Special Topics:
CSci 8980 Edge Computing
Outsourcing IV

Jon B. Weissman (jon@cs.umn.edu)

Department of Computer Science
University of Minnesota
Just-in-Time Provisioning for Cyber Foraging
Thusfar

- **MAUI**
  - method-level, annotate, managed code, "when"
  - Restructure code, energy focus
- **Cuckoo**
  - late-binding to "multiple" servers, "where"
- **CloneCloud**
  - VM-based, VMs running both places, static+dynamic, "how"
- **Comet**
  - fine-grain threads, failure
- **Parametric**
  - native code, parameterized decision, "what"
- **ThinkAir**
  - Optimized "cloud" execution
Cloud Offloading

Rich, interactive applications are emerging in mobile context

- Apple’s Siri, AR apps
- Wearable devices

Cloud offloading
- These applications are too expensive to run on clients alone!
- Offload computation to a back-end server at cloud

Today’s remote cloud is a suboptimal place; high latency and limited bandwidth
Enable disconnected/trusted edge resources
Cloudlet as a Nearby Offload Site

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?
Cloudlet

Focus on deployment and infrastructure
Challenge

• To make this viable and scalable, we need an edge infrastructure (maybe 3rd party)
  – Wide-area: think mobiles and travel
  – Shared: multiple apps running on the edge
  – Enable any apps in any language in any OS + software libraries, etc.
  – Robust
    • Secure: always, but not the focus ... yet
    • Disconnected

• Need to encapsulate apps in VMs
• Granularity?
Options

• **Static provisioning**
  – Store all possible VMs on the edge nodes
  – Feasible?
  – Advantages?

• **Dynamic provisioning**
Just-in-Time 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, ...

A traveler wants to use natural language translation with speaker-trained voice recognition

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

**GOAL**: Just-in-time provisioning of a custom VM for offloading. Ideally 10 s latency (performance only)
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

VM overlay

Synthesize VM
- Decompress
- Apply delta

Resume launch VM

Backend Server in VM

Offload operations

ready
VM Synthesis - Baseline Performance

- 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)
Overview of Optimizations

1. Minimize VM overlay size
   - Creating VM overlay (offline)
     - Launch VM
     - Deduplication
     - Reducing Semantic Gaps
     - VM overlay
   - File save

2. Accelerate VM synthesis
   - VM synthesis (runtime)
     - Transfer
     - VM overlay
     - Pipelining
     - Early Start
     - Launch VM
     - New site
Deduplication

Approach

• Remove redundancy in the VM overlay
  – problem: same bits in *base VM* and *VM overlay* but in different locations in the respective images => why?

• Sources of redundancy
  
    Within *base VM*
    • Shared library copied from base disk
    • Loaded executable binary from base disk

    Between *VM overlay’s memory* and *disk*
    • Page cache, disk I/O buffer
Deduplication

1. Get the list of modified (disk, memory) chunks at the customized VM (delta).
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.

<Modified chunks>

<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>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>disk</td>
<td>0</td>
<td>data</td>
</tr>
<tr>
<td>disk</td>
<td>16384</td>
<td>data</td>
</tr>
</tbody>
</table>

<Overlay chunks>

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

Base VM

Offline operation!
Dedup Results

Figure 4: Benefit of Deduplication
Reducing Semantic Gaps

VM is a black box
- VMM cannot interpret high-level information of memory and disk

E.g: 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)

→ Include **only the state that actually matters** to the guest OS
Reducing Semantic Gaps - Disk

Disk semantic gap bet. 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

Approach

- **Exploit TRIM commands**
  - Allows an OS to inform a disk device which blocks of data are no longer in use (useful for SSDs)
  - Captured the TRIM commands so host knows about deleted data

- **File system introspection**
  - Exploit knowledge of FS disk layout to find free-map, etc.
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
- No way to communicate free page information between the guest and VMM

Approach

- Scan memory snapshot: locate frame free list data structure in kernel memory
- Requires kernel mods in guest OS (Linux only for now)
Semantic Gap Results

![Semantic Gap Results Chart]

- **OBJECT**
  - Disk memory: 26%
  - Face memory: 15%

- **FACE**
  - Disk memory: 10%
  - Face memory: 0%

- **SPEECH**
  - Disk memory: 39%
  - Face memory: 21%

- **AR**
  - Disk memory: 38%
  - Face memory: 0%

- **FLUID**
  - Disk memory: 12%
  - Face memory: 18%
- 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** ✓ Creating VM overlay (offline)

2. **Accelerate VM synthesis**

   - **Deduplication**
   - **Reducing Semantic Gaps**

   *VM synthesis (runtime)*

   - Transfer
   - Pipelining
   - Early Start
   - Launch VM

   *new site*

   VM synthesis time is still too large
Pipelining

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

- Unit of transfer: blob. How big?
- Complexities in removing inter-dependencies among blobs
Pipelining Results

Figure 9: Effect of Pipelining + Earlier Optimizations
Early Start

Idea

• 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
Early Start

Approach

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

Downside of demand fetching?
Diagram of Early Start

Mobile
- Application
- Synthesis client

Transfer VM overlay

Offload Request

VM (back-end server)

VMM (KVM)

Synthesis Server

Disk

Memory

FUSE

Base Disk

Base Memory

Overlay Disk

Overlay Memory

Filling overlay
Review of Optmizations

Creating VM overlay (offline)

- Launch VM
- Deduplication
- Reducing Semantic Gaps

VM overlay

file save

VM synthesis (runtime)

- Transfer
- VM overlay
- Pipelining
- Early Start
- Launch VM

new site
**First-response vs. baseline**

- **First-response** vs. baseline

  - Time between starting VM synthesis and receiving the first offload result:
    - It is faster than remote installation.
    - Except AR, we can get first-response within 10 seconds (up to 8x improvement).

  * Chunks are ordered with segment size of 1 MB

  - **Remote install**: Add libraries and packages to base – very error prone.
Discussion

- http://github.com/cmusatyalab/elijah-cloudlet
On Tuesday

Start Edge Architecture, Services

ECC, Cloud4Home

Have a great weekend!