Just-in-Time Provisioning for Cyber Foraging

Kiryong Ha*, Padmanabhan Pillai†, Wolfgang Richter*
Yoshiiisa Abe*, Mahadev Satyanarayanan*

*Carnegie Mellon University, †Intel Labs

6/27/2013
Cloud Offloading

Rich, interactive applications are emerging in mobile context

- Apple’s Siri, AR apps..
- Wearable devices push this trend even more!

Cloud offloading

- These applications are too expensive to run on clients alone!
- Offload computation to a back-end server at cloud
- MAUI (Mobisys ‘10), Odessa (MobiSys ‘11), COMET (OSDI ‘12)

Today’s cloud is a suboptimal place; high latency and limited bandwidth
Cloudlet: an nearby offloading site dispersed at the edges of the Internet

Let’s bring the cloud closer!

How to launch a custom back-end server at an arbitrary edge?
Challenges in provisioning

1. Support **widest range of user customization** including OS, language, and library
2. Strong **isolation** between untrusted computations
3. Access control, metering, dynamic resource management, ...

→ VM (virtual machine) cleanly encapsulates this complexity, but delays provisioning

**GOAL : Just-in-time provisioning of a custom VM for offloading**
VM Synthesis: dividing a custom VM into two pieces

1) Base VM: Vanilla OS that contains kernel and basic libraries
2) VM overlay: A binary patch that contains customized parts
VM Synthesis

Steps for VM synthesis

User

Cloudlet with pre-populated base VM

Offload operations

Synthesize VM
- Decompress
- Apply delta

Resume launch VM

Backend Server in VM

VM overlay

ready
VM Synthesis – Baseline Performance

- Performance measurement with rich, interactive applications
- **Base VM:** Windows 7 and Ubuntu 12.04
  - 8GB *base disk* and 1GB *base memory*

<table>
<thead>
<tr>
<th>Application</th>
<th>Install size (MB)</th>
<th>Overlay Size</th>
<th>Synthesis time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Disk (MB)</td>
<td>Memory (MB)</td>
<td></td>
</tr>
<tr>
<td><strong>OBJECT</strong></td>
<td>39.5</td>
<td>92.8</td>
<td>113.3</td>
</tr>
<tr>
<td><strong>FACE</strong></td>
<td>8.3</td>
<td>21.8</td>
<td>99.2</td>
</tr>
<tr>
<td><strong>SPEECH</strong></td>
<td>64.8</td>
<td>106.2</td>
<td>111.5</td>
</tr>
<tr>
<td><strong>AR</strong></td>
<td>97.5</td>
<td>192.3</td>
<td>287.9</td>
</tr>
<tr>
<td><strong>FLUID</strong></td>
<td>0.5</td>
<td>1.8</td>
<td>14.1</td>
</tr>
</tbody>
</table>

802.11n WiFi (average 38 Mbps)

*Reduce VM synthesis time as little as 10 seconds!*
Overview of Optimizations

1. Minimize VM overlay size
2. Accelerate VM synthesis

Creating VM overlay (offline)

VM synthesis (runtime)

[Diagram showing the process of creating and synthesizing VM overlays with steps labeled: Launch VM, Deduplication, Reducing Semantic Gaps, VM overlay, transfer, new site, file save, Pipelining, Early Start, Launch VM, and VM overlay.]

6/27/2013
1.1 Deduplication

Approach

• Remove redundancy in the VM overlay

• Sources of redundancy
  
  1) Between *base VM* and *VM overlay*
     • Shared library copied from base disk
     • Loaded executable binary from base disk
  
  2) Between *VM overlay’s memory* and *disk*
     • Page cache, disk I/O buffer
1.1 Deduplication

1. Get the list of modified (disk, memory) chunks at the *customized VM*
2. Perform deduplication to reduce this list to a minimum
   - Compare to 1) *base disk*, 2) *base memory*, 3) other chunks within itself
   - Compare between *modified memory* and *modified disk*

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>4096</td>
<td>data</td>
</tr>
<tr>
<td>mem</td>
<td>16384</td>
<td>data</td>
</tr>
<tr>
<td>mem</td>
<td>20480</td>
<td>data</td>
</tr>
<tr>
<td>mem</td>
<td>28672</td>
<td>data</td>
</tr>
<tr>
<td>mem</td>
<td>36864</td>
<td>data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>reference</th>
<th>Data or pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>4096</td>
<td>unique</td>
<td>data</td>
</tr>
<tr>
<td>mem</td>
<td>16384</td>
<td>self</td>
<td></td>
</tr>
<tr>
<td>mem</td>
<td>20480</td>
<td>Base disk</td>
<td></td>
</tr>
<tr>
<td>mem</td>
<td>28672</td>
<td>Base memory</td>
<td></td>
</tr>
<tr>
<td>mem</td>
<td>36864</td>
<td>unique</td>
<td>data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>disk</th>
<th>0</th>
<th>unique</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>disk</td>
<td>16384</td>
<td>overlay mem</td>
<td></td>
</tr>
</tbody>
</table>

<Modified chunks>  <Overlay chunks>
1.2 Reducing Semantic Gaps

VM is a strong black box

- It ensures isolation between the host, the guest, and other guests
- But, VMM cannot interpret high-level information of memory and disk

**Example:** Download 100 MB file over network and delete it

- Ideally, it should result in no increase in VM overlay size
- However, VMM will see **200 MB of modifications:**
  - 100 MB of changed disk state
  - 100 MB of changed memory state (in-memory I/O buffer cache)

→ Let’s include **only the state that actually matters** to the guest OS
1.2 Reducing Semantic Gaps – Disk

Disk semantic gap between VMM and Guest OS

- File deletion operations only mark blocks as deleted, without discarding the contents
- VMM can’t distinguish between deleted and valid contents

Implementation: TRIM support

- ATA standard originally designed to improve SSD’s overwrite performance
- Allows an OS to inform a SSD which blocks of data are no longer in use

TRIM support at QEMU

- Modify QEMU’s IDE emulation to enable TRIM
- **Guest OS agnostic:** Linux (kernel 2.6.28), Mac OS X (June 2011), Window 7

```plaintext
time:1349399899.473041, sector_number:5244928, sector_size:16
time:1349399899.473046, sector_number:5375998, sector_size:3394
...
```
1.2 Reducing Semantic Gaps – Memory

Memory semantic gap between VMM and Guest OS

- Released memory is moved to the OS’s free page list, but is still filled with garbage
- VMM can’t distinguish between valid memory and garbage data

Approach

- No way to communicate free page information between the guest and VMM
  → scan memory snapshot

Implementation

- Insert a small agent at guest OS
  - Get memory address of the kernel data structure that has the free memory list
  - Need guest help: currently, works only in Linux
- Extract free memory pages by traversing the data structure
• Deduplication optimization reduces the VM overlay size to 44%
• Using semantic knowledge reduces the VM overlay size to 55%
• Both applied together, overlay size is reduced to **28% of baseline**
Overview of Optimizations

1. Minimize VM overlay size
2. Accelerate VM synthesis

Creating VM overlay (offline)

VM synthesis (runtime)

Launch VM

Deduplication

Reducing Semantic Gaps

VM overlay

file save

transfer

VM overlay

Pipelining

Early Start

Launch VM

new site
2.1 Pipelining

• Steps for VM synthesis
  ① Transfer VM overlay  ② Decompress  ③ Apply delta

<Sequential>

< Pipelined >

• Complexities in removing inter-dependencies among blobs
2.2 Early Start

Approach

• From user’s perspective, first response time of offloading is most important
• Starting VM even before finishing VM synthesis?

→ Do not wait until VM synthesis finishes, but start offloading immediately and process the request while synthesis is ongoing
2.2 Early Start

Implementation

1) Reorder the chunks in estimated access-order

2) Break the ordered overlay into smaller segments for demand fetching

→ Start the VM and begin streaming the segments in order, but also allow out-of-order demand fetches to preempt the original ordering
Review of Optimizations

Creating VM overlay (offline)

- Launch VM
- Deduplication
- Reducing Semantic Gaps
- VM overlay

File save

VM synthesis (runtime)

- VM overlay
- Pipelining
- Early Start
- Launch VM

New site

Transfer

[6/27/2013][intro][background][optimization][result][conclusion]
First-response time compared to baseline

Time between starting VM synthesis and receiving the first offload result

- It is faster than remote installation maintaining strong guarantees
- Except AR, we can get first-response within 10 seconds (up to 8x improvement)

* Chunks are ordered with segment size of 1 MB
Future work & Conclusion

Future work

• Open source: [http://github.com/cmusatyalab/ Elijah-cloudlet](http://github.com/cmusatyalab/ Elijah-cloudlet)
• Integrate with OpenStack (open-source cloud computing platform)

Conclusion

• Cloudlets support resource-intensive and interactive mobile apps
• Physical dispersion of cloudlets makes their provisioning a challenge
• We have shown how cloudlets can be rapidly provisioned