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

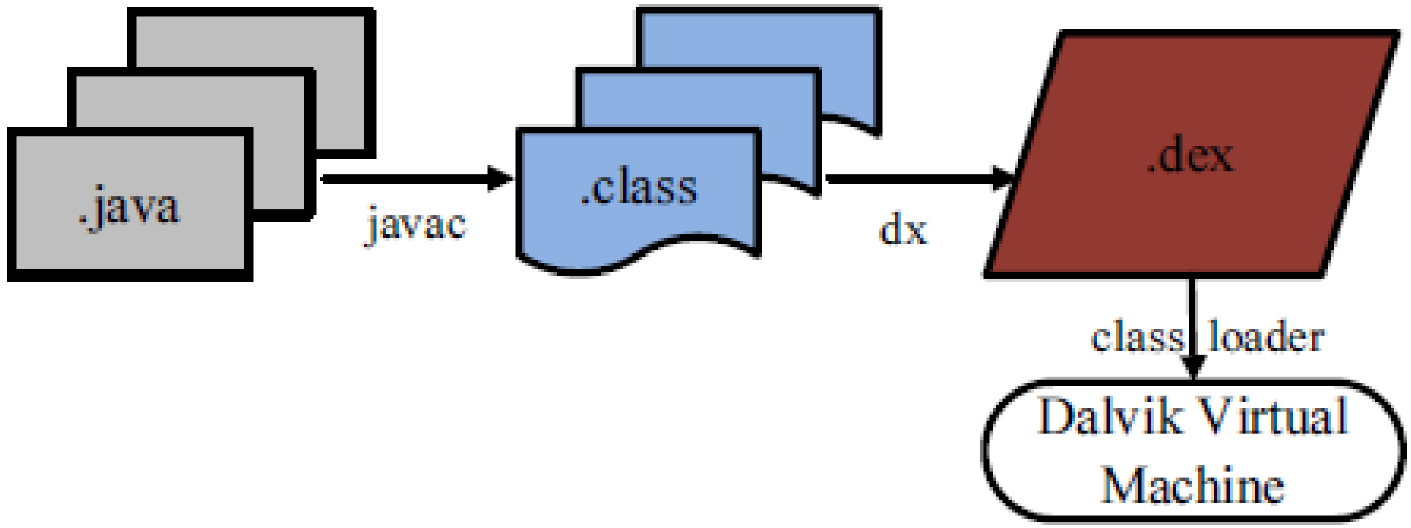
Android platform

- 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
- ***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%

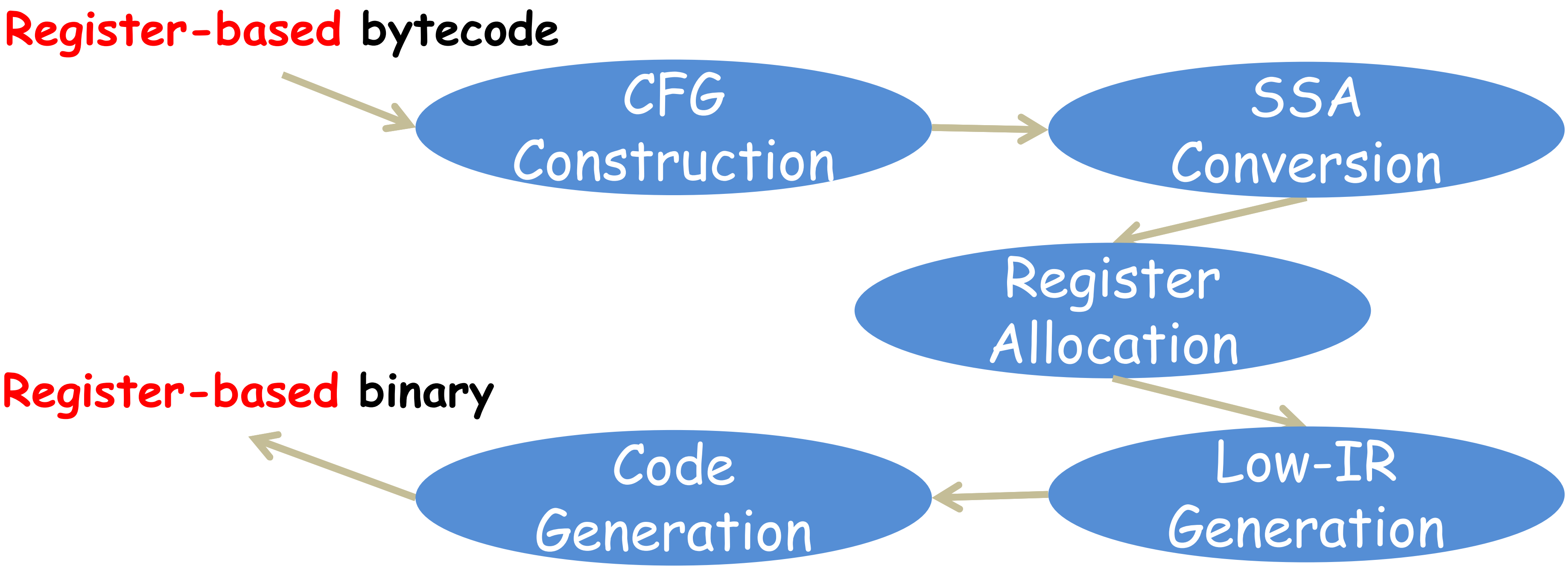
Impact on VM Interpreter

- Virtual machine showdown: stack vs register [VEE'05]
- reduce execution time by **26.5%** on a **C interpreter**

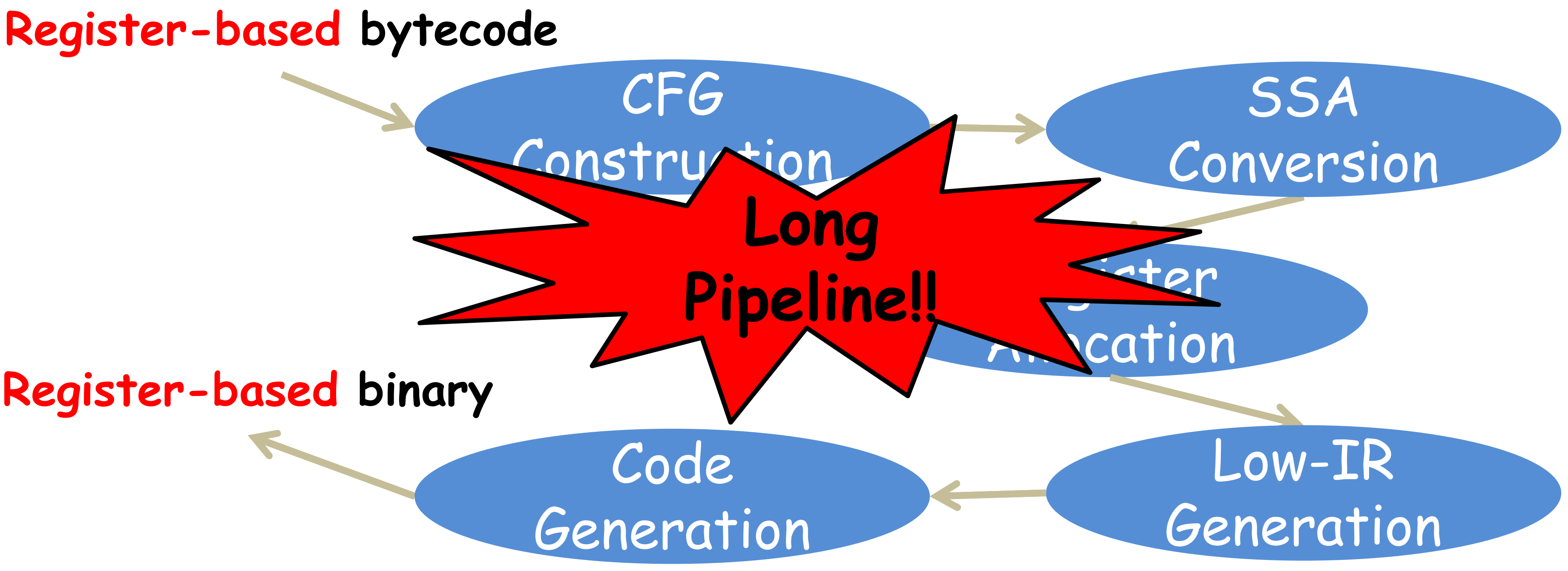
Impact on JIT Compilers

- Unknown yet, this paper's topic

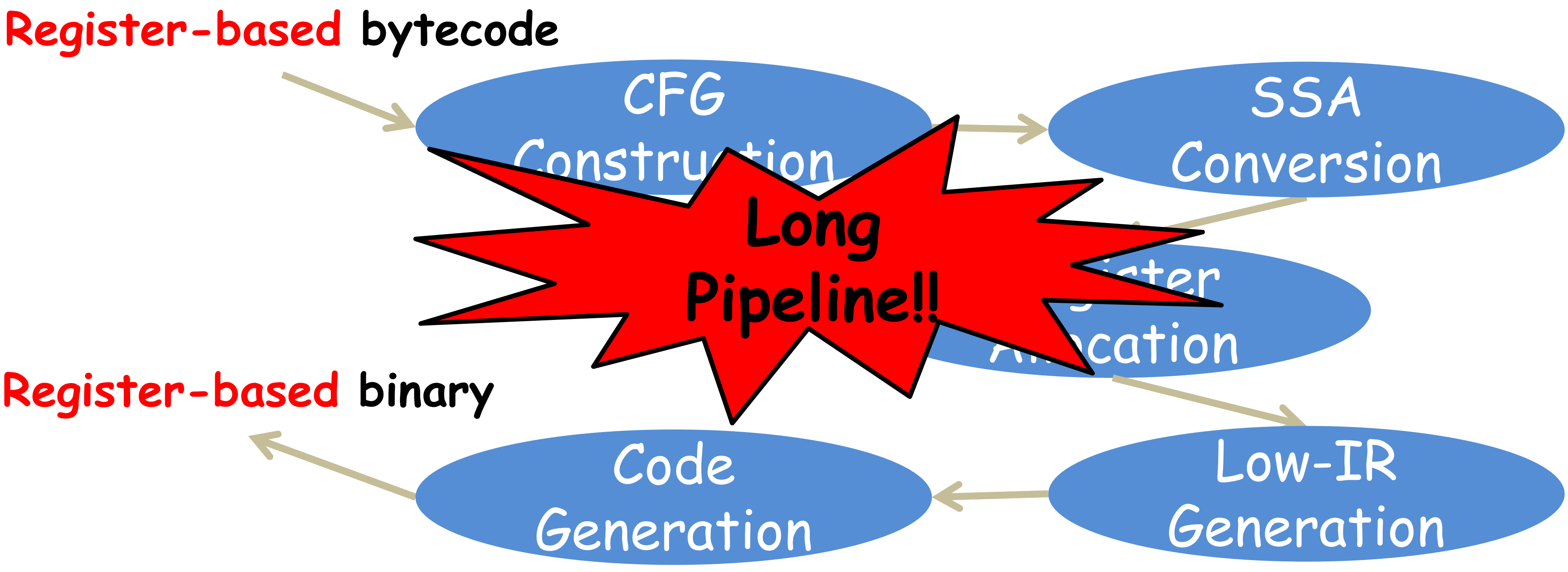
JIT-Droid, Google's JIT Compiler



JIT-Droid, Google's JIT Compiler

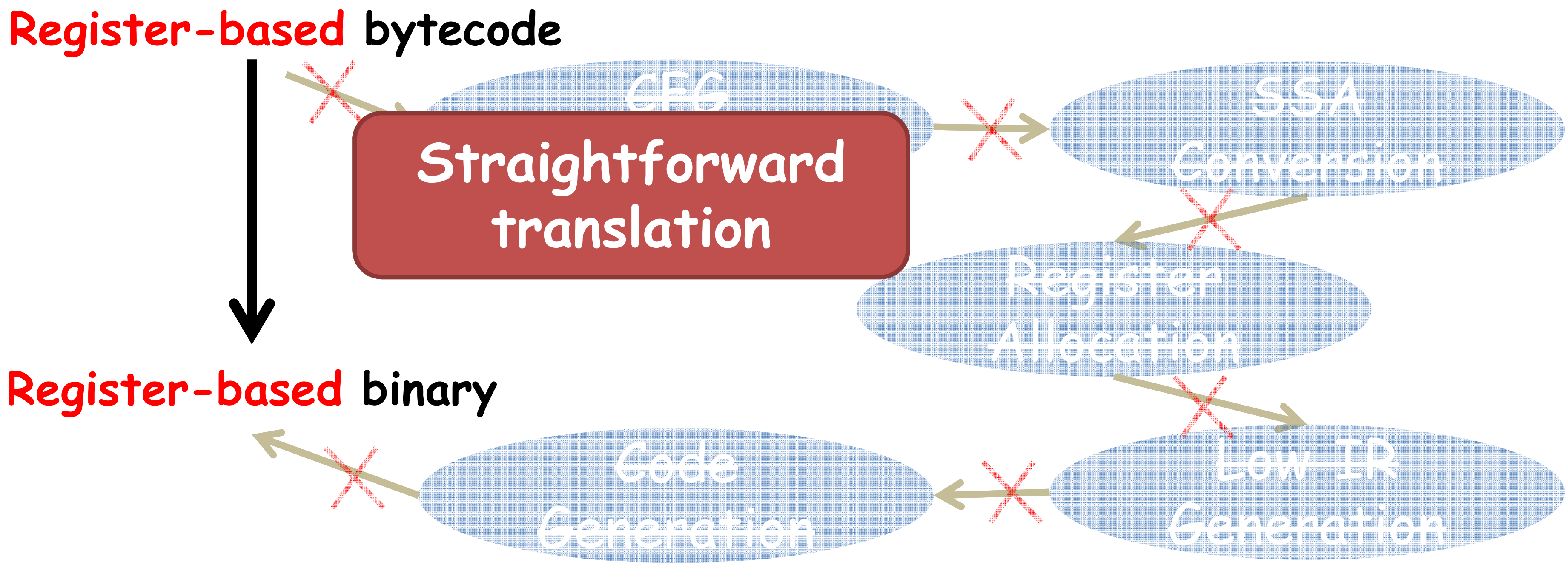


JIT-Droid, Google's JIT Compiler



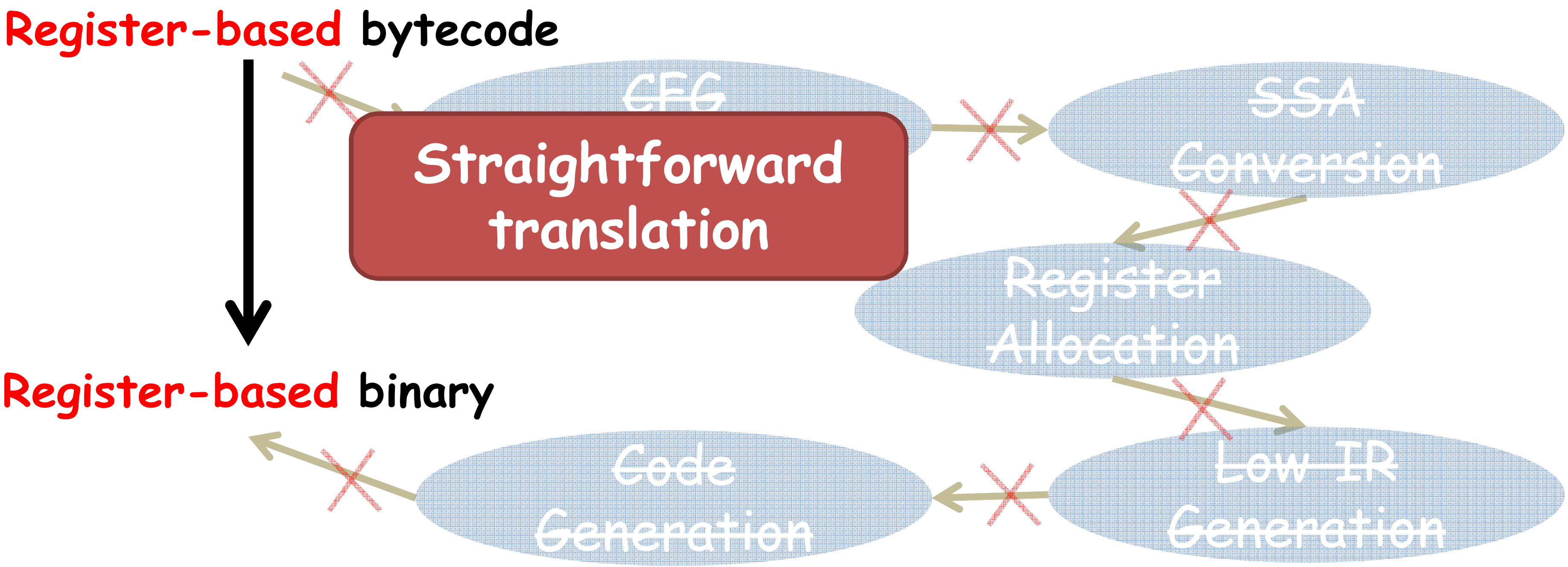
Question: How to exploit the homogeneity between register-based bytecode and register-based machine code?

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

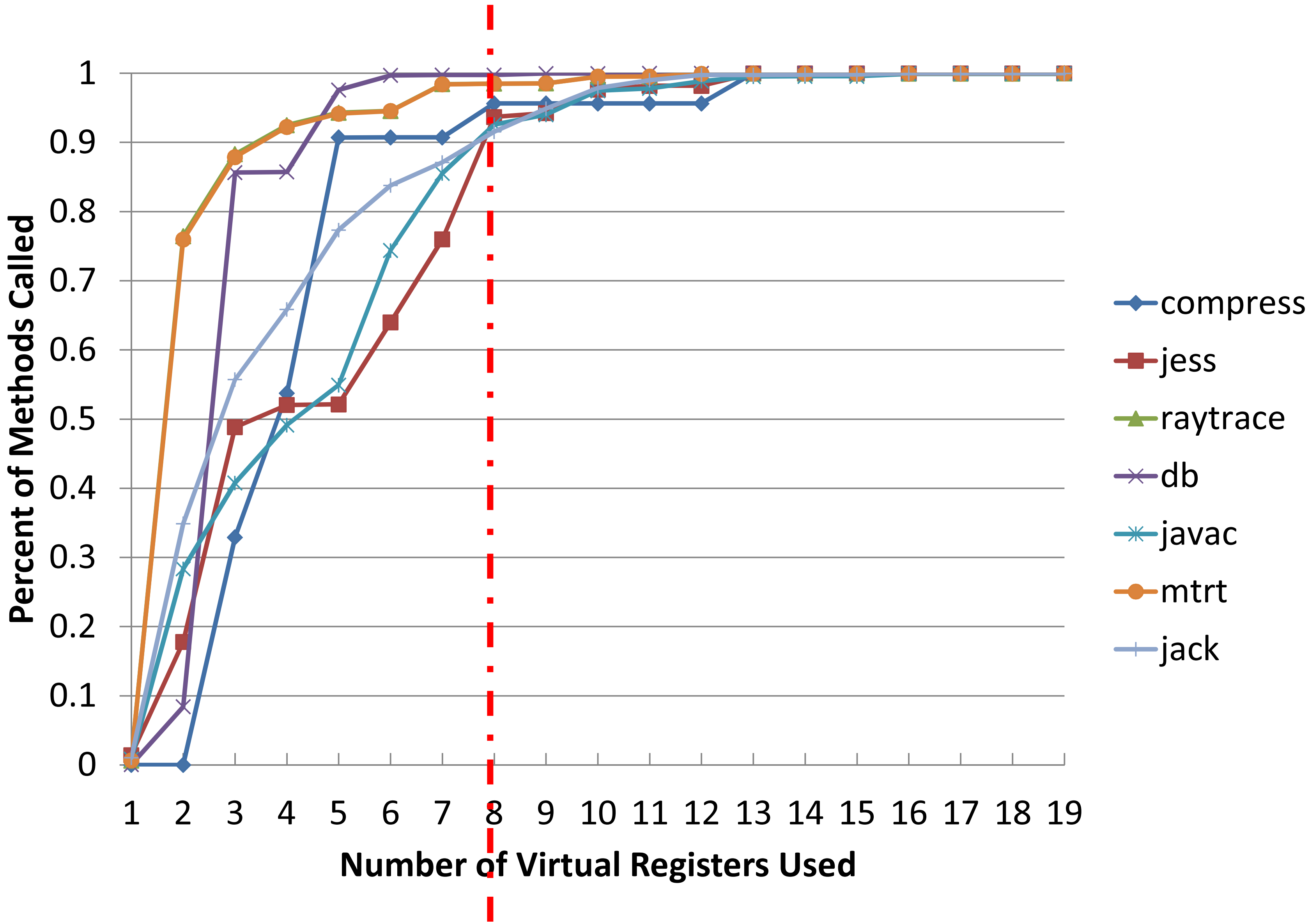
How many registers are enough for most methods?

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

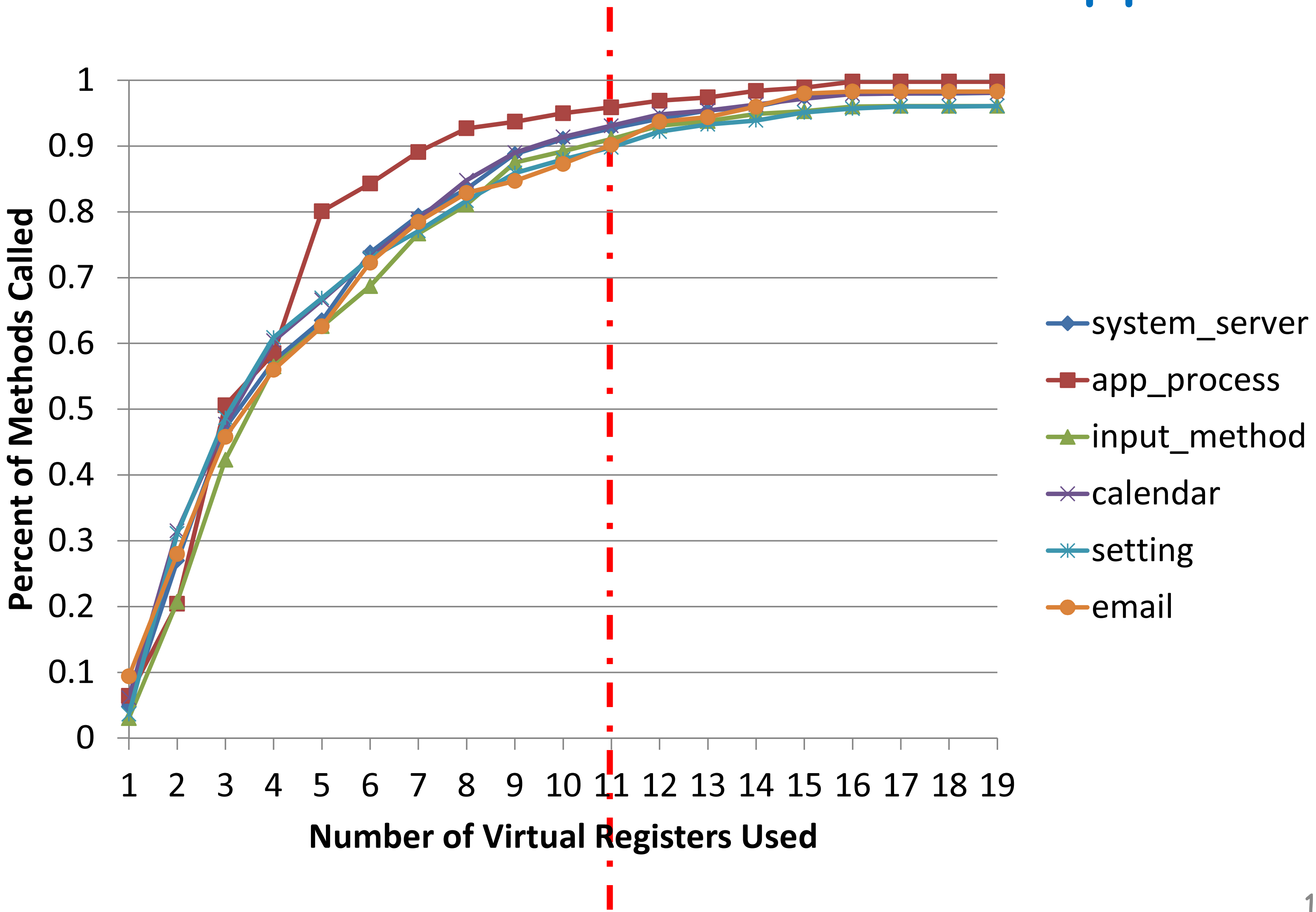
<u>Experiment</u>

- Record all the methods executed and their count
- *Benchmarks*: SPECjvm98 & real Android App.

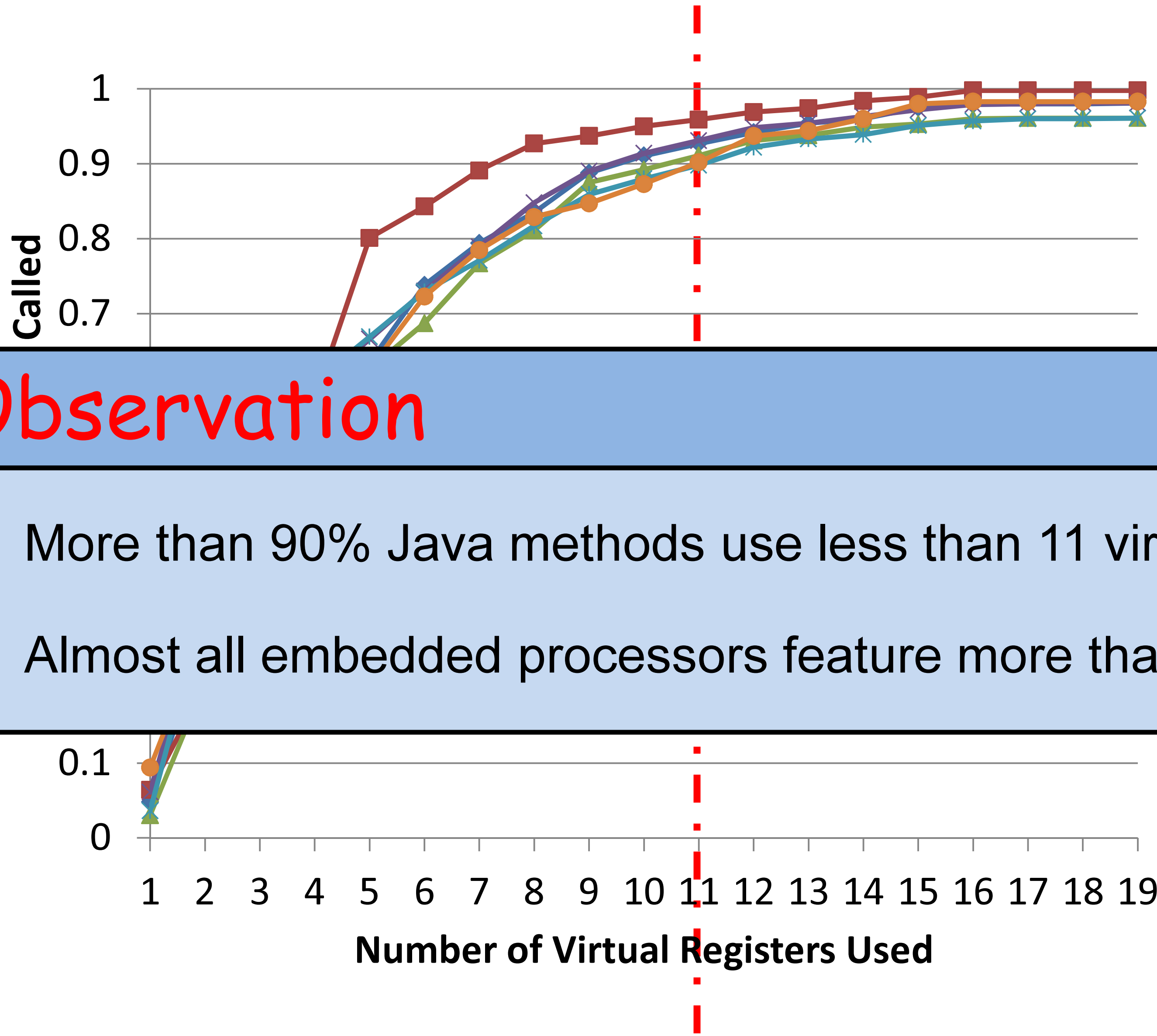
Java Method Characteristics-JVM98



Java Method Characteristics-App.



Java Method Characteristics-App.



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

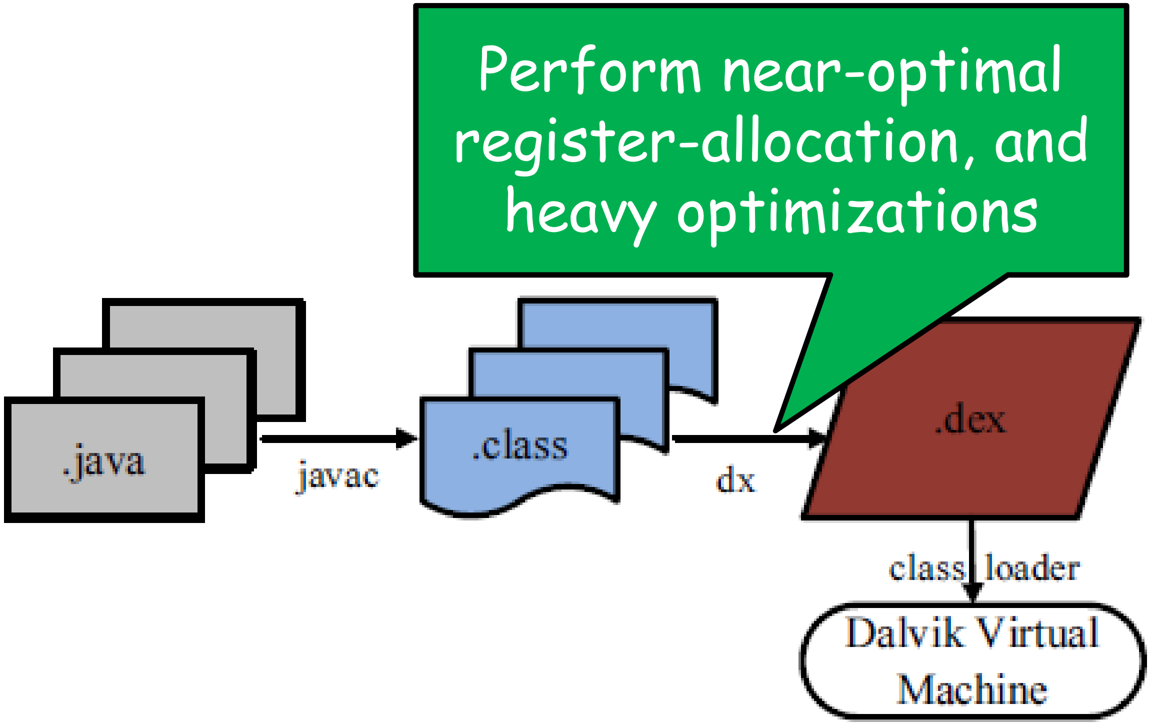
Register-based JIT

Our Prototype

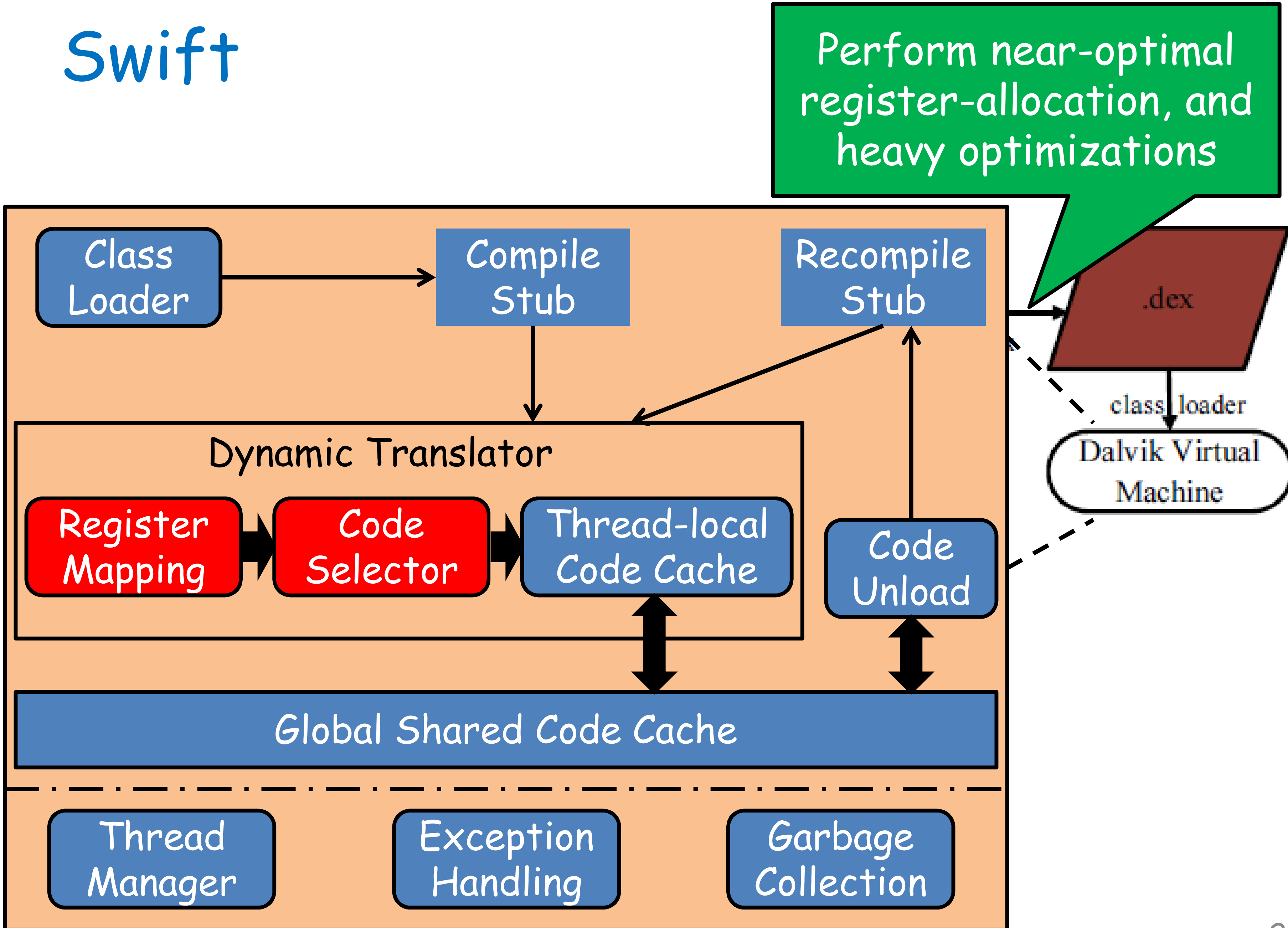
Evaluation Results

Conclusion

Swift


Perform near-optimal register-allocation, and heavy optimizations
.java
javac
.class
dx
.dex
class loader
Dalvik Virtual Machine

Swift


Perform near-optimal register-allocation, and heavy optimizations
Class Loader
Compile Stub
Recompile Stub
.dex
class loader
Dalvik Virtual Machine
Dynamic Translator
Register Mapping
Code Selector
Thread-local Code Cache
Code Unload
Global Shared Code Cache
Thread Manager
Exception Handling
Garbage Collection

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

000 : const/4 v0, #0	0000: mov r3, #0
001 : move v1, v3	0004: mov r4, r1
002 : if-ge v1, v4, 008	0008: cmp r4, r2 000b: bge 001b
004 : add-int/2addr v0, v1	0010: add r3, r3, r4
005 : add-int/lit8 v1, v1, #1	0014: add r4, r4, #1
007 : goto 002	0018: b 0008

Irregular Method

000 : add-int/lit8 v15, v15, #1	0000: ldr r10, [sp, #12] 0004: add r10, r10, #1 0008: str r10, [sp, #12]

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 *Irregular Method*

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

Optimization for *interface-call*

- *interface-call is heavy*
- **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

ARM Chip	CPU Feature	Other
S3C6410	Armv6, 800MHz	16KB I-Cache, D-Cache
OMAP3530	Armv7, 600MHz	16KB I-Cache, D-Cache; 256KB L2 Cache

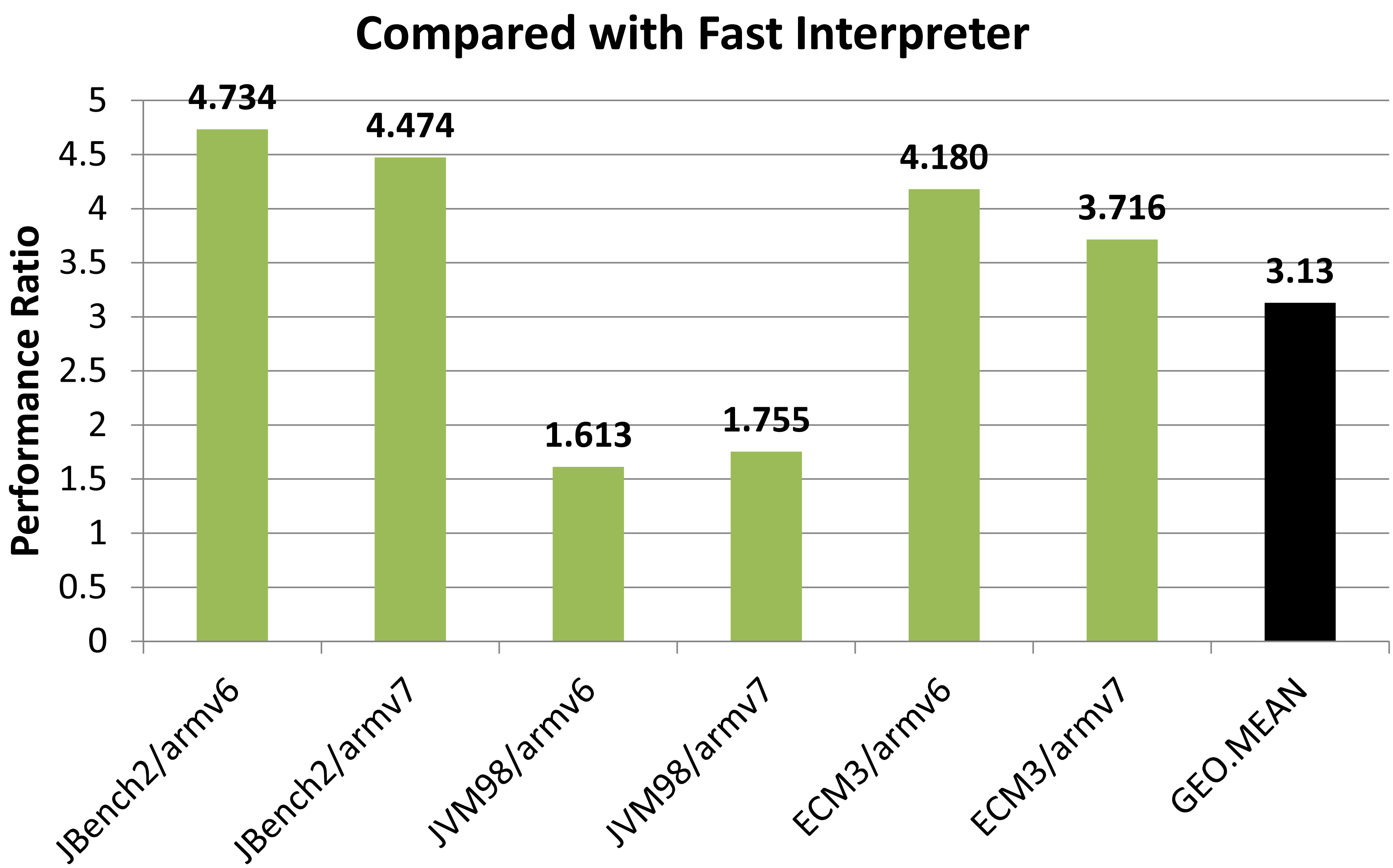
Benchmarks

- SPECjvm98, JemBench2, EmbeddedCaffeineMark3

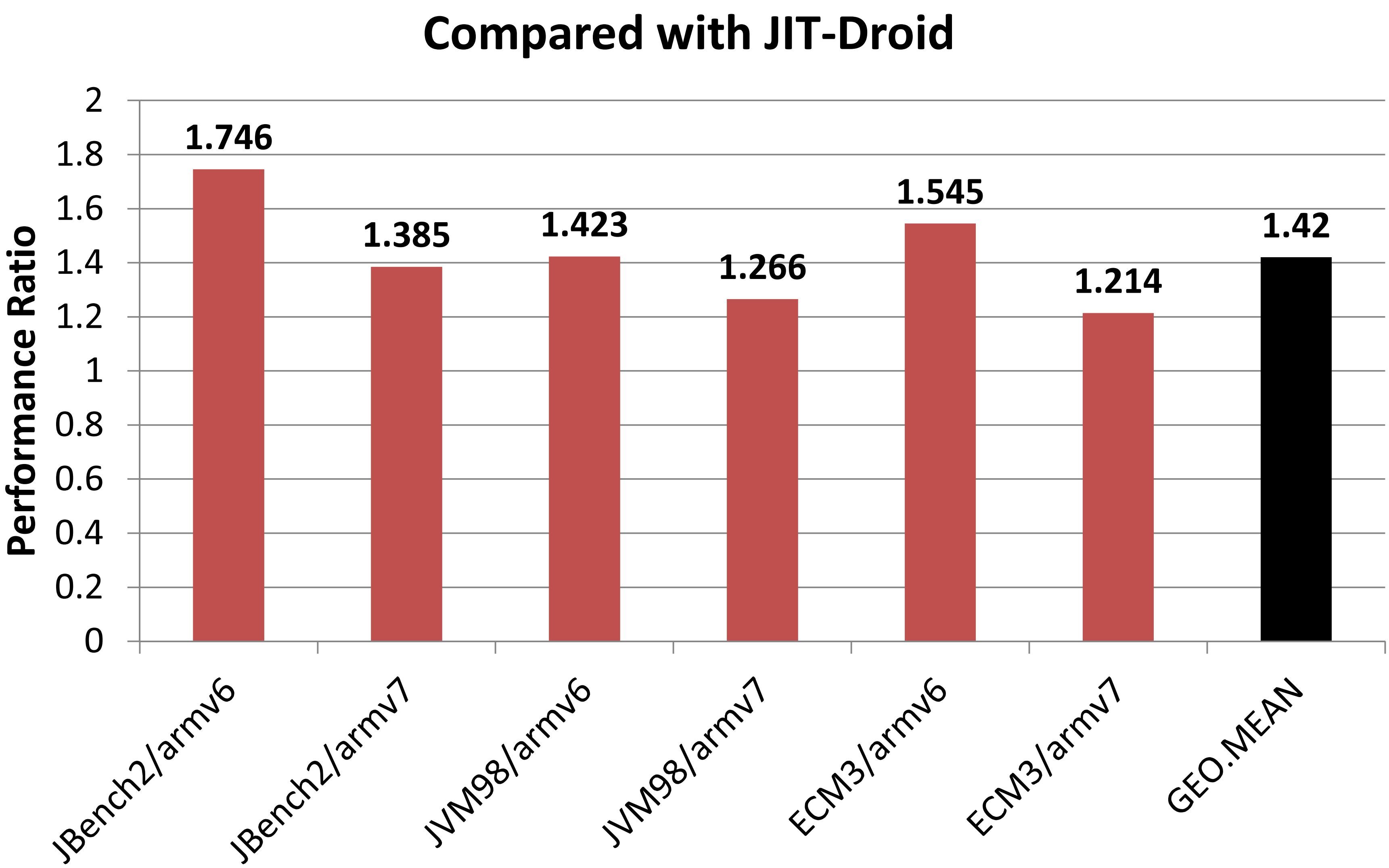
Software Platform

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

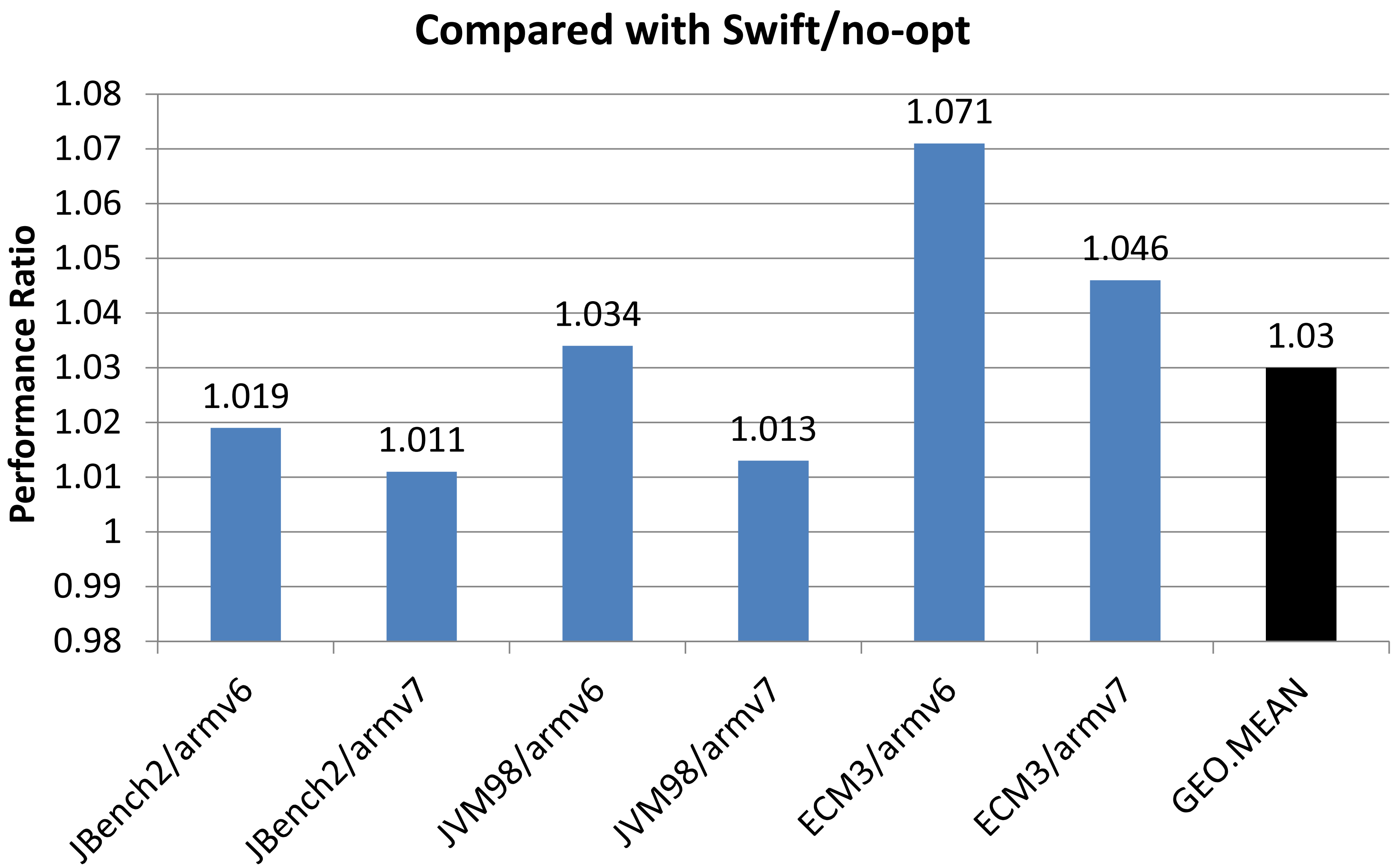
Performance-with Fast Interpreter



Performance-with JIT-Droid



Performance-with Swift/no-opt



Translation Time

Table 1: Translation Time of Swift on OMAP3530

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

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

Benchmark		*Swift(s)*	*JIT-Droid(s)*	*Percent*
SPECjvm98	compress	0.117	0.257	45.5%
	jess	0.185	0.850	21.8%
	db	0.124	0.270	45.9%
	javac	0.274	2.638	10.4%
	mtrt	0.178	0.948	18.8%
	jack	0.175	1.154	15.2%
ECM3		0.098	0.433	22.6%
JemBench2		0.092	2.184	4.2%

Code Size

Table 3: Translated Code Size of Swift on OMAP3530

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

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