Lightweight Application-level Task Migration for Mobile Cloud Computing

Ricky K. K. Ma, Cho-Li Wang
28 Mar 2012

Systems Research Group
Department of Computer Science
The University of Hong Kong
Outline

- Research background and motivation
- System design and implementation
- Performance evaluation
Background

- **Mobile cloud computing:**
  - Mobile apps or widgets connect to the Cloud
  - Support more complex and wider range of applications

More than 4.5 billion mobile-phone users all over the world.

- "Music Anywhere"
- AI voice-recognition engines
Problems

- API lock-in $\rightarrow$ Service Provider lock-in
- Client-server model: restricted form of computing
Motivation:

- Migration techniques are required to **dynamically** move computation between mobile nodes and cloud nodes:
  - **Low overhead:**
    - Especially when using in mobile nodes where processing power and resources are very limited
  - **Portable:**
    - Heterogeneous mobile nodes + Heterogeneous cloud nodes
    - Task migration among mobile nodes and cloud nodes
    - (Language-level virtualization for Cloud Computing)
# HKU eXCloud Project Overview
(part of China National Grid (CNGrid))

## Multi-level Mobility Support

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Migration Technique (System)</th>
<th>Target System Type (Area)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame level</td>
<td>Stack-on-demand (SODEE)</td>
<td>Cloud, cloudlet, mobile network (WAN/LAN)</td>
</tr>
<tr>
<td>Thread level</td>
<td>Thread migration (JESSICA2)</td>
<td>Cluster (LAN)</td>
</tr>
<tr>
<td>Process level</td>
<td>Process migration (G-JavaMPI)</td>
<td>Grid (WAN/LAN)</td>
</tr>
<tr>
<td>VM level</td>
<td>Live VM migration (Xen)</td>
<td>Cluster (LAN)</td>
</tr>
<tr>
<td></td>
<td>Wide-area live VM migration (WAVNet)</td>
<td>Cloud, p2p/desktop cloud (WAN)</td>
</tr>
</tbody>
</table>

**Adaptation granularity**

- **Coarse**
- **Fine**

**System size**

- **Small**
- **Large**
Stack-On-Demand (SOD): Key Ideas

- Allow lightweight task migration

Migrating task on Source Node (a running thread)

Worker process on Destination Node

Stack frame A

Method

Stack frame A

Method

Stack frame A

Method

Stack frame B

Local variables

Program Counter

Local variables

Program Counter

Local variables

Program Counter

Local variables

Program Counter

Heap Area

Object Pre-fetching

Mobile node

Stack-On-Demand (SOD) Details:
“Minimum State Migration”

Cloud node

Execution Resumed

Heap Area

Heap Area Rebuilt

2012-3-30
Existing Approaches to Migrate Tasks

• At JVM level
  • Modifying JVM
    • Sumatra, ITS, CTS, JESSICA\textsubscript{2}
  Requires intensive modification of JVM
• As middleware
  • through JVMTI interface
    • CIA project, G-\textit{JavaMPI}
  • JVM extension
    • Mobile JikesRVM
  JVMTI not available on mobile nodes
• Application-level task migration
  • Rewriting bytecode
    • JavaGoX, Brakes, JavaSplit
  • Rewriting source code
    • WASP, JavaGo
  Requires certain extension of JVM

We focus on application-level task migration
<table>
<thead>
<tr>
<th>Project</th>
<th>Level</th>
<th>Category</th>
<th>Granularity</th>
<th>Capturing techniques</th>
<th>Restoring techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merpati</td>
<td>JVM</td>
<td>Interpreter</td>
<td>thread</td>
<td>Keep state in portable format</td>
<td>Reconstruct based on the state</td>
</tr>
<tr>
<td>JavaThread</td>
<td>JVM</td>
<td>Interpreter</td>
<td>thread</td>
<td>Keep state in portable format</td>
<td>Reconstruct based on the state</td>
</tr>
<tr>
<td>ITS</td>
<td>JVM</td>
<td>Interpreter</td>
<td>thread</td>
<td>Keep state in portable format</td>
<td>Reconstruct based on the state</td>
</tr>
<tr>
<td>CTS</td>
<td>JVM</td>
<td>JIT-compliant</td>
<td>thread</td>
<td>State in portable data structure</td>
<td>Reconstruct based on the state</td>
</tr>
<tr>
<td>JESSICA2</td>
<td>JVM</td>
<td>JIT-compliant</td>
<td>thread</td>
<td>JIT recompilation</td>
<td>Reconstruct based on the state</td>
</tr>
<tr>
<td>CIA</td>
<td>Middleware</td>
<td>extension of JVM</td>
<td>thread</td>
<td>JVMDI + bytecode instrumentation</td>
<td>JVMDI + bytecode instrumentation</td>
</tr>
<tr>
<td>Mobile JikesRVM</td>
<td>Middleware</td>
<td>JVM</td>
<td>thread</td>
<td>Use extensions of JikesRVM</td>
<td>Use extensions of JikesRVM</td>
</tr>
</tbody>
</table>
| Wasp             | Application | Source-code preprocess | thread      | 1. Java Language extended  
                                2. Exception (not asynchronous), but need to add migration points explicitly  
                                3. Part of state always saved in each migration point | State-polling codes                          |
| JavaGo           | Application | Source-code preprocess | thread      | 1. Java Language extended  
                                2. Exception (not asynchronous), but need to add migration points explicitly | State-polling codes                          |
| Our approach     | Application | Bytecode preprocess | stack frame | Asynchronous exception (no need to add migration point and state-polling codes)     | Twin Method Hierarchy  
                                (State-restoring codes executed only during restoration) |
| JavaGoX          | Application | Bytecode preprocess | thread      | Exception (not asynchronous), but need to add migration points explicitly.            | State-polling codes                          |
| MAG/Brakes       | Application | Bytecode preprocess | thread      | State-polling codes.                                                               | State-polling codes                          |
System Design

- **Design goal**
  - **Low overhead**
    - Allow lightweight task migration. Induce low overhead.
  - **Transparency**
    - No need for users to modify their programs
  - **Portability**
    - No need to use a specific JVM.
  - **Flexibility: Adaptation to new environment**
    - allow to use resources in new location to utilize resources (or better resource utilization)
Common approach in application-level migration

- Use of status-polling for detecting requests
- The status-polling codes are executed even when there are no migration

Instrumentation 1: Use of status-polling for detecting requests

1. Status-polling codes are added for each migration point

```java
original statements of the function

call func2()
if (isCapturing()) then
    store stackframe into context
    store artificial PC as index value
    return
end if
remaining statements of the function
```

2. Status-polling codes are added after each function call

- The location of inserted codes determine the migration points
- Finer granularity of migration => more insertion of status-polling codes => large overhead
• Use of status-polling for detecting restoration
• Status-polling codes are added at the beginning of each function call
• The status-polling codes are executed even when there are no migration

**Instrumentation 2: Status-polling for detecting restoration**

```plaintext
if (isRestoring()) then
  get artificial PC from context
  switch (artificial PC)
    case invoke1:
      load stackframe
      goto invoke1
    case ...
    ...
  end switch
end if
original statements of the function
```

Status-polling codes are added at the beginning of each function call
Our approach

- **Fine-grained Task Migration**
  - Among cloud nodes + Between a mobile node and a cloud node
  - Granularity: *Java Stack Frame*

- **Two types of migration**
  - **Active:** Triggered by migration manager
    - E.g. over loading
  - **Pro-active:** Triggered by the program itself
    - Eg. `ClassNotFoundException`, `OutofMemoryException`
    - Migration manager would then receive the requests, choose the appropriate destination and perform the migration
• Task Migration
  
  • State-capturing with Asynchronous Exception
  
  • Avoid status-polling (less time overheads)
  • During normal execution, as no extra codes are executed, no overhead are introduced.
  • Allow finer granularity of migration

<table>
<thead>
<tr>
<th>Instrumentation with use of asynchronous exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. try</td>
</tr>
<tr>
<td>2. original statements of function</td>
</tr>
<tr>
<td>3. catch MigrationException</td>
</tr>
<tr>
<td>4. capture state</td>
</tr>
<tr>
<td>5. throw MigrationException</td>
</tr>
<tr>
<td>6. end try</td>
</tr>
</tbody>
</table>

Capturing codes are inserted as exception handler

No significant overhead introduced during normal execution
- **Issues working with asynchronous exception**
  - **Data inconsistency**
    - intermediate results are stored in *operand stacks*, or in *native methods*
  - **Solution: bytecode rearrangement**
    - *Extra local variables* are used to save intermediate results
    - *Extra flags* are used to inhibit migration at certain points
  - **Deadlock**
    - Can lead to deadlock if asynchronous exception is used in uncontrolled manner
**State Restoring with Twin Method Hierarchy**

- A bytecode instrumentation technique, minimize the overhead in normal execution

**Twin Method Hierarchy**

- Keep both *instrumented* and *original* methods
- Normal execution: original methods
- Restoration: the instrumented methods with restoration statements are executed.
  - Checking statements are added at the beginning of the duplicated functions
- When restoration is completed, the original method will be executed
Example

void func1(){
    func2();
    return;
}

1. During normal execution, original method func1() and func2() are executed => no overhead introduced

void SOD_func1() {
    if (isRestoring()) {
        restore_state();
        if (need_restore_other_frame)
            goto Label1
        else
            goto previously_suspended_location
    }
    func2();
    Label2:
    return;
    label1:
    SOD_func2();
goto Label2
}

2. After restoration, method func1() and func2() are executed => no overhead introduced

Original method
Method used during restoration
Instrumented method is executed during restoration only

original method is executed after restoration has been done
Performance Evaluation

- **Cloud server nodes**
  - Each node: 2 x Intel E5540 4-Core Xeon 2.53 GHz CPU, 32GB DDR3 RAM,
  - OS: Fedora 11 x86_64
  - JVM: Sun JDK 1.6 (64 bit)
  - Network: Gigabit Ethernet

- **Mobile nodes**
  - **iPhone 4 handset**: 800MHz CPU, 512 MB RAM
  - JVM: JamVM 1.5.1b2-3, slightly modified to expose the asynchronous exception API
  - Java class library: GNU Classpath 0.96.1-3
  - Connected to Cluster through Wi-Fi (bandwidth controlled by a router)
Performance Evaluation

- Focus on performance of task migration with SOD migration

<table>
<thead>
<tr>
<th>Evaluations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Overhead analysis in cloud nodes (No Migration)</td>
</tr>
<tr>
<td>B</td>
<td>Overhead analysis in mobile nodes (No Migration)</td>
</tr>
<tr>
<td>C</td>
<td>Migration from mobile node to cloud node (by active migration)</td>
</tr>
<tr>
<td>D</td>
<td>Migration from mobile node to cloud node (by pro-active migration)</td>
</tr>
</tbody>
</table>
### Evaluation A & B: Overhead Analysis

#### Testing programs

<table>
<thead>
<tr>
<th>App</th>
<th>Description</th>
<th>Max. stack height</th>
<th>Total field size (byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>Calculate 46th Fib. No.</td>
<td>46</td>
<td>&lt; 10</td>
</tr>
<tr>
<td>NQ</td>
<td>Solve N-Queens problem with board size 14</td>
<td>16</td>
<td>&lt; 10</td>
</tr>
<tr>
<td>FFT</td>
<td>Calculate 256-point 2D FFT</td>
<td>4</td>
<td>&gt; 64M</td>
</tr>
</tbody>
</table>

#### Evaluation of Three Migration Techniques:

- **SOD migration using JVMTI (SOD_JVMTI)**
  - implemented as JVMTI agents
  - A middleware approach (Only available on Cloud nodes)
- **SOD migration using status-checking (SOD_P)**
  - implemented at application level
- **SOD migration using asynchronous exception (SOD_AE)**
  - implemented at application level
### Evaluation A: Execution time on cloud nodes (overhead when NO migration)

<table>
<thead>
<tr>
<th></th>
<th>Orig time (s)</th>
<th>SOD_JVMTI time (s)</th>
<th>overhead (%)</th>
<th>SOD_AE time (s)</th>
<th>overhead (%)</th>
<th>SOD_P time (s)</th>
<th>overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>12.11</td>
<td>12.13</td>
<td>0.17</td>
<td>12.14</td>
<td>0.25</td>
<td>18.4</td>
<td>51.78</td>
</tr>
<tr>
<td>NQ</td>
<td>6.35</td>
<td>6.4</td>
<td>0.79</td>
<td>6.7</td>
<td>5.51</td>
<td>7.24</td>
<td>14.02</td>
</tr>
<tr>
<td>FFT</td>
<td>10.53</td>
<td>10.63</td>
<td>0.95</td>
<td>10.82</td>
<td>2.75</td>
<td>10.6</td>
<td>0.47</td>
</tr>
</tbody>
</table>

- **SOD_JVMTI impos{es the smallest overhead**
- the lower layer implementations.
- **Mobile devices do not support JVMTI**
- **SOD_AE is slightly higher than SOD_JVMTI (< 5%)**
**Evaluation B: Execution time on mobile nodes (overhead when NO migration)**

<table>
<thead>
<tr>
<th></th>
<th>Orig</th>
<th>SOD_AE</th>
<th>SOD_P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>time (s)</td>
<td>overhead (%)</td>
</tr>
<tr>
<td></td>
<td>time (s)</td>
<td>time (s)</td>
<td>overhead (%)</td>
</tr>
<tr>
<td>Fib</td>
<td>10.85</td>
<td>10.86</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NQ</td>
<td>32.13</td>
<td>32.23</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>5.39</td>
<td>5.4</td>
<td>0.19</td>
</tr>
</tbody>
</table>

- **SOD_JVMTI** not reported (as JVMTI is not available for JVM in mobile devices)
- **SOD_AE has the smallest overhead (<0.31%)**
Evaluation C: Migration from mobile device to cloud node

- **Active migration** for performance improvement
  - Migrate computation-intensive tasks from mobile devices to Cloud nodes. Upon finish of tasks, execution with data are migrated back to mobile devices.

- **Performance gain**: FFT: x3.8, NQ: x30, Fib: x57 times.

<table>
<thead>
<tr>
<th></th>
<th>exec. time w/o mig. (s)</th>
<th>exec. time w/ mig. (s)</th>
<th>Speed up</th>
<th>(A) capture time (ms)</th>
<th>(B) transfer time (ms)</th>
<th>(C) restore time (ms)</th>
<th>Total migration latency (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>56.79</td>
<td>0.99</td>
<td>57</td>
<td>140.33</td>
<td>94.33</td>
<td>11.67</td>
<td>0.246</td>
</tr>
<tr>
<td>NQ</td>
<td>32.67</td>
<td>1.04</td>
<td>30</td>
<td>183.26</td>
<td>86.31</td>
<td>10.52</td>
<td>0.280</td>
</tr>
<tr>
<td>FFT</td>
<td>6.06</td>
<td>1.26</td>
<td>3.8</td>
<td>156.48</td>
<td>232.46</td>
<td>14.58</td>
<td>0.403</td>
</tr>
</tbody>
</table>

Total migration latency (s) = A+B+C

*FFT: Fast Fourier Transform, NQ: Number of Quotients, Fib: Fibonacci*
Evaluation D: Migration from mobile nodes to cloud node (Pro-Active)

- Two applications executed in mobile nodes
  - DBRetrieve and FaceDetect
  - Both require special resources not available in mobile nodes

- **DBRetrieve**: During execution
  - trying to execute JDBC driver ….
  - the required driver is missing

```
exception
NoClassDefFoundError
```

- mobile node
- cloud node
- database server
- queries executed
- **FaceDetect**: Finds regions of faces in photos that are stored in iPhone
- **Requires OpenCV library**
  - open-source library for real-time computer vision and image processing
  - platform-dependent, not available in iPhone

trying to call the library OpenCV …

```
Exception NoClassDefFoundError
```

mobile node  \[ \rightarrow \]  cloud node
Performance Evaluation

<table>
<thead>
<tr>
<th>apps</th>
<th>capture time (ms)</th>
<th>transfer time (ms)</th>
<th>restore time (ms)</th>
<th>total migration latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBRetrieve</td>
<td>85</td>
<td>76</td>
<td>6</td>
<td>167</td>
</tr>
<tr>
<td>FaceDetect</td>
<td>103</td>
<td>155</td>
<td>7</td>
<td>265</td>
</tr>
</tbody>
</table>

If no SOD migration, the applications cannot be executed in mobile devices at all due to the missing resources.
Conclusion and Future Work

• Java bytecode transformation technique
  • Transparent task migration in a portable and efficient manner

• Application level ➔ Higher portability
  • Migration can take place among mobile nodes and cloud nodes
  • Does not impose significant overhead on mobile devices

• SOD Support More Flexible Computing in Cloud
  • RMI-style, process roaming, workflow model
  • Improved resource utilization
  • Avoid (API & provider) lock-in

• Future Work:
  • Killer applications of SOD?
  • Migration policies? (Resource-driven, Cost-driven, Energy-aware,..)
  • Object pre-fetching, frame-based task scheduling, ..
Thank you!

Q & A