Swift: A Register-based JIT Compiler for Embedded JVMs

Yuan Zhang, Min Yang, Bo Zhou, Zhemin Yang, Weihua Zhang, Binyu Zang

Fudan University

Eighth Conference on Virtual Execution Environment (VEE 2012)
DEX: a new Java bytecode format

Android platform
➢ Built in Java language
➢ Using Java to develop applications
➢ Dalvik Virtual Machine, support Android applications

DEX: bytecode format in Android
➢ Register-based bytecode format
➢ Not compatible with traditional stack-based bytecode
➢ \textit{dx}: a tool to transform traditional bytecode to DEX
**DX: translation tool**

**DEX: bytecode format in Android**
- Register-based bytecode format
- Not compatible with traditional stack-based bytecode
- `dx`: a tool to transform traditional bytecode to DEX
Traditional Bytecode versus DEX

Traditional bytecode

- Stack-based bytecode, widely supported
- All operations are aided by a *virtual stack*
- E.g. *iadd* instruction for integer addition

DEX: Android bytecode

- Register-based, becoming popular with Android
- Each method has unlimited virtual registers
- Each instruction can directly reference any register
Why register-based bytecode format?

First proposed by Davis et al. [IVME’03]

- reduce instruction count by 34.9%
- increase bytecode size by 44.9%
Why register-based bytecode format?

First proposed by Davis et al. [IVME’03]
- reduce instruction count by 34.9%
- increase bytecode size by 44.9%

Impact on VM Interpreter
- Virtual machine showdown: stack vs register [VEE’05]
- reduce execution time by 26.5% on a C interpreter
Why register-based bytecode format?

First proposed by Davis et al. [IVME’03]
- reduce instruction count by 34.9%
- increase bytecode size by 44.9%

Impact on VM Interpreter
- Virtual machine showdown: stack vs register [VEE’05]
- reduce execution time by 26.5% on a C interpreter

Impact on JIT Compilers
- Unknown yet, this paper’s topic
JIT-Droid, Google's JIT Compiler

- Register-based bytecode
  - CFG Construction
  - SSA Conversion
  - Register Allocation
  - Code Generation
  - Low-IR Generation

- Register-based binary
JIT-Droid, Google’s JIT Compiler

Register-based bytecode

Register-based binary

CFG Construction

SSA Conversion

Register Allocation

Low-IR Generation

Long Pipeline!!

Code Generation
JIT-Droid, Google’s JIT Compiler

Register-based bytecode

Register-based binary

CFG Construction

SSA Conversion

Register Allocation

Low-IR Generation

Code Generation

Long Pipeline!!

Question: How to exploit the homogeneity between register-based bytecode and register-based machine code?
JIT-Droid, Google's JIT Compiler

Strategy: Why not straightforward translation?
JIT-Droid, Google's JIT Compiler

**Strategy:** Why not straightforward translation?

**Challenge:** How to guarantee code quality with fast compilation speed?
Outline

Java Method Characteristics

Register-based JIT

Our Prototype

Evaluation Results

Conclusion
Java Method Characteristics

How many registers are enough for most methods?

- Most Java methods are small
- Each method handle one specific logic

Experiment

- Record all the methods executed and their count

Observation

1. More than 90% Java methods use less than 11 virtual registers.
2. Almost all embedded processors feature more than 11 registers.
Outline

Java Method Characteristics

Register-based JIT

Our Prototype

Evaluation Results

Conclusion
Swift

Perform near-optimal register-allocation, and heavy optimizations
Swift

Perform near-optimal register-allocation, and heavy optimizations

Class Loader → Compile Stub → Recompile Stub

Dynamic Translator

Register Mapping → Code Selector → Thread-local Code Cache

Global Shared Code Cache

Thread Manager, Exception Handling, Garbage Collection

.Dex

class loader

Dalvik Virtual Machine
Register-Mapping Table

Regular Method

- **Def**: all virtual regs. can be mapped to physical regs.
- 1-1 mapped between virtual regs. and physical regs.

Irregular Method

- **Def**: more virtual regs. than available physical regs.
- Some virtual regs are mapped to spill area in stack
- 1-1 mapped between virtual regs. and physical regs. or spill area location
Template-based Code Selector

Generate code by traverse DEX Instruction

*Computation Instruction*
- 189/232, such as addition, division, subtraction, etc
- Easy to find corresponding machine instruction

*VM-Related Instruction*
- 43/232, such as object lock operation, object creation
- Generate call to VM function

*Handle Spill Area*
- Generate load instr. Before read
- Generate store instr. After write
Outline

Java Method Characteristics

Register-based JIT

Our Prototype

Evaluation Results

Conclusion
Swift on ARM

Instruction Set

- ARM, 32 bits, support by all variants
- Thumb, 16 bits, support by armv6
- Thumb2, 16-32 bits mixed, support by armv7 or higher

Physical Registers

- 16 general purpose registers
- r13-stack register, r14-link register, r15-program counter
- remain 13 free registers, {r0-r12}
### Translation Example

#### Regular Method

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>000</strong>: <code>const/4 v0, #0</code></td>
<td><strong>0000</strong>: <code>mov r3, #0</code></td>
<td></td>
</tr>
<tr>
<td><strong>001</strong>: <code>move v1, v3</code></td>
<td><strong>0004</strong>: <code>mov r4, r1</code></td>
<td></td>
</tr>
<tr>
<td><strong>002</strong>: <code>if-ge v1, v4, 008</code></td>
<td><strong>0008</strong>: <code>cmp r4, r2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>000b</strong>: <code>bge 001b</code></td>
<td></td>
</tr>
<tr>
<td><strong>004</strong>: <code>add-int/2addr v0, v1</code></td>
<td><strong>0010</strong>: <code>add r3, r3, r4</code></td>
<td></td>
</tr>
<tr>
<td><strong>005</strong>: <code>add-int/lit8 v1, v1, #1</code></td>
<td><strong>0014</strong>: <code>add r4, r4, #1</code></td>
<td></td>
</tr>
<tr>
<td><strong>007</strong>: <code>goto 002</code></td>
<td><strong>0018</strong>: <code>b 0008</code></td>
<td></td>
</tr>
</tbody>
</table>

#### Irregular Method

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>000</strong>: <code>add-int/lit8 v15, v15, #1</code></td>
<td><strong>0000</strong>: <code>ldr r10, [sp, #12]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>0004</strong>: <code>add r10, r10, #1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>0008</strong>: <code>str r10, [sp, #12]</code></td>
<td></td>
</tr>
</tbody>
</table>
Code Unloader

Unloading Strategies (Zhang et al. LCTES’04, PPPJ’04)

- **Good Strategy**: precisely select unload candidate
- **Drawback**: complex the design, adds runtime overhead

Unload Strategy in Swift

- A simple but maybe imprecise strategy
- Mark all methods on the stack at GC time
- Unload those methods unmarked twice
Lightweight Optimizations

Optimization for \textit{Irregular Method}

- \textbf{Bad Scenario:} frequently referenced variable is mapped to stack area
- \textbf{Solution:} detect all the loops and map virtual registers in the loop to physical registers first

Optimization for \textit{interface-call}

- \textit{interface-call} is heavy
- \textbf{Solution:} use a class-test to exploit the object type locality at the call-site
Outline

Java Method Characteristics

Register-based JIT

Our Prototype

Evaluation Results

Conclusion
Experimental Environment

Hardware Platform

<table>
<thead>
<tr>
<th>ARM Chip</th>
<th>CPU Feature</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3C6410</td>
<td>Armv6, 800MHz</td>
<td>16KB I-Cache, D-Cache</td>
</tr>
<tr>
<td>OMAP3530</td>
<td>Armv7, 600MHz</td>
<td>16KB I-Cache, D-Cache; 256KB L2 Cache</td>
</tr>
</tbody>
</table>

Benchmarks

- SPECjvm98, JemBench2, EmbeddedCaffeineMark3

Software Platform

- Swift, Android 2.1
- Fast Interpreter, Android 2.3.4
- JIT-Droid, Android 2.3.4
Performance-with Fast Interpreter

Compared with Fast Interpreter

Performance Ratio

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Performance Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>J Bench2/armv6</td>
<td>4.734</td>
</tr>
<tr>
<td>J Bench2/armv7</td>
<td>4.474</td>
</tr>
<tr>
<td>JVM98/armv6</td>
<td>1.613</td>
</tr>
<tr>
<td>JVM98/armv7</td>
<td>1.755</td>
</tr>
<tr>
<td>ECM3/armv6</td>
<td>4.180</td>
</tr>
<tr>
<td>ECM3/armv7</td>
<td>3.716</td>
</tr>
<tr>
<td>GEO.MEAN</td>
<td>3.13</td>
</tr>
</tbody>
</table>

30
Performance-with JIT-Droid

Compared with JIT-Droid

<table>
<thead>
<tr>
<th>Sample</th>
<th>Performance Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>JBench2/armv6</td>
<td>1.746</td>
</tr>
<tr>
<td>JBench2/armv7</td>
<td>1.385</td>
</tr>
<tr>
<td>JVM98/armv6</td>
<td>1.423</td>
</tr>
<tr>
<td>JVM98/armv7</td>
<td>1.266</td>
</tr>
<tr>
<td>ECM3/armv6</td>
<td>1.545</td>
</tr>
<tr>
<td>ECM3/armv7</td>
<td>1.214</td>
</tr>
<tr>
<td>GEO.MEAN</td>
<td>1.42</td>
</tr>
</tbody>
</table>
Performance-with Swift/no-opt

Compared with Swift/no-opt

Performance Ratio

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Performance Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>JBench2/armv6</td>
<td>1.019</td>
</tr>
<tr>
<td>JBench2/armv7</td>
<td>1.011</td>
</tr>
<tr>
<td>JVM98/armv6</td>
<td>1.034</td>
</tr>
<tr>
<td>JVM98/armv7</td>
<td>1.013</td>
</tr>
<tr>
<td>ECM3/armv6</td>
<td>1.071</td>
</tr>
<tr>
<td>ECM3/armv7</td>
<td>1.046</td>
</tr>
<tr>
<td>GEO.MEAN</td>
<td>1.03</td>
</tr>
</tbody>
</table>

32
## Translation Time

Table 1: Translation Time of Swift on OMAP3530

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Trans. Time(s)</th>
<th>Exec. Time(s)</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjvm98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>0.117</td>
<td>1.613</td>
<td>0.128%</td>
</tr>
<tr>
<td>jess</td>
<td>0.185</td>
<td>77.924</td>
<td>0.237%</td>
</tr>
<tr>
<td>db</td>
<td>0.124</td>
<td>64.753</td>
<td>0.191%</td>
</tr>
<tr>
<td>javac</td>
<td>0.274</td>
<td>113.124</td>
<td>0.243%</td>
</tr>
<tr>
<td>mtrt</td>
<td>0.178</td>
<td>66.280</td>
<td>0.268%</td>
</tr>
<tr>
<td>jack</td>
<td>0.175</td>
<td>87.321</td>
<td>0.201%</td>
</tr>
<tr>
<td>ECM3</td>
<td>0.098</td>
<td>23.930</td>
<td>0.409%</td>
</tr>
<tr>
<td>JemBench2</td>
<td>0.092</td>
<td>27.400</td>
<td>0.334%</td>
</tr>
</tbody>
</table>

Swift costs no more than 0.3s to translate all the methods in each case, occupying less than 0.5% of total execution time.
## Translation Time Comparison

Table 2: Translation Time of Swift and JIT-Droid

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Swift(s)</th>
<th>JIT-Droid(s)</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjvm98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>0.117</td>
<td>0.257</td>
<td>45.5%</td>
</tr>
<tr>
<td>jess</td>
<td>0.185</td>
<td>0.850</td>
<td>21.8%</td>
</tr>
<tr>
<td>db</td>
<td>0.124</td>
<td>0.270</td>
<td>45.9%</td>
</tr>
<tr>
<td>javac</td>
<td>0.274</td>
<td>2.638</td>
<td>10.4%</td>
</tr>
<tr>
<td>mtrt</td>
<td>0.178</td>
<td>0.948</td>
<td>18.8%</td>
</tr>
<tr>
<td>jack</td>
<td>0.175</td>
<td>1.154</td>
<td>15.2%</td>
</tr>
<tr>
<td>ECM3</td>
<td>0.098</td>
<td>0.433</td>
<td>22.6%</td>
</tr>
<tr>
<td>JemBench2</td>
<td>0.092</td>
<td>2.184</td>
<td>4.2%</td>
</tr>
</tbody>
</table>
### Code Size

Table 3: Translated Code Size of Swift on OMAP3530

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Unload On(KB)</th>
<th>Unload Off(KB)</th>
<th>Save Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjvm98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>122.442</td>
<td>313.229</td>
<td>60.9%</td>
</tr>
<tr>
<td>jess</td>
<td>154.969</td>
<td>549.314</td>
<td>71.8%</td>
</tr>
<tr>
<td>db</td>
<td>104.468</td>
<td>336.174</td>
<td>68.9%</td>
</tr>
<tr>
<td>javac</td>
<td>484.338</td>
<td>875.173</td>
<td>44.7%</td>
</tr>
<tr>
<td>mtrt</td>
<td>142.130</td>
<td>443.936</td>
<td>68.0%</td>
</tr>
<tr>
<td>jack</td>
<td>212.583</td>
<td>577.368</td>
<td>63.2%</td>
</tr>
<tr>
<td>ECM3</td>
<td>150.483</td>
<td>251.656</td>
<td>40.2%</td>
</tr>
<tr>
<td>JemBench2</td>
<td>193.340</td>
<td>233.205</td>
<td>17.1%</td>
</tr>
</tbody>
</table>

The code unloader saves **50.1%** code space in average, and it has only **3.9%** performance degradation (*see our paper*).
Outline

Java Method Characteristics

Register-based JIT

Our Prototype

Evaluation Results

Conclusion
Contribution

A study on Java method characteristics

- More than 90% methods use less than 11 registers

Propose an efficient & effective JIT compiler for register-based bytecode

- Register mapping & straightforward translation

Evaluate proposed JIT in Android system

- OMAP3530, S3C6410
- SPECjvm98, JemBench2, EmbeddedCaffeineMark3
- \(42\%\) faster than default Android JIT compiler
Discussion

Register-based versus stack-based

- Complement of previous research [IVME’03, VEE’05]

Register-based JIT Compiler

- Embedded JIT, non-optimizing compiler

Register-based bytecode

- Responsibility division between offline static compiler and online dynamic compiler
- Balance between AOT Compiler and JIT Compiler
Q & A

Parallel Processing Institute

http://ppi.fudan.edu.cn