Swift: A Register-based JIT Compiler for Embedded JVMs

Yuan Zhang Min Yang Bo Zhou Zhemin Yang Weihua Zhang Binyu Zang
Parallel Processing Institute, Fudan University
{yuanxzhang, m_yang, bo-zhou, yangzhemin, zhangweihua, byzang}@fudan.edu.cn

Abstract

Code quality and compilation speed are two challenges to JIT compilers, while selective compilation is commonly used to trade-off these two issues. Meanwhile, with more and more Java applications running in mobile devices, selective compilation meets many problems. Since these applications always have flat execution profile and short live time, a lightweight JIT technique without losing code quality is extremely needed. However, the overhead of compiling stack-based Java bytecode to heterogeneous register-based machine code is significant in embedded devices. This paper presents a fast and effective JIT technique for mobile devices, building on a register-based Java bytecode format which is more similar to the underlying machine architecture.

Through a comprehensive study on the characteristics of Java applications, we observe that virtual registers used by more than 90% Java methods can be directly fulfilled by 11 physical registers. Based on this observation, this paper proposes *Swift*, a novel JIT compiler on register-based bytecode, which generates native code for RISC machines. After mapping virtual registers to physical registers, the code is generated efficiently by looking up a translation table. And the code quality is guaranteed by the static compiler which is used to generate register-based bytecode. Besides, we design two lightweight optimizations and an efficient code unloader to make *Swift* more suitable for embedded environment. As the prevalence of Android, a prototype of *Swift* is implemented upon DEX bytecode which is the official distribution format of Android applications.

Swift is evaluated with three benchmarks (SPECjvm98, EmbeddedCaffeineMark3 and JemBench2) on two different ARM SOCs: S3C6410 (armv6) and OMAP3530 (armv7). The results show that *Swift* achieves a speedup of 3.13 over the best-performing interpreter on the selected benchmarks. Compared with the state-of-the-art JIT compiler in Android, JITC-Droid, *Swift* achieves a speedup of 1.42.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages; D.3.4 [*Programming Languages*]: Processors—compilers, run-time environments, code generation

General Terms Performance, Language, Design

Keywords Register-based Bytecode, Just-In-Time Compiler, Embedded JVM, Android

VEE'12, March 3–4, 2012, London, England, UK.

1. Introduction

The widespread of Internet has given people access to information in an easier way, while the technology of mobile computing helps to achieve this on a level never experienced before. Mobile devices as simple as mobile phones and as complex as mobile internet devices with various network connections and all kinds of sensors (such as cameras, GPS, near-field communication and accelerometers, etc) are the current computing wave, competing heavily with desktops and laptops for market and popularity. As a widely anticipated open source operating system (OS) for mobile devices, Android runs on diverse devices, such as phones, tablet PCs, netbooks, etc. Taken Java's portability and its large community into account, Android OS and its applications are implemented in Java language. Programmers can easily develop powerful and rich applications for these kinds of devices without concerning their architectures and configurations.

To be machine-independent, Java is compiled into a portable instruction format called bytecode [18]. When Java is proposed by Sun, it mainly targets client applications running in the browser, so it uses a stack-based bytecode which is more compact, minimizing the consumption of network bandwidth. However, the mismatch between the stack-based architecture of Java virtual machine (JVM) and underlying register-based architecture commonly found in mobile devices introduces much overhead during just-in-time (JIT) compilation.

To reduce the compilation overhead of JIT, traditional JIT systems usually equip two compilers at least: a non-optimizing compiler which compiles all the methods without any optimizations or only few lightweight optimizations; an optimizing compiler which recompiles hot methods and performs heavy optimizations to boost performance. Due to the compilation overhead, the non-optimizing compiler usually uses a simple local register allocator or just mimics the behavior of virtual stack, sacrificing code quality. Meanwhile, the number of methods compiled by the optimizing compiler must be limited for its significant compilation overhead. When computing begins moving to embedded environment which is constrained by CPU cycles and power supply, a lightweight JIT technique without losing code quality is extremely needed.

In Android, a new register-based Java bytecode format, DEX [1], is firstly used to narrow the gap between bytecode format and machine instruction format. All instructions in DEX bytecode format are designed to directly operate on virtual registers. Although register-based bytecode becomes popular since the appearance of Android, it has been discussed many years ago. In [13], Davis et al. first propose a virtual register machine architecture for JVM and compare the standard Java stack-based bytecode with their proposed register-based bytecode. They find that the register-based format reduces the total instruction count by 34.88%, while increasing the bytecode size by 44.81%. Shi et al. [30] extend the prior work by implementing a register-based JVM and compare the performance of these two kinds of bytecode. They find that the

register-based format reduces the execution time by 26.5% on a switch-based C interpreter [14, 15, 27]. Although JIT compilation techniques on stack-based bytecode is well studied, the research about constructing a register-based JIT is still blank. Actually, being much closer to Instruction Set Architecture (ISA) of physical machines, register-based bytecode creates an opportunity to improve the construction of JIT compiler on RISC processors. This paper focus on using register-based bytecode to design a lightweight JIT compiler without losing code quality.

Currently, Google has developed a JIT compiler (JITC-Droid) [11] in Android. JITC-Droid is very similar to the dynamic native optimization system proposed by Sullivan et al. [32] which works in two phases: *trace selection* and *trace compilation*. In JITC-Droid, a Java method is first executed by a very efficient interpreter which also detects the start points of hot traces by recording the execution counter at branch targets. When the execution counter exceeds a threshold, JITC-Droid enters the *trace selection* phase. The trace is selected by a special interpreter and composed of "non-control-flow" instructions. The *trace compilation* phase includes the following steps: CFG construction, SSA conversion, linear-scan register allocation, low IR construction and code generation. This process is very common in the traditional JIT compilers [12, 26, 31] which are proposed upon stack-based bytecode. After the trace is compiled, JITC-Droid will mark the start point of the trace so that the interpreter could immediately invoke the compiled code when the control flow transfers to this trace. Although JITC-Droid improves the performance a lot for some computational benchmarks against the best-performing interpreter in Android, it doesn't utilize the opportunity created by register-based bytecode. By exploiting the similarity between register-based bytecode and underlying register-based architectures commonly found in embedded processors, further performance gains can be expected.

Through a study on the characteristics of Java applications, we observe that virtual registers used by more than 90% Java methods can be directly fulfilled by physical registers. Based on this observation, this paper proposes *Swift*, a novel JIT compiler on register-based bytecode, which generates native code for RISC machines. After mapping virtual registers to physical registers, the code is generated efficiently by looking up a translation table. And the code quality is guaranteed by the static compiler which is used to generate register-based bytecode. Besides, we design two lightweight optimizations and an efficient code unloader to make *Swift* more suitable for embedded environment. As the prevalence of Android, a prototype of *Swift* is implemented upon DEX bytecode which is the official distribution format of Android applications.

Our proposed solution is better than traditional Just-In-Time compilers in the following two aspects. First, *Swift* can generate effective code online without extra effort, because register allocation and some optimizations are performed offline when compiling Java code to register-based bytecode. Second, our solution can compile all the methods executed without any prediction, because the translation is very fast. To the best of our knowledge, *Swift* is the first JIT compiler utilizing the opportunity created by register-based bytecode.

As the prevalence of Android, a prototype of *Swift* is implemented upon the generally accepted DEX bytecode. The major contributions of this paper are the following:

1. Through a study on the characteristics of Java methods, we show that register-based bytecode can be exploited to improve the construction of JIT compiler on RISC processors.
2. Based on the observation in the study, we propose a fast and effective Just-In-Time compiler called *Swift* specifically for register-based bytecode.
3. We implement a workable prototype of *Swift* for DEX bytecode in Dalvik Virtual Machine [10].
4. We evaluate *Swift* with three popular benchmarks on two platforms, including comparison with other two execution engines in Dalvik Virtual Machine.

The rest of the paper is organized as follows. Section 2 introduces register-based bytecode and illustrates our motivation. Section 3 describes our JIT system and its core components. Section 4 details the prototype of *Swift* for DEX bytecode, including two lightweight optimizations. In section 5, we evaluate *Swift* with three popular benchmarks on two ARM SOCs: S3C6410 (armv6) and OMAP3530 (armv7). The related work is discussed in Section 6. Section 7 concludes and makes some discussions.

2. Register-based Bytecode

In this section, we take DEX bytecode as an example to introduce register-based bytecode and compare it with traditional stack-based bytecode. Then through a study on Java method characteristics, we show the opportunity for JIT construction on register-based bytecode.

2.1 Bytecode Comparison

Designed to be machine independent, bytecode acts as an intermediate representation for managed languages. The original Java bytecode specified by Sun is stack-based. In this stack-based bytecode format, instructions operate on two sources of operands: *local area* and *virtual stack*. *Local area* is used to hold local variables of a Java method, while *virtual stack* is the place where computing occurs. Excluding the instructions interacting with VM, there are two kinds of instructions in this bytecode format: 1) data movement instruction which transfers data between *local area* and *virtual stack*; 2) computation instruction which fetches operands from the *virtual stack* and stores the result back to *virtual stack* according to the type of the instruction.

In Android, another kind of bytecode format called DEX bytecode is used to distribute Java applications. To run DEX bytecode, Google develops a Java virtual machine called Dalvik [10]. Different from the stack-based bytecode, DEX is based on virtual registers. The instructions in DEX can be grouped into two categories: 1)VM-related instructions which interact with virtual machine; 2)computation instructions which just compute on the virtual registers. There are about 232 instructions in DEX, and 43 of them are VM-related instructions. The remaining instructions are very simple instructions such as arithmetic instructions, logic instructions, etc.

Figure 1 shows an example of *sum* function and its corresponding bytecode in stack-based format and DEX format. From Figure 1, we can find that in stack-based bytecode, data are frequently transferred between *local area* and *virtual stack*, which is quite inefficient. By contrast, every instruction in DEX directly operates on virtual registers, eliminating unnecessary data movements.

As Android is widely accepted by hardware manufacturers, software companies and wireless operators, DEX is becoming more and more popular, especially in embedded devices. To be compatible with the stack-based bytecode which is the standard format for Java, Google develops a translation tool called ***dx***. It translates several *.class* files into one single *.dex* file. The translation process is showed in Figure 2. When allocating the virtual registers, ***dx*** use a graph-coloring algorithm with unlimited registers and the virtual registers are numbered from zero. To reduce the size of the *.dex* file, ***dx*** also compresses the *.dex* file by sharing a constant pool among all classes it handled.

In DEX, each method has its own virtual register set. The *sum* function showed in Figure 1 has 5 virtual registers and 3 of them

```
int sum(int start, int end){
    int sum = 0;
    for(; start < end; start ++)
        sum += start;
    return sum;
}
```

(a) *sum* function

```
000: iconst_0           //push const 0
001: istore_3           //pop 0 to v3(sum)
002: iload_1            //push v1(start)
003: iload_2            //push v2(end)
004: if_icmpgt 017      //compare start and end
007: iload_3            //push v3(sum)
008: iload_1            //push v1(start)
009: iadd               //push (sum + start)
010: istore_3           //sum = sum + start
011: iinc    1, #1      //add v1(start) by 1
014: goto    002
017: iload_3            //push v3(sum)
018: ireturn            //return sum
```

(b) Stack-based Bytecode

```
000: const/4 v0, #0           //v0(sum) = 0
001: move v1, v3              //v1 = v3(start)
002: if-ge v1, v4, 008        //test v1, v4(end)
004: add-int/2addr v0, v1     //v0 = v0 + v1
005: add-int/lit8 v1, v1, #1  //v1 = v1 + 1
007: goto 002
008: return v0                //return sum
```

(c) DEX Instructions

Figure 1. Comparison of Stack-based bytecode and DEX bytecode in *sum* function.

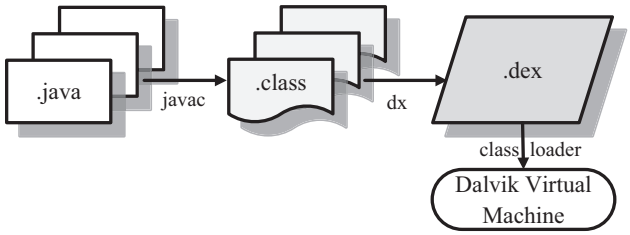

Figure 2. Transform several *.class* files to single *.dex* file.

are marked as *input registers*, ranging from v2 to v4. The remaining 2 registers are marked as *local register* used to store local variables ranging from v0 to v1. Through Figure 1, we find that the DEX instructions are very similar to machine instructions. We will show that this similarity can be exploited to construct a lightweight JIT compiler in the following.

2.2 Java Method Characteristics

The principle of separation logic is widely accepted today, especially in object-oriented programming languages, such as Java. In Java, programs are often divided into many packages, and each package is constructed by structured classes with inheritance. Each class is further divided into small methods to handles one specific logic. This loosely-coupled design makes Java program easy to extend. As a result, we can find that most Java methods are relatively simple. So we assume that a small amount of variables are enough for most Java methods to express their logic. To confirm this assumption, we design an experiment on Dalvik Virtual Machine.

Experimental Environment. The experiment is performed on OMAP3530 manufactured by Texas Instrument which has a 600MHz Cortex-A8 (armv7) CPU core with 16KB I-Cache, 16 KB D-Cache and 256KB L2 cache. We instrument the interpreter in Dalvik virtual machine to record all the Java methods executed. Dalvik runs in interpreter mode on 0xlab[1] Android 2.1 OS which is a customization for OMAP3530. To increase the reality of our experiment, we use six cases (*compress*, *jess*, *db*, *javac*, *mtrt*, *jack*)[2] in SPECjvm98 [4] with a problem size of 100 and six Android core applications (*system_server*, *app_process*, *input_method*, *calendar*, *setting*, *email*) to collect data. In register-based bytecode, virtual registers of a Java method are acted as variables of this method.

Observation. The *y-axis* in Figure 3 is the percentage of invoked methods which use no more virtual registers than the number specified by *x-axis*. We observe that more than 90% methods called in the experiment use no more than 11 virtual registers.

Motivation. Most embedded devices equip a RISC processor for its power efficiency, simple design and low price. And RISC processors often have many programmer-visible registers, for example there are 16 programmer-visible registers in ARM architecture and 32 programmer-visible registers in MIPS architecture. Excluding some special usage registers, there are still many registers available. Based on the observation above, we can conclude that this small amount of virtual register requirement can be easily satisfied with physical registers without performing any complex register allocation algorithm.

Besides, the homogeneity between register-based bytecode and RISC machine code eases the work of code selection. For example, we can find corresponding RISC instructions for 189 out of 232 DEX instructions. With the adequate amount of physical registers in RISC processors, we can easily allocate physical registers for all the virtual registers in most executed methods and machine code can be generated straightforwardly by replacing register-based bytecode with corresponding RISC instructions. The quality of the generated code is guaranteed, because the static compiler have performed a near-optimal register allocation and heavy optimizations when compiling Java code to register-based bytecode.

3. Register Mapping Translation

In this section, we presents our JIT compilation system for register-based bytecode. We begin this section with a high-level design issues that drive our work and then describe the architecture of *Swift*. The final part of this section details *Swift*'s two core components: *Register Mapping Table* and *Template-based Code Selector.*

3.1 Design Issues

To construct a lightweight JIT compiler with good quality for embedded JVMs, the design of *Swift* takes the following aspects into consideration:

- **Minimal Runtime Overhead.** JIT compilers always add additional burden to the applications. Unlike desktop application and server application, applications on embedded devices such as Android have a short lifecycle. As a embedded JIT compiler, *Swift* should feature lightweight compilation techniques, such as efficient register allocator and code selector to control its

[1] http://0xlab.org/

[2] *mpegaudio* is not included because ***dx*** fails to compile it's obfuscated bytecode to DEX.

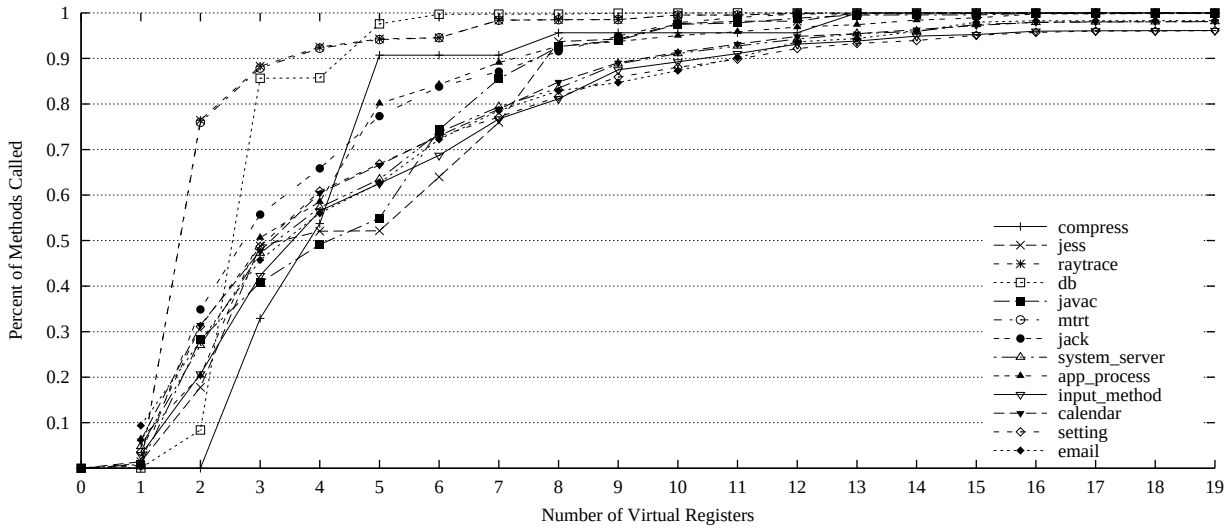

Figure 3. The percentage of invoked methods with varied number of virtual registers.

compilation overhead. Otherwise, the time used to compile bytecode can't be amortized.

- **Simple Design.** There are many architectures widely used in embedded devices, such as ARM, MIPS, etc. The design of a JIT compiler for embedded JVMs should be simple enough to minimize the efforts of porting it to other embedded architectures.
- **Moderate Code Size.** Current embedded systems such as Android encourage cooperation among multiple applications. Constrained by memory, the amount of applications running at the same time is also limited. While *Swift* has to use additional memory to store generated code, memory impact should also be taken into account.
- **Application portability.** Application portability is the most important issue in the design of managed execution environment. The design of *Swift* should not rely on any assumption about the application distribution format. So we don't consider annotation-assisted solutions [23, 24] or hybrid solutions which combines JIT compilation with Ahead-of-Time compilation [21, 33].

3.2 Swift Architecture

Like a non-optimizing compiler, *Swift* translates all methods executed except the *static class initializers* which are executed only once. To reduce the overhead of interpreting a method, *Swift* translates the method before it is executed. In *Swift* there is no specialized translator thread(s) and every method is translated by the thread which first executes it.

To prevent multiple threads from translating the same method which is very inefficient, a thread must acquire the lock of the method before translating it. The lock is represented by an integer and the acquire operation is implemented with atomic instructions. All the other threads must yield CPU if they attempt to translate a method whose lock is acquired by another thread.

Figure 4 depicts the architecture of *Swift*. As it shows, the translation process can be separated into two phases. The first phase is the local translation process, in which each thread translates the invoked method and stores the generated code in a thread-local code cache. In the second phase, the code in local cache is committed to a global shared code pool and the commit operation is protected by a global lock to avoid write contention.

Register allocation and code selection are two main parts in state-of-the-art JIT compiler [12, 26, 31]. Actually, in *Swift* the register allocator is quite straightforward, because physical registers are fixedly assigned to virtual registers by using a register-mapping algorithm. For code selection, *Swift* adopts a template-based translation technique which is lightweight enough for a Just-In-Time compiler. The code is generated within a single pass over the method's bytecode. Without performing any data-flow or control-flow analysis at runtime, the translation process is very fast.

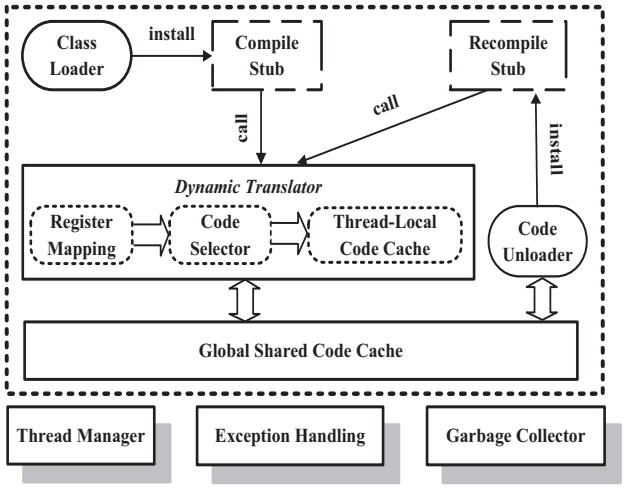

Figure 4. The architecture of *Swift*.

Like other JIT system, *Swift* must cache all the generated code for future use. As a result, the memory consumption of code cache is a critical problem on resource-constrained embedded devices. To reduce the code area, *Swift* features a code unloader which unloads generated code of unnecessary methods. A recompile stub is installed on the unloaded method to trigger recompilation before it is executed later. Besides, to support the execution of Java

programs, *Swift* also need to cooperate with other VM modules, such as garbage collector (GC), etc.

3.3 Register Mapping Translator

As we find through the study on the characteristics of Java methods, more than 90% Java methods use no more than 11 virtual registers and these amount of registers can be easily satisfied by underlying RISC processors. Because ***dx*** performs a graph-coloring based algorithm to allocate virtual registers when translating *.class* files into a *.dex* file, it is unnecessary to perform additional complex allocation algorithm on DEX in *Swift*. Thus, a register mapping table is used per method, to record the static relations between virtual registers and physical registers.

Although the amount of free physical registers in the underlying processor is enough for most methods, there exists some large methods which use more virtual registers due to complex logic, bad programming habit, etc. *Swift* need to do special handling when translating these methods. To differentiate these two kinds of methods, we designate those methods whose virtual registers can be completely mapping to physical registers as *regular methods* and the remaining methods are designated as *irregular methods*.

Register Mapping Table. In DEX, virtual registers in a method are continuously numbered from zero, and the amount can be acquired in the method meta info from *.dex* file, so a register table can be constructed according to the register mapping algorithm before traversing DEX instructions. The register mapping algorithm should satisfy following rules and the detailed algorithm used in our prototype can be found in section 4.1:

1) In a *irregular method*, not all the virtual registers can be mapped to physical registers. Those virtual registers that hasn't been mapped are allocated in a continuous area in the corresponding stack frame of the function. This part of stack frame is called *spill area*.
2) In the scope of a method, the mapping from a virtual register to a physical register or *spill area location* is fixed and only determined by the method itself.
3) In every method, any physical register or *spill area location* should only be assigned to at most one virtual register. And every virtual register should have one corresponding physical register or *spill area location* even if it is not used by any DEX instruction[3].
4) In the translation of some DEX instructions such as VM-related instructions, temporary registers are needed due to their complex logic. In this situation, some physical registers (even the ones that have been assigned to some virtual registers) are selected as temporary registers. They are saved to the stack before the translation starts and restored after this DEX instruction is translated.

Template-based Code Selector. After mapping virtual registers to physical registers, machine code is generated by looking up a predefined translation template table indexed by DEX opcode. With this template table, the translator linearly traverses all the DEX instructions of the method and fills the local code cache. The translation can be finished in a single pass without complex logic, hence the translation is very fast.

The template table for *irregular methods* is different to the table for *regular methods*, because in *irregular methods* those virtual registers mapped to *spill area* need special handling when they are used in DEX instructions. To handle such a virtual register, we need to check whether it is a source register or a destination register. If it is a source register, a temporary register is allocated to store its value and used in the generated code of this DEX instruction. If it is a destination register, we just need to store the result to the corresponding *spill area*. Because the mapping from virtual registers to physical registers or *spill area* is fixed, the translation process is still very efficient.

[3] The *sum* function described in section 2.1 has 5 virtual registers, but only 4 registers is used in its DEX instructions. This is because the *sum* function is an instance method, v2 register is used to store *this* reference although it is not used in any DEX instruction.

4. Prototype of Swift

Based on the register mapping translator described above, we implement a prototype of *Swift* upon the generally accepted DEX bytecode. As ARM is the most popular architecture used in Android-powered devices, the implementation is built on ARM. However, our JIT technique is not restricted to ARM, it can be used in any other RISC processors, such as MIPS, SPARC. And more performance gains can be expected on these processors, because they have more registers than ARM. In this section, we detail this prototype, including an efficient code unloader and two lightweight optimizations.

4.1 Translator Implementation

On ARM platform, there are many kinds of instruction set supported in different ARM variants. *ARM instruction set* is a 32bits ISA that supported in all ARM variants. *Thumb instruction set* is a 16bits ISA which is more compact. Limited by the instruction width, Thumb instructions can't access all the registers and only part of them can support conditional execution. *Thumb2 instruction set* is an enhanced version of Thumb ISA and supported by armv6t2 and armv7. It is a mixed 16bits and 32bits ISA, so it is more powerful than Thumb ISA and more compact than ARM ISA.

Although Thumb2 ISA has obvious advantages, many chips applied in the market don't implement this ISA. To keep compatibility with all ARM variants while not harming performance, *Swift* chooses to generate ARM instructions. There are 16 programmer-visible registers designed in all ARM variants. Excluding three special purpose registers (*stack pointer register*, *link register* and *program counter register*), there are still 13 free registers(r0-r12) left for *Swift* to map virtual registers.

Regular Method. For every *regular method*, physical registers are assigned to virtual registers from r0 to r12. As other special usage registers, *link register* is not assigned to any virtual register. However, since *link register* is only used in *branch-with-link* instruction and saved in the method entry, we reuse it as temporary register.

```
DEX Instruction:
    000: const/4 v0, #0
    001: move v1, v3
    002: if-ge v1, v4, 008
    004: add-int/2addr v0, v1
    005: add-int/lit8 v1, v1, #1
    007: goto 002
    008: return v0
Generated ARM Code:
    0000: mov   r3, #0
    0004: mov   r4, r1
    0008: cmp   r4, r2
    000b: bge   001b
    0010: add   r3, r3, r4
    0014: add   r4, r4, #1
    0018: b 0008
    001b: //omit the complicated return logic here
```

Figure 5. Generated code of *sum* function.

Figure 5 shows the generated code slice of the *sum* function described in section 2.1. The logic of method entry and exit is

omitted from the figure, because they are not related to the translation algorithm. They are explained separately in the following. From Figure 5, we can conclude that the generated code is of good quality.

Irregular Method. For *regular methods*, *Swift* only allocates temporary registers for VM-related instructions. However, *irregular methods* even need temporary registers to translate computational instructions for handling the accesses to virtual registers that are mapped to *spill area*. *Link register* is still used as a temporary register for *irregular method*. And we find four temporary registers are enough for the translation of all the computational instructions in *irregular methods*, so three additional registers(r10-r12) are used as temporary registers. As a result, for every *irregular method*, physical registers are assigned to virtual registers from r0 to r9.

```
DEX Instruction:
    add-int/lit8 v15, v15, #int 1
Generated ARM Code:
    ldr r10,   [sp, #12]
    add r10,   r10, #1
    str r10,   [sp, #12]
```

Figure 6. Special handling of *spill area*.

Figure 6 shows the handling for accesses to those virtual registers allocated in *spill area*. The *add-int/lit8* DEX instruction also appears in the *sum* function described in section 2.1. The virtual register v15 is mapped into *spill area* with an offset of 12 relative to *stack pointer*. Before the add operation, a load instruction must be generated to load v15 to a temporary register and in this case, r10 is selected as the temporary register. After the add operation finishes, we need to store r10 to the mapped *spill area location* of v15, because its value has been changed since last load.

Register Mapping Algorithm. In DEX, virtual registers can be separated into two categories: *input virtual registers* which are used to store arguments of the method and *local virtual registers* which are used to store local variables. For any method, we first map *input virtual registers* to physical registers starting from r0, and then the *local virtual registers*. For example, in the *sum* function described in section 2.1, its *input virtual registers* v2-v4 are mapped to physical registers r0-r2 linearly, and *local virtual registers* v0-v1 are mapped to physical registers r3-r4 linearly. Those virtual registers that can not be mapped to physical registers are mapped to fixed *spill area locations*. This mapping algorithm is easy for caller to pass arguments because caller can determine the physical registers assigned to *input virtual registers* in callee before invocation.

Register allocator is critical to construct a fast and effective JIT compiler. In [22] Krall implements a simple register allocator for CACAO JavaVM. Through stack analysis, a register is allocated when a stack variable is pushed, and the register is recycled when the stack variable is poped. Local variables are always cached in registers. When the register is not enough, spill area is used. This strategy is simple but not effective, because it doesn't eliminate redundant data movements between stack variables and local variables. In LaTTe compiler[34], stack-based bytecode is first translated into pseudo RISC code with symbolic registers, and then register allocated to generate SPARC code. The register allocator use a linear-scan allocation algorithm with the local variable lookahead strategy. The authors think this is a good trade-off between quality and speed. Actually, the trade-off made by *Swift* is better than this one, because the register mapping table in *Swift* can be constructed with negligible overhead, and the quality is guaranteed by the static compiler which is used to compile Java source code to register-based bytecode.

4.2 Calling Convention

To support method invocation, *Swift* designs a convention between caller and callee to clarify their responsibilities.

1) **Caller, Before Call.** In Java, the size of return type is 8 bytes at most, so r0-r1 are used to pass return value in *Swift*. Before the invocation, caller must save *caller saved registers* and pass arguments to callee. The *caller saved registers* include r0-r1 as well as the physical argument registers in callee. The amount of *input virtual registers* in callee can be acquired from the *invoke* DEX instruction, so the physical argument registers in callee can be determined by caller according to the register mapping algorithm described previously.
2) **Callee, Method Entry.** In the method entry, callee must save *stack pointer register* and *link register* first and save *callee saved registers*. The *callee saved registers* include the physical registers assigned to *local virtual registers* in callee, excluding r0 and r1. Callee doesn't save r0-r1 because caller has saved them already.
3) **Callee, Method Exit.** Before callee returns, callee must restore *callee saved registers*, save return value to r0-r1, and restore *stack pointer register* and *link register*. Additionally, to cooperate with GC, callee also checks whether there is a GC suspend request.
4) **Caller, After Call.** After callee returns, caller must move r0-r1 to the correct result registers according to the DEX instruction and restores *caller saved registers*.

In this calling convention, responsibilities are clearly separated between caller and callee into 4 phases. There is no live registers across a method call, and we store floating point value in integer registers because DEX doesn't distinguish register types used to store floating point value and integer value..

During the method invocation, caller also need to resolve callee first according to different invoke types. In DEX, the callee of *virtual-call* or *interface-call* can only be resolved at invocation point due to the dynamic binding feature in Java. For these two type of calls, a runtime routine is called to resolve callee every time before invocation.

4.3 Code Unloading

Swift translates every method before it is executed, because the translation is very efficient. Although this translation strategy has little impact on the execution time, it increases the memory footprint, since the size of native code is much larger than that of its bytecode equivalent [36]. For embedded devices, this cost is significant, especially when memory is severely constrained.

Meanwhile, not all the translated methods are always needed after translation. As Zhang et al. [36] shows, for most Java workloads over 60% of methods are effectively live for less than 5% of the total time they are in the system. To reduce the size of code memory, *Swift* adaptively unloads the translated code. This is a compromise between the interpreter which doesn't store any code and JIT compiler which caches all the translated code. If an unloaded method is later invoked, *Swift* will also recompile it before invocation. Since our translation is very efficient, we could trade a few CPU cycles for memory space.

In [36, 37], Zhang et al. compares different unloading strategies including *Online eXhaustive profiling(OnX)*, *Online Sample-based profiling(OnS)*, *Offline exhaustive profiling(Off)*, and *No Profiling(NP)*. The first three strategies try to improve the efficiency of code unloading by finding the precise unloading candidates, while the *NP* strategy is too aggressive to unload a lot of methods that would be invoked in the near future. However, the *OnX* strategy and

OnS strategy suffer from increasing runtime overhead too much, and the *Off* strategy doesn't scale to all applications.

In *Swift*, we don't use a complex but maybe precise unloading strategies for the following considerations: 1) our JIT compiler is designed to be lightweight, a complex code unloader would make it too complicated; 2) a precise unloading strategy always needs exhaustive profiling which burdens the application; 3) the overhead of recompiling a method in *Swift* is quite low, so a precise unloading strategy seems not necessary. In *Swift*, code unloading is performed at GC point. To reduce the overhead of code unloading, it is only performed when the total code size exceeds a user-defined threshold. When GC thread enumerates the stack of every thread, it marks all the methods on the stack by increasing the counter of method. Any method which is observed unmarked twice is unloaded immediately. When a method is unloaded, all the memory resource related to it is released, and a recompile stub is installed. The impact of code unloading is evaluated in the next section.

4.4 Lightweight Optimizations

To further improve the performance of *Swift*, we also design two optimizations which are lightweight enough for embedded devices.

Optimization for *Irregular Method*. As described in section 4.1, physical registers are first mapped to *input virtual registers* and then the *local virtual registers* in *irregular methods*. When the physical registers are not enough for allocation, *spill area* is used. Any access to *spill area location* needs special handling as Figure 6 depicted. This mapping algorithm may be quite inefficient when *spill area locations* are frequently accessed.

To reduce the effect of this problem, we design a lightweight optimization to improve the previously mentioned register mapping algorithm for *irregular methods*. Firstly, a loop detection algorithm is used to detect all the loops in the method. Secondly, we order all the basic blocks according to their occurrences in loops. Finally, we map those virtual registers which are used in basic blocks with high loop occurrence to physical registers, and the remaining virtual registers are all mapped to *spill area*.

Optimization for *interface-call*. In Java, a class is allowed to inherit at most one class, so a *virtual-call* can be resolved by directly accessing the corresponding entry in object's vtable. However, a class is allowed to implement multiple interfaces, so we need to perform a linear search in all the interfaces the class implemented to resolve an *interface-call*. As a result, the overhead of an *interface-call* is quite larger than a *virtual-call*.

We use a class-test mechanism to reduce the overhead of an *interface-call*. When an *interface-call* is resolved, we record the class type of the object and the resolved callee in a cache near the call site. When this call is invoked again on the object with the same class type as the recorded class, then the callee can be directly fetched from the record. If the class type is mismatched, the callee info is resolved through the slow path. This optimization relies on the speculation that the object type of an *interface-call site* is always the same in most cases.

5. Evaluation

Previous sections describe our proposed JIT compiler which generates code by mapping virtual registers to physical register. This section shows our evaluation on it.

5.1 Experimental Environment

The evaluation is performed on two ARM SOCs: the first one is OMAP3530 manufactured by Texas Instrument which has a 600MHz Cortex-A8 (armv7) CPU core with 16KB I-Cache, 16 KB D-Cache and 256KB L2 cache; the second one is S3C6410 manufactured by Samsung which has a 800MHz ARM1176JZF-S (armv6) CPU core with 16KB I-Cache and 16 KB D-Cache (no L2 cache). The reason for including the armv6 based SOC is that there are quantities of armv6 based SOCs already applied in the market, although SOCs based on armv7 are very popular. This testbed configuration represents the main stream of ARM architecture.

Our prototype of *Swift* is developed on 0xlab Android 2.1 OS which is customized for OMAP3530. To the best of our knowledge, JITC-Droid is the only JIT compiler working on DEX bytecode, so *Swift* is compared with the best-performing interpreter and JITC-Droid in Android. To test the performance of interpreter and JITC-Droid, we patch 0xlab Android 2.1 with the source code of Android 2.3.4 fetched from the Android Open Source Project. We use the same source code to build executables for S3C6410 just by changing the building configuration.

For evaluation, three standard benchmarks (SPECjvm98 [4], EmbeddedCaffeineMark3 [2] and JemBench2 [28]) are used. Although the real workload Android applications, such as Angry Birds are the most appropriate to evaluate *Swift*, these applications are hard to measure quantitatively. According to the study in section 2.2, we believe these benchmarks can represent the real workload on Dalvik because they share the same feature exploited by *Swift*.

JemBench2 consists of kernel cases (*sieve*, *bubblesort*), real-world applications (*kfl*, *lift*, *udpip*), and parallel cases (*matrix multiplication*, *Nqueens*, *raytrace*, *AES*). EmbeddedCaffeineMark3 is a subset of the complete CaffeineMark suite, including the following cases: *sieve*, *loop*, *logic*,*string*, *float*, and *method*. SPECjvm98 is not aimed to run on embedded systems, but it provides large workloads for performance evaluation. It is used with a problem size of 100 and six cases (*compress*, *jess*, *db*, *javac*, *mtrt*, *jack*)[4].

5.2 Performance Evaluation

To evaluate the performance of *Swift*, experiments are performed on two SOCs with three benchmarks, and *Swift* is compared with the best-performing interpreter, JITC-Droid and *Swift* itself with the two optimizations described in section 4.4 disabled. We disable code unloading in *Swift* to collect performance data. The impact of code unloading is evaluated separately in the following. All the results are formatted as a ratio to the best-performing interpreter.

Figure 7 shows the performance results. As we expect, *Swift* gains an significant speedup on both SOCs for all the cases in the three benchmarks. Compared with the best performing interpreter, *Swift* gains an average speedup of 1.61, 4.18 and 4.74 on SPECjvm98, EmebeddedCaffeineMark3, and JemBench2 respectively. *Swift* also gains an average speedup of 1.42 over JITC-Droid. The impact of our proposed two lightweight optimizations is obvious. They achieve a further speedup of 1.11 and 1.05 when compared with the best-performing interpreter and JITC-Droid, respectively.

Swift acts better with SPECjvm98 on OMAP3530 than on S3C6410, because all the cases in SPECjvm98 exhibit a large memory footprint (large than 1GB) and OMAP3530 has a 256KB L2 cache while S3C6410 has no L2 cache. Compared with JITC-Droid, *Swift* has a higher speedup on S3C6410 than that on OMAP3530. This is because on S3C6410 JITC-Droid generates Thumb instructions which are less effective than ARM instructions used by *Swift*. Although this makes a little unfair for the comparison, huge engineer efforts are needed to perform an absolutely fair experiment. Actually the experiment on OMAP3530 is fair because on OMAP3530 JITC-Droid generates Thumb2 instructions which has the same ability as ARM instructions used by *Swift* and we can still find a significant speedup of Swift against JITC-Droid on OMAP3530 from figure 7.

[4] *mpegaudio* is not included because ***dx*** fails to compile it's obfuscated bytecode to DEX.

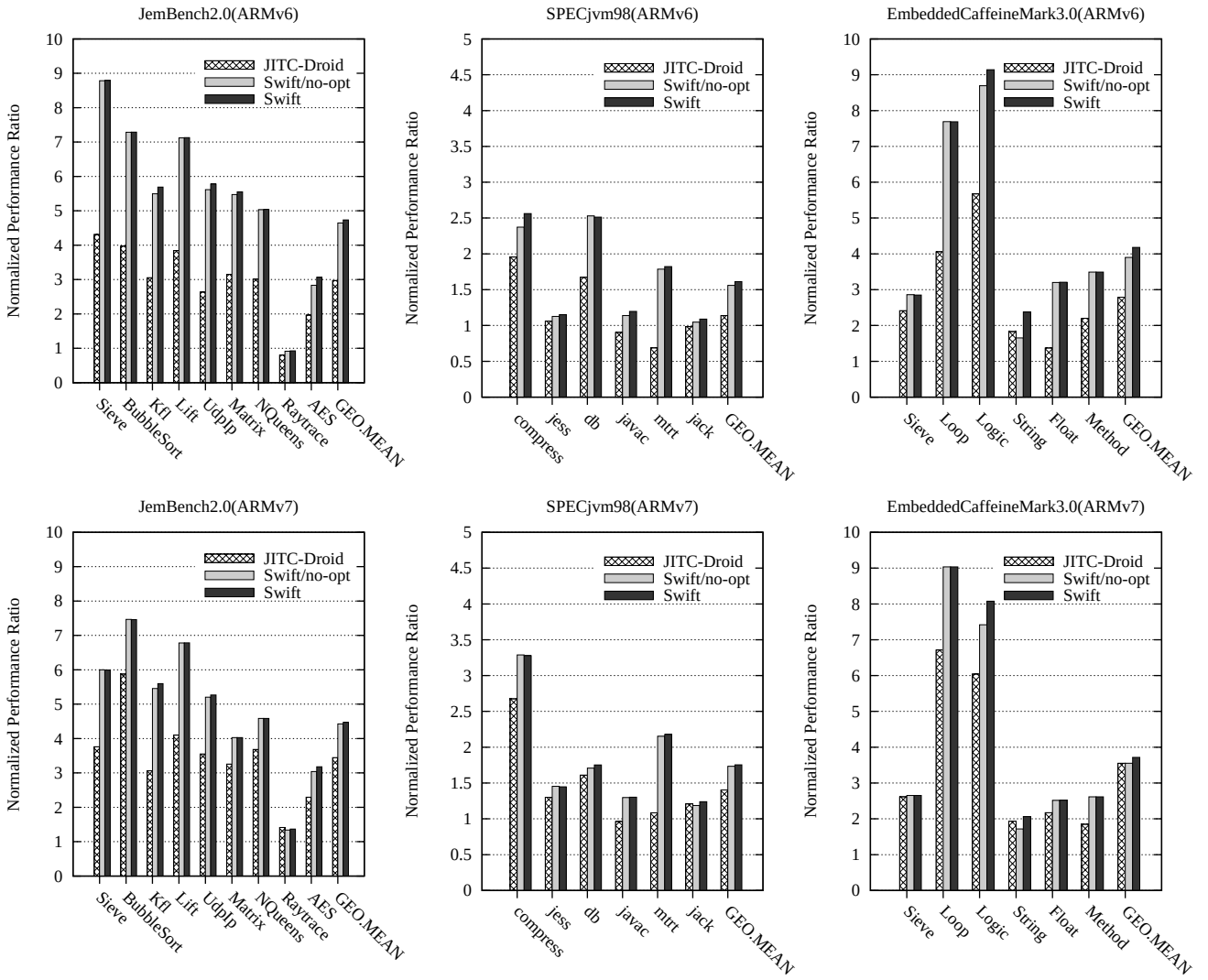

Figure 7. Normalized performance ratio of JITC-Droid and *Swift* to the best-performing interpreter in Dalvik. The results is showed in 6 diagrams of two SOCs and three benchmarks. A geometric mean ratio is presented last for each configuration.

Case	*Trans. Time(s)*	*Exec. Time(s)*	*Percent*
compress	0.117	91.170	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%
JBench2	0.092	27.4	0.334%

Table 1. Translation Time of *Swift* for SPECjvm98, JemBench2, and EmebeddedCaffeineMark3 on OMAP3530.

Case	*Trans. Time(s)*	*Exec. Time(s)*	*Percent*
compress	0.257	109.167	0.236%
jess	0.850	85.913	0.990%
db	0.270	69.848	0.387%
javac	2.638	132.372	1.993%
mtrt	0.948	151.828	0.624%
jack	1.154	88.73	1.301%
ECM3	0.433	24.168	1.793%
JBench2	2.184	28.874	7.565%

Table 2. Translation Time of *JITC-Droid* for SPECjvm98, JemBench2, and EmebeddedCaffeineMark3 on OMAP3530.

Translation Time. Table 1 and table 2 shows the translation time of *Swift* and JITC-Droid on OMAP3530. The situation on S3C6410 is similar. Because the cases in JemBench2 and EmebeddedCaffeineMark3 can't be measured individually, the whole benchmark's translation time is presented. *Swift* is configured to perform both optimizations described in section 4.4. For all the cases, *Swift* costs no more than 0.3 secs to translate all the executed Java methods. Translation time used by *Swift* occupies no more than 0.5% of the total execution time for every case. Compared to JITC-Droid, the translation time of Swift is less than 20% of the translation time of JITC-Droid in average. Considering the translation time of Swift and JITC-Droid both occupy a quite small

part of the total execution time, the bigger performance gain of *Swift* is attributed to the better code quality. By exploiting the the similarity between register-based bytecode and RISC ISA, *Swift* is lightweight enough for resource-constrained devices.

5.3 Impact of Code Unloading

Table 3 shows the translated code size of *Swift* on OMAP3530 when code unloader is enabled and disabled. The translated code size on S3C6410 is similar. The evaluations in this section are all performed on OMAP3530 if not specially mentioned. The cases in JemBench2 and EmebeddedCaffeineMark3 can't be measured individually, so the whole benchmark's translated code size is presented. The two optimizations described in section 4.4 are not compatible with code unloader, so they are disabled in this experiment.

Case	*Unload On(KB)*	*Unload Off(KB)*	*Percent*
compress	122.442	313.229	39.1%
jess	154.969	549.314	28.2%
db	104.468	336.174	31.1%
javac	484.338	875.173	55.3%
mtrt	142.13	443.936	32.0%
jack	212.583	577.368	36.8%
ECM3	150.483	251.656	59.8%
JBench2	193.34	233.205	82.9%

Table 3. Translated code size of *Swift* for SPECjvm98, JemBench2, and EmebeddedCaffeineMark3 on OMAP3530. The code unloader is configured to unload code when the code size is larger than 200KB.

As the translated code size varies during the execution of the program, the average code size is used to represent the code size in the stable state. It is calculated by the following formula. In this formula, $code\_size_i$ is the size of code cache at the time t_i. The code size information is collected at every GC point.

$$average\_code\_size = \sum_{t_i=t_1}^{t_n} code\_size_i * (t_i - t_{i-1})/t_n$$

We can find that all the cases in SPECjvm98 exhibit a large code memory footprint due to the complex logic of these workloads, and our code unloader saves more than 40% code memory for each case. EmbeddedCaffeineMark3 and JemBench2 don't consume a large code area as SPECjvm98 and our code unloader doesn't save much space for them. In these two benchmarks, GC is not triggered frequently, and our code unloader is performed at GC point, so many translated methods exist too long in the code cache before being unloaded. This problem can be easily solved by adding a unloader trigger before committing local compiled code into the global code cache. As the GC pressure in EmbeddedCaffeineMark3 and JemBench2 is not high, the delay of code unloading doesn't have a significant impact on memory.

Impact on Performance. From Figure 8, we can find code unloading has little impact on performance for most cases and some cases even gain a further speedup on unloading, such as *udpip* in JemBench2 and *string* in EmbeddedCaffeineMark3. We can find some cases in SPECjvm98 have obvious performance degradation when enabling code unloading. This is due to the frequently recompilation of unloaded methods. For example, our collected data shows that in *javac* the translation time grows from 0.27 secs to 5.71 secs, and in *jack* the translation time grows from 0.15 secs to 1.97 secs. This overhead can be easily eliminated by caching unloaded method into the file system and loading back before the method is invoked later. Overall speaking, the impact of code unloading on performance is minor.

6. Related Work

The related work to this paper are divided into three parts: JIT construction, bytecode format and mobile code system.

State-of-the-art JITs. JIT is widely used in managed runtimes. There has been many Java Just-in-Time compilers for the desktop and server environment, such as Sun Hotspot [26], IBM J9 [31], Open Runtime Platform [12], etc. It has been studied extensively in compilation for dynamic scripting languages, such as PyPy [9] for Python, SPUR [7] and TraceMonkey [16] for JavaScript, and LuaJIT [35] for Lua. HotpathVM [17] and Maxpath [8] are JITs that target Java. HotpathVM introduces a tree-SSA representation for fast analysis and efficient enough for resource-constrained devices. Maxpath is a trace-JIT without an interpreter. It uses a method-based non-optimizing compiler to select hot trace and compiles the trace with an optimizing trace compiler. Maxpath achieves better performance than the original method-JIT. *Swift* can use the same technique in Maxpath to improve the performance. Guo et al.[19] study the soundness of traditional method-based optimizations on trace compilation (and vice versa), while Hiroshi et al.[20] describe their work of developing a trace-JIT by retrofitting a method-JIT. All the JITs described above are all based on stack-based bytecode.

Stack-based versus Register-based. The stack-based architecture and the register-based architecture are two counterparts in the design of machine instruction set architecture [25, 29]. As the most popular virtual machine, JVM uses a virtual stack architecture [18] for evaluating expressions, rather than the register architectures that are commonly used in real processors. In [13], Davis et al. first propose the virtual register machine architecture for JVM and compare the stack-based bytecode with the register-based bytecode. The result shows that the register-based format reduces the total instruction count by 34.88%, while increasing the bytecode size by 44.81%. Shi et al. [30] extend the previous work by implementing a register-based JVM and compare the performance of these two kinds of bytecode. They find that the register-based format reduces the execution time by 26.5% on a switch-based C interpreter [14, 15, 27]. Although JITC-Droid in Android works on register-based bytecode, it doesn't exploit the similarity between register-based bytecode and the underlying register-based architecture commonly found in physical machines, but acts like a traditional stack-based JIT compiler [12, 26, 31]. To the best of our knowledge, this paper is the first to concern the JIT construction with the insight of the opportunity created by register-based bytecode.

Mobile Code System. Mobile code is the program that can be shipped unchanged and executed on a heterogeneous collection of processors. Java was originally used as mobile code in browsers to support rich web applications. To enforce safety, traditional mobile code system sacrifices performance through abstract machine interpretation. In [5] Adl-Tabatabai et al. propose OmniVM, an efficient and universal mobile code system. OmniVM uses dynamic code generation to improve performance and software fault isolation to enforce safety. Although the instruction set in OmniVM is also register-based, it is quite different from register-based Java bytecode, such as DEX. First of all, OmniVM has limited (32) registers while the amount of registers in DEX is unlimited. Second, OmniVM only support several primitive types while DEX support complete type system of Java source code and opaque object layout. Third, DEX has many Java semantics, such as thread synchronization, exception handling, etc. Fourth, DEX has no separated floating-point registers while OmniVM has 16 special floating-point registers. From the perspective of intermediate representation level, OmniVM instructions set is much like a common hardware instruction set, and lower than Java bytecode. Observing the opportunity of register-based Java bytecode, *Swift* constructs a lightweight JIT compiler without sacrificing code quality.

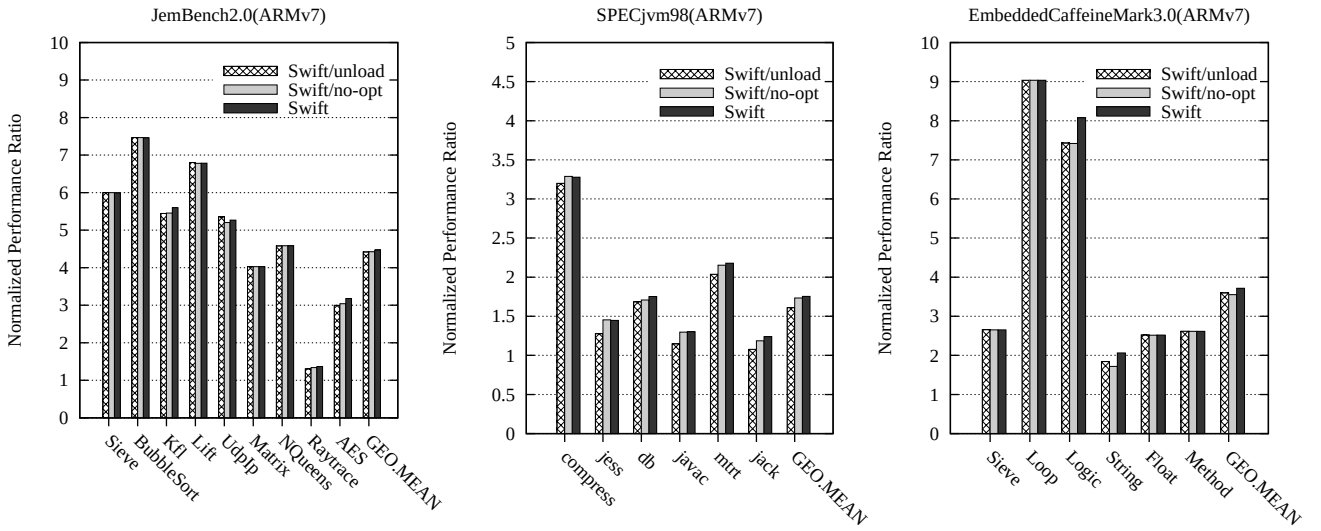

Figure 8. Performance impact of code unloading in *Swift* for JemBench2, SPECjvm98, and EmebeddedCaffeineMark3 on OMAP3530. The code unloader is configured to unload code when the code size is larger than 200KB.

7. Conclusions and Discussions

Since mobile computing becomes a new trend, JIT techniques for embedded JVMs attract more attention. However, JIT compilers suffer significant overhead when compiling stack-based Java bytecode to heterogeneous register-based machine code. This paper presents *Swift*, a fast and effective JIT compiler specifically for register-based bytecode. Taken the characteristics of Java methods and the similarity between register-based bytecode and RISC ISA into account, *Swift* proposes a new translation technique based on register mapping. The translation is very fast and code quality is guaranteed by the static compiler which is used to compile Java source code to register-based bytecode. The evaluation on two ARM SOCs shows the overall performance of *Swift* is competitive, indicating that *Swift* is promising.

Besides, our proposed compilation technique can be further optimized. To support all the ARM variants, current prototype of *Swift* generates ARM instructions which is not the best choice on armv7 architecture. Jazelle RCT [3] is a special processor mode, making small changes to the Thumb2 extended Thumb instruction set. These changes make the instruction set particularly suitable for dynamic code generation system in managed execution environments. It allows JIT compilers to generate smaller compiled code without impacting performance. By exploiting the Jazelle RCT instruction set, *Swift* can gain a further performance speedup with code size reduced. In section 6, we have discussed the possibility of combining Swift with a trace-based JIT. Swift can also be combined with mature staged/selective compilation technique. In such multiple-compiler system, Swift acts as a non-optimizing compiler which compiles all the methods before they are executed and heavy optimizations can be performed by another JIT compiler with a hot method detector. Possible optimizations include SIMD vectorization, method inline, etc. Targeting embedded devices, the detector should have high detection accuracy because the penalty of aggressively optimizing a false hot method is unacceptable and it should not add significant overhead to the application.

Discussion. Previous research [13, 30] has showed that register-based interpreter has performance advantages over stack-based interpreter. As a complement of previous research, our work has demonstrated that register-based bytecode also has advantages to construct a lightweight JIT with good performance. Besides, the register-based JIT proposed by our paper is not only applicable to embedded JVMs, but also can be used as the non-optimizing compiler in a staged/selective compiler system to reduce the startup time and boost normal applications with flat execution profile.

The good performance of our proposed translator shows that, the homogeneity between register-based bytecode and register-based architecture can be exploited to improve the performance of Just-In-Time compiler. We think register-based bytecode is a new responsibility division between online dynamic compiler and offline static compiler, and it is a good balance between Ahead-Of-Time compiler [6, 33] and Just-In-Time compiler which can enjoy good performance without losing portability. So register-based bytecode may be a better choice to distribute applications than stack-based bytecode, especially in embedded systems.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and feedback. This work was funded by Ministry of Industry and Information Technology of China under grant numbered 2009ZX01036-001-003.

References

[1] Dalvik executable format. http://source.android.com/tech/dalvik/dex-format.html.

[2] Embeddedcaffeinemark3.0. Pendragon Software Corporation.

[3] Jazelle rct technology. ARM Ltd., http://www.arm.com/products/processors/technologies/jazelle.php.

[4] Specjvm98. Standard Performance Evaluation Corporation.

[5] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 127–136, New York, NY, USA, 1996. ACM.

[6] C. Badea, A. Nicolau, and A. V. Veidenbaum. A simplified java bytecode compilation system for resource-constrained embedded processors. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 218–228, New York, NY, USA, 2007. ACM.

[7] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. Spur: a trace-based jit compiler for cil. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010. ACM.

[8] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010. ACM.

[9] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[10] D. Bornstein. Dalvik vm internals. http://sites.google.com/site/io/dalvik-vm-internals.

[11] B. Cheng and B. Buzbee. A jit compiler for android's dalvik vm. http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html.

[12] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The open runtime platform: a flexible high-performance managed runtime environment: Research articles. *Concurr. Comput. : Pract. Exper.*, 17:617–637, April 2005.

[13] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, IVME '03, pages 41–49, New York, NY, USA, 2003. ACM.

[14] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 278–288, New York, NY, USA, 2003. ACM.

[15] M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:2003, 2003.

[16] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[17] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM.

[18] J. Gosling. Java intermediate bytecodes. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 111–118, New York, NY, USA, 1995. ACM.

[19] S. Guo and J. Palsberg. The essence of compiling with traces. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 563–574, New York, NY, USA, 2011. ACM.

[20] I. Hiroshi, H. Hiroshige, W. Peng, and N. Toshio. A trace-based java jit compiler retrofitted from a method-based compiler. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '11, New York, NY, USA, 2011. ACM.

[21] D.-H. Jung, S.-M. Moon, and H.-S. Oh. Hybrid java compilation and optimization for digital tv software platform. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 73–81, New York, NY, USA, 2010. ACM.

[22] A. Krall. Efficient javavm just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 205–, Washington, DC, USA, 1998. IEEE Computer Society.

[23] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. *SIGPLAN Not.*, 36:156–167, May 2001.

[24] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 156–167, New York, NY, USA, 2001. ACM.

[25] G. J. Myers. The case against stack-oriented instruction sets. *SIGARCH Comput. Archit. News*, 6:7–10, August 1977.

[26] M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *In USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[27] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 150–159, New York, NY, USA, 1996. ACM.

[28] M. Schoeberl, T. B. Preusser, and S. Uhrig. The embedded java benchmark suite jembench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 120–127, New York, NY, USA, 2010. ACM.

[29] P. U. Schulthess and E. P. Mumprecht. Reply to the case against stack-oriented instruction sets. *SIGARCH Comput. Archit. News*, 6:24–27, December 1977.

[30] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual machine showdown: stack versus registers. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 153–163, New York, NY, USA, 2005. ACM.

[31] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Syst. J.*, 39:175–193, January 2000.

[32] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, IVME '03, pages 50–57, New York, NY, USA, 2003. ACM.

[33] C.-S. Wang, G. Perez, Y.-C. Chung, W.-C. Hsu, W.-K. Shih, and H.-R. Hsu. A method-based ahead-of-time compiler for android applications. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 15–24, New York, NY, USA, 2011. ACM.

[34] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. Latte: A java vm just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 128–, Washington, DC, USA, 1999. IEEE Computer Society.

[35] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09, pages 79–88, New York, NY, USA, 2009. ACM.

[36] L. Zhang and C. Krintz. Adaptive code unloading for resource-constrained jvms. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 155–164, New York, NY, USA, 2004. ACM.

[37] L. Zhang and C. Krintz. Profile-driven code unloading for resource-constrained jvms. In *Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, PPPJ '04, pages 83–90. Trinity College Dublin, 2004.