List

- Method Invocation Time & Duration
- Method Invocation Stack
- Value of Parameters
- Value of the Return
- Value of the Fields
Three different approaches for dynamically instrumenting methods
  – Create proxy methods
  – Directly instrument the prolog and epilog of an identified method
    • In case we cannot insert the proxy
  – Instrument all call-sites of an identified method
    • For tracing “JNI” methods in the JVM without the JNI prefix mechanism
Using the Middle Tier to Understand Cross-Tier Delay in a Multi-tier Application

Trace Thread Blocking

- Kernel Module
  - Based on Kprobe (Linux)
  - Inserted into OS scheduler
  - Only collect thread interruptible native states (blocked)

- JVMTI Agent
  - Assist to map native threads to corresponding Java threads

Application Java Process

JVMTI agent

Kernel Module

OS Kernel Space
Using the Middle Tier to Understand Cross-Tier Delay in a Multi-tier Application

Merge the Data – Analysis Engine

Process Method Invocation Data

Merge Java And Native Level

Aggregate by Contexts (Tiers)

Method Invocation Trace

Thread Blocking Trace

Analysis Result

fd = fd@343f343f
Address = 9.186.62.68

Thread = WebContainer1
Start = 10 ms
End = 20 ms
fd = fd@343f343f
Callstack = Socket Read …

Thread = WebContainer1
SocketRead on 9.186.62.68
Start = 10ms; End = 20ms

Thread = WebContainer1
Blocked on 9.186.62.68
Duration = 8 ms

Thread = WebContainer2
Blocked on 9.186.62.68
Duration = 6 ms

Thread = WebContainer1
Blocking start = 11 ms
Blocking end = 19 ms

Thread = WebContainer1
Blocked on 9.186.62.100
Duration = 20 ms

Thread = WebContainer2
Blocked on 9.186.62.100
Duration = 12 ms

Thread = WebContainer2
Blocked on 9.186.62.100
Duration = 32 ms
Case Study

- **DayTrader**
  - Multi-tier architecture
  - J2EE application
  - Simulate Stock Trading

**Deployment Details in the Study**

![Diagram of DayTrader architecture]

- **Client Simulator**
  - Intel Dual-Core Xeon LV 1.66GHz, 2 Processors, 4G Memory

- **DayTrader**
  - Version: 2.0

- **Application Server**
  - IBM WAS Version 6.1.0

- **JVM**
  - IBM Java 5.0

- **OS**
  - RHEL Kernel 2.6.18

- **Database**
  - IBM DB2 8.0

- **Intel P4 Xeon, 2.4GHz**
  - 1 Processor, HT Enabled
  - 2G Memory
Using the Middle Tier to Understand Cross-Tier Delay in a Multi-tier Application

**Low Clients Loads → High Clients Loads**

<table>
<thead>
<tr>
<th>Config</th>
<th>Low Clients Load</th>
<th>High Clients Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Limit client requests to 650/s</td>
<td>Increase client requests to 2900/s</td>
</tr>
<tr>
<td>Utilization</td>
<td>WAS CPU % = 6.5%</td>
<td>WAS CPU % = 30.9%</td>
</tr>
<tr>
<td>Cross Tier Wait Time Analysis from Middle Tier’s Perspective</td>
<td><strong>DB Server Delay</strong> 33.83 s (21.71%) <strong>WAS Delay</strong> 0.69 s (0.44%) <strong>Clients Delay</strong> 121.26 s (77.85%)</td>
<td><strong>WAS Delay</strong> 29.62 s (6.25%) <strong>Clients Delay</strong> 50.92 s (10.75%) <strong>DB Server Delay</strong> 393.10 s (83.00%)</td>
</tr>
</tbody>
</table>

Clients cause the most cross tier waiting time

DB server causes the most cross tier waiting time
Analyzing Socket Read Time

- Cross tier delay on DB server is in SocketRead invocations
  - Action: Study the socket read time in two loads

![Graphs showing Socket Read Time](image)

- Conclusion
  - DB server’s slow response causes the low 30% utilization in WAS
- Action
  - Upgrade DB server to 2 Xeon 5345 Processors, total 8 way.
  - Result: Client Request Rate > 4,600/s. WAS CPU utilization = 51%
# High Clients Loads ➔ Upgrade DB Server

<table>
<thead>
<tr>
<th>Config</th>
<th>High Clients Load</th>
<th>Upgrade DB Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Increase client requests to 2900/s</td>
<td>Reach to over 4,600/s</td>
</tr>
<tr>
<td>Utilization</td>
<td>WAS CPU % = 30.9%</td>
<td>WAS CPU % = 51%</td>
</tr>
</tbody>
</table>
| Cross Tier Wait Time Analysis from Middle Tier’s Perspective | WAS Delay 29.62 s (6.25%)  
Clients Delay 50.92 s (10.75%)  
DB Server Delay 393.10 s (83.00%)  
DB server causes the most cross tier waiting time | WAS Delay 45.81 s (9.94%)  
Clients Delay 53.52 s (11.61%)  
DB Server Delay 361.69 s (78.46%)  
Blocking time on DB server reduced. |
Overhead Analysis

- **Approaches Used to Reduce Overhead**
  - **Kernel Module**
    - Only filter block threads
  - **Byte Code Instrumentation**
    - Only instrument selected method invocations
    - Aggressively use final and private keywords
    - Cache trace events in an array based in-memory buffer

- **Resulting Overhead**
  - **Config:** DB server uses the upgraded hardware configuration (8 way)

<table>
<thead>
<tr>
<th>Tracing Rule</th>
<th>Request Rate</th>
<th>Slow Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>4,699/s</td>
<td>0.0%</td>
</tr>
<tr>
<td>With the tool</td>
<td>4,169/s</td>
<td>11.3%</td>
</tr>
</tbody>
</table>
Thank you!

Q & A