COMET: Code Offload by Migrating Execution Transparently

Mark Gordon, Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen

Presented by Janani Thirunavukkarasu

Csci 8980: Mobile Cloud Computing

**Some of the contents have been taken from the Speaker’s presentation at OSDI 2012**
Overview

- Introduction
- Distributed Shared Memory
- COMET Design
- Evaluation
- Summary
What is COMET?

Distributed Runtime Environment for near transparent offloading from smartphones
What is offloading?

- Mobile devices
  - Have limited resources
  - Are well connected

- Name the systems/methods we’ve seen thus far to bring network resources to mobile?

Can they be grouped based on operations or execution models?
Related Work

- CloudLet – optimized server discoverability
- Weblets – part of cloud aware applications
- Offloading Frameworks – Capture and Migrate
  - Cuckoo: first progg construct
  - MAUI: Annotation based
  - CloneCloud: Static Analysis
- COMET extends of MAUI/CloneCloud systems and DSM based systems like Jessica
What is COMET? (Continued)

- How is COMET different?
  - Thread and synchronization support
  - Offload part of methods

- How does COMET achieve this?
  - DSM techniques
  - VM Synchronisation
COMET's Goals

1. Improve mobile computation speed
2. Require no programmer effort
3. Generalize well with existing applications
4. Resist network failures
Overview

- Introduction
- Distributed Shared Memory
- COMET Design
- Evaluation
- Summary
Distributed Shared Memory

- COMET is offloading + DSM
  - Offloading bridges computation disparity
  - DSM provides logically shared address space
- DSM usually applied to cluster environments
  - Low latency, high throughput
- Mobile relies on wireless communication
DSM (continued)

- Conventional DSM (Munin)

  - Waited an RTT for a write
  - Read could take RTT also
Java Memory Model

- Dictates which writes a read can observe
- Specifies 'happens-before' partial order
  - Within single thread accesses are totally ordered
  - Across threads Lazy Release Consistency locking

Can you think of any other problems here?
Field DSM

- Fundamental memory unit is the field
  - Known alignment, known width
- Track dirty fields locally
- Need 'happens-before' established?
  - Transmit dirty fields! (mark fields clean)
- Design discussion
  - Any limitations?
  - Which techniques would you use on a cluster DSM? OR Message Passing?
Overview

- Introduction
- Distributed Shared Memory
- COMET Design
- Evaluation
- Summary
VM-synchronization

- Used to establish 'happens-before' relation
- Directed operation between pusher and puller
- Synchronizes
  - Bytecode sources
  - Java thread stacks
  - Java heap
Bytecode Update (Step 1 of 3)

Operation begins by sending any new code

Pusher

I load $xyz$.dex

Send $xyz$.dex

Puller
Stack Update (Step 2 of 3)

- Next we send over thread stacks

```
Thread id: 2
job2::run
  pc: 5
  registers[42, 555, 0]
workLoop
  pc: 6
  registers[0, [obj:9]]
start
  pc: 3
  Registers[101, [obj:9]]
```

Pusher

Puller
Heap Update (Step 3 of 3)

- Finally send over heap update
  - We send updates to any changed (or new) field
  - Only send updates of 'shared' heap

```
[obj:2].y = 1
[obj:4].z = [obj:3]
...
```
Lock ownership

- Annotate with lock ownership flag
- Establish 'happens-before' with VM-sync
Thread Migration

- Thread migration trivial
  - Push VM-sync
  - Transfer lock ownership

Pusher

Puller
Native Methods

- Written in C with bindings for Java
  - Math.sin(), OSFileSystem.write(), VMThread.currentThread()

- Native methods exist to
  - Access device resources (file system, display, etc)
  - For performance reasons
  - To work with existing libraries

- Not generally safe to run on either endpoint
  - Manually white list safe native methods
Failure Recovery

- VM-synchronization is recovery safe
- Always leave enough information on client
- If server is lost resume threads running locally!
Tau-Scheduler

\[ T = 2 \times \text{VM-synchronization time} \]
Implementation

- Built from gingerbread CyanogenMod source
- ~5000 lines of C code
- JIT not included

```
Engine.c:offMigrateThread()
offWriteU1(self, OFF_ACTION_MIGRATE);
deactivate(self);
offThreadWaitForResume(self);
```
Overview

- Introduction
- Distributed Shared Memory
- COMET Design
- Evaluation
- Summary
Benchmarks

- 8 applications from Google Play
  - Average speed-up of 2.88X on WiFi / 1.28X on 3G
  - Average energy saving of 1.51X on WiFi / 0.84X on 3G

- 2 computation benchmark applications
  - 10.4X speed-up w/ WiFi on Linpack
  - 500+X speed-up w/ multi-threaded factoring
MacroBenchmarks

The image shows a bar chart comparing energy usage across different benchmarks and network conditions (Baseline, WiFi, 3G). The x-axis represents various benchmarks including Calculus, Chess, Edgedetect, Fractal, Metro, Photoshop, Poker, Sudoku, Linpack, and Factor. The y-axis represents energy usage in Joules (J). The chart highlights the energy consumption differences under Baseline and 3G network conditions for each benchmark.
Rhino

- Java JavaScript Interpreter
- Ran with SunSpider JavaScript benchmark
Overview

- Introduction
- Distributed Shared Memory
- COMET Design
- Evaluation
- Summary
Summary

- **Positives**
  - The notion of DSM
  - Google play applications were used for evaluation. Not home grown applications well suited to the infrastructure.
  - Correctness of multi thread applications if guaranteed by maintaining “happens-before” relationship
  - Network failure recovery model is simple and efficient

- **Negatives**
  - Not enough information regarding when the server VM is initialized.
  - Good offloading mechanism but can’t be extended to heterogenous hardware, different languages or >1 server or clusters. Not flexible in current state
  - Real data was not used for testing. Can’t be sure of user workloads.
Questions?