Mobile App Acceleration via Fine-Grain Offloading to the Cloud

Chit-Kwan Lin

H. T. Kung

Copyright © 2014 UpShift Labs, Inc.
Confluence of Forces Points to Offloading to the Cloud

**Devices**
- Proliferation of smartphones & tablets
- Complex tasks (e.g., imaging, learning, recognition)
- Emphasis on battery efficiency

**Cloud**
- Cloud computing infrastructures
- Economies of scale
- On-demand provisioning

**Network**
- Internet infrastructure
- Wireless infrastructure (Wi-Fi & cellular)
- High bandwidths & low latencies

**Compute Offloading**
A Simple DSM Supports Offloading

Mobile App

Cloud Daemon

Distributed Shared Memory

Memory

Object

variable_X

method_Y()

Memory (replica)

Object

variable_X

method_Y()
Advantages of Dynamic + Fine-Grain

• **Dynamic**
  – Can offload arbitrary work at runtime
  – Can optimize resource utilization (e.g., battery) at runtime

• **Fine-grain**
  – At the level of a method invocation
  – Feels more responsive when failure requires local restart to fix
  – New class of small workloads for cloud providers
    • Useful for leveling out utilization while providing low-latency services for end-users
Challenges

• **Latency**
  – Dictates granularity: if update takes 5s, then workloads that run <5s don’t benefit from offloading

• **Bandwidth**
  – We should only send deltas, but determining delta encoding has non-trivial costs
    • e.g., rsync can take 3 round trips (weak hash comparison, strong hash comparison, data) to generate a delta encoding, on top of time to calculate hashes

• **Compute**
  – We should compress to save bandwidth, but compression can be computationally expensive

• **Battery**
  – Shouldn’t end up consuming more battery budget
Compressive Sensing

- Randomly mix signal elements by random projection onto lower-dimensional space
- Random $\Phi$ preserves Euclidean length/distance of sparse vectors with high probability when $M \geq O(K \log N/K)$
- Decode $x$ from $y$ by solving $y = \Phi x$ via optimization (linear programming)

$\Phi$

$M \times N$

Random sampling matrix $(M < N)$

$x$

$y$

$M \times 1$

Compressive samples

$N \times 1$

K-sparse signal
Key Insight

Writes (deltas) to memory typically constitute a sparse signal that can be compressively sampled.
Memory starts out synchronized, with byte values $x_0$

Both device and server know $\Phi$

Server calculates $y_0 = \Phi x_0$
Compressive Replication (2)

Device  \[ y_1 = \Phi x_1 \]  Server  \[ y_0 = \Phi x_0 \]

Some values in memory change, resulting in \( x_1 \)
Device calculates  \( y_1 = \Phi x_1 \)
Device transmits  \( y_1 = \Phi x_1 \) to server
Compressive Replication (3)

Server calculates $y' = y_0 - y_1$

$y' = \Phi x'$ has the same form as compressive sensing decoding problem, so server solves for $x'$

$x' = x_0 - x_1$ is the delta encoding!
Server calculates $x_0 - x' = x_1$, which is the new memory state

Server is now updated
Novel Characteristics

• **All-in-one**
  – Delta encoding + compression

• **Delta encoding figured out by server, not device**
  – Automatically recovered during decoding
  – Just send compressive samples; no add’l network costs

• **Codec is resource-commensurate**
  – Device: low-complexity encoder
  – Server: higher-complexity decoder
  – Unlike traditional compressors
What do we compare against?

- **No similar replication methods**
  - Compressive replication gives all-in-one delta encoding + compression
  - e.g., rsync
    - Compression is an add’l step
    - Needs multiple round-trips to determine delta encoding

- **Compressed snapshots is a more appropriate comparison**
  - No add’l round-trip overheads
  - Just compress the whole memory page (snapshot)
  - snappy, zlib

- **Metrics**
  - Replicate 64KB memory block
  - Total Latency = encoding + network + decoding
  - Compression ratio (bandwidth cost)
## Latency/Compression Trade-off

<table>
<thead>
<tr>
<th>Method</th>
<th>Encoding (ms)</th>
<th>Network (ms)</th>
<th>Decoding (ms)</th>
<th>Total Latency (ms)</th>
<th>Compression Ratio</th>
<th>Update Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>snappy</td>
<td>4</td>
<td>15</td>
<td>--</td>
<td>19</td>
<td>3.8 : 1</td>
<td>17.2</td>
</tr>
<tr>
<td>zlib</td>
<td>487</td>
<td>13</td>
<td>--</td>
<td>500</td>
<td>6.0 : 1</td>
<td>10.9</td>
</tr>
<tr>
<td>compressive replication</td>
<td>53</td>
<td>12</td>
<td>70</td>
<td>135</td>
<td>7.3 : 1</td>
<td>9.0</td>
</tr>
</tbody>
</table>

**Near ~100ms threshold of user-perceptible app delay**

**Highest compression ratio**
### Latency/Compression Trade-off

<table>
<thead>
<tr>
<th>Method</th>
<th>Encoding (ms)</th>
<th>Network (ms)</th>
<th>Decoding (ms)</th>
<th>Total Latency (ms)</th>
<th>Compression Ratio</th>
<th>Update Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>snappy</td>
<td>4</td>
<td>15</td>
<td>--</td>
<td>19</td>
<td>3.8 : 1</td>
<td>17.2</td>
</tr>
<tr>
<td>zlib</td>
<td>487</td>
<td>13</td>
<td>--</td>
<td>500</td>
<td>6.0 : 1</td>
<td>10.9</td>
</tr>
<tr>
<td>compressive replication</td>
<td>53</td>
<td>12</td>
<td>70</td>
<td>135</td>
<td>7.3 : 1</td>
<td>9.0</td>
</tr>
</tbody>
</table>

Unlike snappy/zlib, comp ratio fixed & b/w cost predictable because $y = \Phi x$ is independent of input’s Kolmogorov complexity

Remains good w/ high Kolmogorov complexity, which would confound snappy/zlib
Example Handwriting Recognition App

- **UpShift platform for compressive offloading** of iOS apps
- **SVM** for Chinese handwriting recognition
  - Stroke count is measure of task complexity
- **Device:** iPad (3rd gen)
- **Server:** Amazon g2.xlarge
  - GPU for compressive replication decoding
  - CPU for SVM evaluation
- **Network:** 802.11g
  - Office setting
  - 19ms RTT to us-east-1a server
On-device execution time scales poorly with task complexity.

Offloaded execution time stays short due to low overhead of compressive offloading. Users expect this.

More complex tasks have greater speedup.
Battery Savings

High task complexity (25 strokes), 250 iterations

Offloading allows 60% more work to be done with the same battery budget.
Summary

- Low-latency DSM updates enable fine-grain offloading
- UpShift supports offloading ~100ms workloads, while keeping resource utilization low
- Achieves significant speedups and battery savings
- Key insight: memory writes are a sparse signal that can be compressively sampled
- Implications for future ARM-based DC’s or x86-based devices
Questions?

ck@upshiftlabs.com