Effective Software

Lecture 2: Virtual machine, JVM bytecode, (de-)compilers, disassembler, profiling

David Šišlák

david.sislak@fel.cvut.cz

» **Virtual machine** model (.NET, JVM – Scala, Jython, JRuby, Clojure, …)
  • source code
  • compiled into VM *bytecode*
  • hybrid run-time environment (platform dependent VM implementation)
    – interpreted *bytecode*
    – *compiled assembly-code* (native CPU code)
    – automated platform capability optimizations (e.g. use of SIMD)

» comparison of *bytecode* to *assembly-code*
  • (+) platform independence (portable) – architecture (RISC/CISC, bits), OS
  • (+) reflection – observe, modify own structure at run-time
  • (+) small size
  • (-) slower execution – interpreted mode, compilation latencies
  • (-) less control on assembly code – less options for custom optimization
JAVA Virtual Machine – Memory Layout

Stack

Thread
- Program Counter
- Stack
- Native Stack

Frame
- Return Value
- Local Variables
- Operands Stack
- Current Class Constant Pool Reference

Non Heap

Other native memory

Metaspace
- Code Cache
- Method Area

Heap (Intern Strings)

Young Generation
- Eden Space
- Survivor Spaces

Old / Tenured Generation
- Minor Garbage Collection
- Major Garbage Collection

Thread specific

Shared by many threads
JAVA Virtual Machine - Frame

» frame
  » each thread has stack with frames (outside of heap, fixed length)
    StackOverflowError vs. OutOfMemoryError
  » frame is created each time method is invoked (destroyed after return)
    - *interpreted frame* per exactly one method
    - *compiled frame* includes all in-lined methods
  » frame size determined at compile-time (in class file for interpreted)
  » variables (any type)
    » {this} – *instance call only!*
    » {method arguments}
    » {local variables}
  » operand stack (any type)
    » LIFO
  » reference to run-time
    constant pool (class def)
    » method + class is associated
References in JVM are called Ordinary Object Pointers (OOP)
- **compressed** – 32 bit – able to address 32GB heap (using object alignment)
- **regular** – 64 bit
JAVA Virtual Machine – Stack-oriented Machine

- stack-oriented - stack machine model for passing parameters and output for instructions

\[(2 + 3) \times 11 + 1\]

JVM bytecode – sequence of *instructions* composed of
- *opcode* – operation code, what should be done
- opcode specific *parameters* – some has no params, some multiple
JAVA Virtual Machine – Opcodes

» **JVM opcode** (1 Byte only always):
  » load and store (aload_0, istore, aconst_null, ...)
  » arithmetic and logic (ladd, fcmpl, ...)
  » type conversion (i2b, d2i, ...)
  » object manipulation (new, putfield, getfield, ...)
  » stack management (swap, dup2, ...)
  » control transfer (ifeq, goto, ...)
  » method invocation (invokespecial, areturn, ...) – frame manipulation
  » exceptions and monitor concurrency (athrow, monitorenter, ...)

» prefix/suffix – i, l, s, b, c, f, d and a (reference)

» variables as registers – e.g. istore_1 (variable 0 is **this** for instance method)

```
mov    %rax,%r8
shl    $0x5,%eax
sub    %r8d,%eax
add    %ecx,%eax
inc    %edx
```  

**VS.**

```
iconst_0
istore_3
iload_3
bipush 100
```  

**CPU assembly-code**  
**AT&T syntax**  
**JVM bytecode**
JAVA Virtual Machine – Object Oriented Language

» **Class file** – product of source code compilation
  * one per each class
  * method bytecode is included

```java
ClassFile {
  u4               magic;
  u2               minor_version;
  u2               major_version;
  u2               constant_pool_count;
  cp_info          constant_pool[constant_pool_count - 1];
  u2               access_flags;
  u2               this_class;
  u2               super_class;
  u2               interfaces_count;
  u2               interfaces[interfaces_count];
  u2               fields_count;
  field_info       fields[fields_count];
  u2               methods_count;
  method_info      methods[methods_count];
  u2               attributes_count;
  attribute_info   attributes[attributes_count];
}
```
```java
public class Employee<Type> {
    private Type data;
    public int id;

    public Employee(Type data, int id) {
        update(data, id);
    }

    private void update(Type data, int id) {
        this.data = data;
        this.id = id;
    }

    public Type employeeData() {
        return data;
    }
}
```
javap – JAVA disassembler included in JDK (readable form of class file)

```
  public class employee.Employee-Type extends java.lang.Object extends java.lang.Object
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
  Constant pool:
  #1 = Methodref      #6.#25   // java/lang/Object."<init>":(O)V
  #2 = Methodref      #5.#26   // employee.Employee.update:(Ljava/lang/Object;I)V
  #3 = Fieldref       #5.#27   // employee.Employee.data:Ljava/lang/Object;
  #4 = Fieldref       #5.#28   // employee.Employee.id:I
  #5 = Class           #29     // employee/Employee
  #6 = Class           #30     // java/lang/Object
  #7 = Utf8            data
  #8 = Utf8            Ljava/lang/Object;
  #9 = Utf8            Signature
  #10 = Utf8           TType;
  #11 = Utf8           id
  #12 = Utf8           I
  #13 = Utf8           <init>
  #14 = Utf8           (Ljava/lang/Object;I)V
  #15 = Utf8           Code
  #16 = Utf8           LineNumberTable
  #17 = Utf8           (TType;I)V
  #18 = Utf8           update
  #19 = Utf8           employeeData
  #20 = Utf8           ()Ljava/lang/Object;
  #21 = Utf8           ()TType;
  #22 = Utf8           <Type:Ljava/lang/Object;>Ljava/lang/Object;
  #23 = Utf8           SourceFile
  #24 = Utf8           Employee.java
  #25 = NameAndType    #13:#31   // "<init>":(O)V
  #26 = NameAndType    #18:#14   // update:(Ljava/lang/Object;I)V
  #27 = NameAndType    #7:#8     // data:Ljava/lang/Object;
  #28 = NameAndType    #11:#12   // id:I
  #29 = Utf8           employee/Employee
  #30 = Utf8           java/lang/Object
  #31 = Utf8           ()V

  Signature: #22      // <Type:Ljava/lang/Object;>Ljava/lang/Object;
```
» **descriptor** is used by VM – no generics included

» **signature** is used for compilation – contain Generics
JAVA Virtual Machine – Example 1 – Disassembled Method

```
public Type employeeData();
descriptor: ()Ljava/lang/Object;
flags: ACC_PUBLIC
Code:
  stack=1, locals=1, args_size=1
  0: aload_0
  1: getfield  #3 // Field data:Ljava/lang/Object;
  4: areturn

LineNumberTable:
  line 17: 0
Signature:  #21 // ()TType;
```

» **getfield**
  - takes 1 ref from stack
  - build an index into runtime pool of class instance by reference **this**

» **areturn**
  - takes 1 ref from stack
  - push onto the stack of calling method

opcode offset in bytecode for the method `employeeData`
public employee.Employee(Type, int);
descriptor: (Ljava/lang/Object;I)V
flags: ACC_PUBLIC
Code:
  stack=3, locals=3, args_size=3
  0: aload_0
  1: invokespecial #1               // Method java/lang/Object."<init>"():V
  4: aload_0
  5: aload_1
  6: iload_2
  7: invokespecial #2              // Method update:(Ljava/lang/Object;I)V
  10: return
LineNumberTable:
  line 7: 0
  line 8: 4
  line 9: 10
Signature: #17                    // (Ljava/lang/Object;I)V

private void update(Type, int);
descriptor: (Ljava/lang/Object;I)V
flags: ACC_PRIVATE
Code:
  stack=2, locals=3, args_size=3
  0: aload_0
  1: aload_1
  2: putfield       #3            // Field data:Ljava/lang/Object;
  5: aload_0
  6: iload_2
  7: putfield       #4            // Field id:I
  10: return
LineNumberTable:
  line 12: 0
  line 13: 5
  line 14: 10
Signature: #17                    // (Ljava/lang/Object;I)V
JAVAX Virtual Machine – Example 1 – Decompiler

» procyon – open-source JAVA decompiler

```java
package employee;

public class Employee<Type> {
    private Type data;
    public int id;

    public Employee(final Type type, final int n) {
        this.update(type, n);
    }

    private void update(final Type data, final int id) {
        this.data = data;
        this.id = id;
    }

    public Type employeeData() {
        return this.data;
    }
}
```

Original source code

```java
public class Employee<Type> {
    private Type data;
    public int id;

    public Employee(Type data, int id) {
        update(data, id);
    }

    private void update(Type data, int id) {
        this.data = data;
        this.id = id;
    }

    public Type employeeData() {
        return data;
    }
}
```

De-compiled source code
private static Integer daysInMonth(int month, int year) {
    int retVal;
    switch (month) {
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:                
            retVal=31;
            break;
        case 2:                
            retVal = (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)) ? 29 : 28;   
            break;
        case 4: case 6: case 9: case 11: 
            retVal = 30;
            break;
        default:                
            throw new IllegalArgumentException("Unknown month: " + month);
    }
    return new Integer(retVal);
}

private static int compute() {
    int month = 4;
    int year = 2000;
    int o=0;
    for (int i=0; i<1_000_000; i++) {
        o+=daysInMonth(month, year);
    }
    return o;
}
JAVA Virtual Machine – Example 2 – daysInMonth Bytecode

```java
private static java.lang.Integer daysInMonth(int year, int month) {
    int retVal;
    switch (month) {
        case 1:
            retVal = 31;
            break;
        case 2:
            retVal = (year % 4 == 0 &&
                      (year % 100 != 0 ||
                       year % 400 == 0)) ?
                      29 : 28;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            retVal = 30;
            break;
        default:
            throw new IllegalArgumentException("Unknown month: "+month);
    }
    return new Integer(retVal);
}
```
StackMapTable – define variable types in Frame (variables and stack) at every bytecode jump target position
- first stack map automatically determined from method descriptor
- others are differential updates
JAVA Virtual Machine – Example 2 – compute Bytecode

```
private static int compute() {
    int month = 4;
    int year = 2000;
    int o=0;
    for (int i=0; i<1_000_000; i++) {
        o+=daysInMonth(month, year);
    }
    return 0;
}
```

No optimization during source code compilation!

Interpreted code execution is as inefficient as your source code!!!

bytecode offset 10 is related to for cycle bytecode start where there are 4 ints as local variables and no stack
source code compilation ($source\ code \Rightarrow byte\ code$)

- bytecode is not better than your source code
  - invariants in loop are not removed
- no optimizations like
  - loop unrolling
  - algebraic simplification
  - strength reduction

optionally bytecode could be modified before execution by JVM

- e.g. ProGuard – obfuscator including bytecode optimizations
  - shrinker – compact code, remove dead code
  - optimizer
    - modify access pattern (private, static, final)
    - inline bytecode
      - obfuscator – renaming, layout changes
      - preverifier – ensure class loading

Test yourself
- compute method is simplified
- faster interpretation
- better JIT output

obfuscation = make code difficult to be understood by humans but with the same functionality
JAVA Virtual Machine – Bytecode Compilation in run-time

» **Just-in-time (JIT)**
  » converts bytecode into assembly code in run-time
  » check OpenJDK sources for very detailed information
  [http://openjdk.java.net](http://openjdk.java.net)

» JIT includes **adaptive optimization** (adaptive tiered compilation since version 7)
  » balance trade-off between JIT and interpreting instructions
  » monitors frequently executed parts “hot spots” **including data** on caller-callee relationship for virtual method invocation
  » triggers dynamic re-compilation based on current execution profile
  » inline expansion to remove context switching
  » optimize branches
  » can make risky assumption (e.g. skip code) ->
    » unwind to valid state
    » deoptimize previously JITed code even if code is already executed

» **Ahead-of-Time Compilation (AOT)** – remove warm-up phase
  • compile into assembly code prior to launching the virtual machine
JAVA Virtual Machine – JIT Compilation

» *Just-in-time (JIT) compilers* – asynchronous (3 C1, 7 C2 threads for 32 cores)
  
  » **C1 compiler** – much faster than C2
    
    » simplified inlining, using CPU registers
    
    » window-based optimization over small set of instructions
    
    » intrinsic functions with vector operations **SIMD** (Math, arraycopy, ...)
  
  » **C2 compiler** – high-end fully optimizing compiler
    
    » dead code elimination, loop unrolling, loop invariant hoisting, common sub-expression elimination, constant propagation
    
    » full inlining, full deoptimization (back to level 0)
    
    » escape analysis, null check elimination,
    
    » pattern-based loop vectorization and super word packing (**SIMD**)

» **JIT compilation tiers** – adaptive compilation levels in JVM

```
CompLevel_none     = 0,    // Interpreter
CompLevel_simple   = 1,    // C1
CompLevel_limited_profile = 2, // C1, invocation & backedge counters
CompLevel_full_profile   = 3, // C1, invocation & backedge counters + mdo
CompLevel_full_optimization = 4, // C2
```

» **on-stack replacement** (OSR) – optimization during execution of a method
  
  » start at bytecode jump targets (goto, if_)
Assembly Code

» reasons to study *assembly code* (both Java and C/C++)
  • educational reasons
    – predict efficient coding techniques
  • debugging and verification
    – how well the code looks like
  • optimize code
    1. *for speed*
      • avoid poorly compiled patterns
      • data fits into cache
      • predictable branches or no branches
      • use vector programming if possible (*SIMD*)
        » 256bit registers with AVX2 since Intel Sandy Bridge
        » 512bit AVX-512 since Intel Knight Landing (Xeon Phi)
    2. *for size*
      • primarily code cache efficiency
JAVA Virtual Machine – Example 2 – Tiered Compilation

-XX:+PrintCompilation (-XX:+PrintInlining)

{millis from start} {compilation_task_id} {flags} {tier} {class:method} (bytecode size)@OSR {removing not rentrant/zombie}

67  1  3  java.lang.String::hashCode (55 bytes)
68  2  3  java.lang.String::charAt (29 bytes)
69  3  3  java.lang.String::length (6 bytes)
74  4  3  java.lang.String::indexOf (70 bytes)
74  5  n  0  java.lang.System::arraycopy (native) (static)
74  6  3  java.lang.String::equals (81 bytes)
75  8  3  java.lang.Object::<init> (1 bytes)
75  9  3  java.lang.Math::min (11 bytes)
75  7  3  java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)
75  10  3  java.lang.AbstractStringBuilder::append (50 bytes)
76  11  3  java.lang.String::getChars (62 bytes)
81  12  1  java.lang.ref.Reference::get (5 bytes)
81  13  3  java.lang.StringBuilder::append (8 bytes)
82  14  3  java.lang.String::indexOf (7 bytes)
83  16  3  java.lang.Number::<init> (5 bytes)
83  19  1  java.lang.Object::<init> (1 bytes)
84  8  3  java.lang.Object::<init> (1 bytes) made not rentrant
84  18  3  SwitchTest::daysInMonth (144 bytes)
84  17  3  java.lang.Integer::<init> (10 bytes)
84  15  1  java.lang.Integer::intValue (5 bytes)
84  20  4  SwitchTest::daysInMonth (144 bytes)
86  18  3  SwitchTest::daysInMonth (144 bytes) made not rentrant
88  21  %  3  SwitchTest::compute @ 10 (35 bytes)
88  22  3  SwitchTest::compute (35 bytes)
89  23  %  4  SwitchTest::compute @ 10 (35 bytes)
91  21  %  3  SwitchTest::compute @ -2 (35 bytes) made not rentrant
91  23  %  4  SwitchTest::compute @ -2 (35 bytes) made not rentrant
92  24  %  4  SwitchTest::compute @ 10 (35 bytes)
94  25  4  SwitchTest::compute (35 bytes)
95  22  3  SwitchTest::compute (35 bytes) made not rentrant

Notice standard compilation path 0 -> 3 -> 4
JVM – Example 2 – daysInMonth Assembly Code – Tier 3

-XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly

all examples are in JVM 8 64-bit, Intel Haswell CPU, AT&T syntax

tier 3 - C1 with invocation & backedge counters + MethodDataOop counter because: count="256" iicount="256" hot_count="256"

stack initialization, **invocation counter** in MDO (0xDC) + **trigger** C2 (tier 4)

```
127 17 b 3      SwitchTest::daysInMonth (144 bytes)
Decoding compiled method 0x0000000108d5f910:
Code:
[Entry Point]
[Verified Entry Point]
[Constants]
# {method} {0x000000012169d568} 'daysInMonth' 'IIJjava/lang/Integer;' in 'SwitchTest'
# parm0: rsi = int
# parm1: rdx = int
# [sp+0x30] (sp of caller)
0x0000000108d59380: mov %eax,-0x1400(%rsp)
0x0000000108d59387: push %ebp
0x0000000108d59388: sub $0x80,%rsp
0x0000000108d5938f: mov %edi,%edi
0x0000000108d59392: movabs $0x12169db40,%rax
0x0000000108d5939c: mov %edx,%edx
0x0000000108d593a2: add $0x8,%edx
0x0000000108d593a5: mov %edx,%edx
0x0000000108d593b2: movabs $0x12169d68,%rax
0x0000000108d593b5: and $0xff8,%edx
0x0000000108d593b6: cmp $0x0,%edx
0x0000000108d593b8: len
0x0000000108d593b9: mov $0x8,%edx
0x0000000108d593ba: mov abs $0x0,%edx
0x0000000108d593bb: cmp $0x0,%edx
0x0000000108d593bc: jne 0x0000000108d59996 ;*iLOAD_0
; SwitchTest::daysInMonth@0 (line 7)
```

RSP – current stack position
R15 – current threat meta information
RAX – return value

month, year

stacking banging technique, StackOverflowException

stack frame allocation, saving registers

0x1ff8 >> 3 = 1024 invocations **trigger tier 4 (C2)**
JVM – Example 2 – daysInMonth Assembly Code – Tier 3

```
0x0000000108a953c4: cmp $0x1,%esi
0x0000000108a953c7: je 0x00000000108a95597
0x0000000108a953cd: cmp $0x2,%esi
0x0000000108a953d0: je 0x00000000108a95435
0x0000000108a953d6: cmp $0x3,%esi
0x0000000108a953d9: je 0x00000000108a95597
0x0000000108a953df: cmp $0x4,%esi
0x0000000108a953e2: je 0x00000000108a9557d
0x0000000108a953e8: cmp $0x5,%esi
0x0000000108a953eb: je 0x00000000108a95597
0x0000000108a953f1: cmp $0x6,%esi
0x0000000108a953f4: je 0x00000000108a9557d
0x0000000108a953fa: cmp $0x7,%esi
0x0000000108a953fd: je 0x00000000108a95597
0x0000000108a95403: cmp $0x8,%esi
0x0000000108a95406: je 0x00000000108a95597
0x0000000108a9540c: cmp $0x9,%esi
0x0000000108a9540f: je 0x00000000108a9557d
0x0000000108a95415: cmp $0xa,%esi
0x0000000108a95418: je 0x00000000108a95597
0x0000000108a9541e: cmp $0xb,%esi
0x0000000108a95421: je 0x00000000108a9557d
0x0000000108a95427: cmp $0xc,%esi
0x0000000108a9542a: je 0x00000000108a95597
0x0000000108a95430: jmpq 0x00000000108a956d0 ;*tableswitch
 ; - SwitchTest::daysInMonth@1 (line 7)
```
JVM – Example 2 – daysInMonth Assembly Code – Tier 3

target for month=4, **backedge counter** tracking in MDO (0x290):  

```
0x0000000108a9557d: movabs $0x12169db40,%rdx ; {metadata(method data for [method] {0x000000012169d568} 'daysInMonth' '(II)java/lang/Integer;' in 'SwitchTest')}
0x0000000108a95587: incl $0x290(%rdx)
0x0000000108a9558d: mov $0x1e,%ebx
0x0000000108a95592: jmpq 0x0000000108a955ac ;*goto
; - SwitchTest::daysInMonth@105 (line 26)
```

EBX=30 is retVal  
jump target, **inlined TLAB allocation** of Integer object:

```
0x0000000108a955ac: movabs $0x7c0011320,%rdx ; {metadata('java/lang/Integer')}
0x0000000108a955b6: mov $0x60(%r15),%rax
0x0000000108a955ba: lea $0x10(%rax),%rdi
0x0000000108a955be: cmp $0x70(%r15),%rdi
0x0000000108a955c2: ja 0x0000000108a955bc
0x0000000108a955c8: mov %rdi,0x60(%r15)
0x0000000108a955cc: mov $0x8(%rdx),%rcx
0x0000000108a955d3: mov %rcx,%rax
0x0000000108a955d6: mov %dx,%rcx
0x0000000108a955d9: shr $0x3,%rcx
0x0000000108a955dd: mov %rcx,0x8(%rax)
0x0000000108a955df: xor %rcx,%rcx
0x0000000108a955e8: mov %rcx,0xc(%rax)
0x0000000108a955e3: mov %rcx,%rcx
0x0000000108a955e6: xor %rcx,%rcx
;*new ; - SwitchTest::daysInMonth@135 (line 30)
```

**0x10 Integer instance size**  
no space in **TLAB** -> new TLAB + external allocation  
with header init returns after the inlined allocation  
**object initialization**, header filed with prototype mark

RAX Integer instance address

Heap object structure (64-bit JVM):  
- header 12 or 16 Bytes  
- object data super class first, type grouped

<table>
<thead>
<tr>
<th>8B - mark word</th>
</tr>
</thead>
<tbody>
<tr>
<td>4B / 8B – Klass ref.</td>
</tr>
<tr>
<td>... object data</td>
</tr>
</tbody>
</table>

Heap array object structure (64-bit JVM):  
- header 16 or 20 Bytes  
- sequence of array values

<table>
<thead>
<tr>
<th>8B - mark word</th>
</tr>
</thead>
<tbody>
<tr>
<td>4B / 8B – Klass ref.</td>
</tr>
<tr>
<td>4B – array length</td>
</tr>
<tr>
<td>sequence of values</td>
</tr>
</tbody>
</table>
inlined Integer constructor with supers, invocation counts in MDOs (0xDC)

- Integer::<init>, Number::<init>, Object::<init>

  - currently in tier 3 (C1 counters in MDO)

```assembly
0x0000000108a95e9: mov %rax,%rdx
0x0000000108a95ec: movabs 0x12169db40,%rdi
0x0000000108a95f6: addq $0x1,0x358(%rsi)
0x0000000108a95fe: movabs 0x1214df850,%rdx
0x0000000108a9608: mov 0xdc(%rdx),%esi
0x0000000108a960e: add %edx,%esi
0x0000000108a9611: mov %esi,0xdc(%rdx)
0x0000000108a9617: movabs 0x12134738,%rdx
0x0000000108a9621: and $0x7ffff8,%esi
0x0000000108a9627: cmp $0x0,%esi
0x0000000108a9632: je 0x0000000108a9599c9
0x0000000108a9633: mov %rax,%rdx
0x0000000108a963c: movabs 0x1214df850,%rsi
0x0000000108a9645: mov 0xdc(%rdx),%esi
0x0000000108a964f: mov 0xdc(%rdx),%esi
0x0000000108a9655: add %edx,%esi
0x0000000108a9658: mov %esi,0xdc(%rdx)
0x0000000108a965e: movabs 0x12133a9d8,%rdx
0x0000000108a9668: and $0x7ffff8,%esi
0x0000000108a966e: cmp $0x0,%esi
0x0000000108a9671: je 0x0000000108a9599e0
0x0000000108a9677: mov %rax,%rdx
0x0000000108a967e: movabs 0x1214df720,%rsi
0x0000000108a9684: addq $0x1,0x108(%rsi)
0x0000000108a968c: movabs 0x1214df720,%rsi
0x0000000108a9690: mov 0xdc(%rdx),%esi
0x0000000108a9696: mov 0xdc(%rdx),%esi
0x0000000108a969c: add %edx,%esi
0x0000000108a969f: mov %esi,0xdc(%rdx)
0x0000000108a96a5: movabs 0x12192df8,%rdx
0x0000000108a96a9: mov 0xdc(%rdx),%esi
0x0000000108a96af: mov 0xdc(%rdx),%esi
0x0000000108a96b5: cmp $0x0,%esi
0x0000000108a96b7: je 0x0000000108a959f7f
0x0000000108a96be: mov $%ecx,0x8(%rax)

{metadata(method data for [method] {0x000000012169db40} 'daysInMonth' 'II' 'java/lang/Integer'; in 'SwitchTest')}

{metadata(method data for [method] {0x000000012134738} 'init' 'IV' in 'java/lang/Integer')}

{metadata(method data for [method] {0x000000012134738} 'init' 'IV' in 'java/lang/Integer')}

{metadata(method data for [method] {0x000000012133a9d8} 'init' 'IV' in 'java/lang/Number')}

{metadata(method data for [method] {0x000000012133a9d8} 'init' 'IV' in 'java/lang/Number')}

{metadata(method data for [method] {0x000000012133a9d8} 'init' 'IV' in 'java/lang/Number')}

{metadata(method data for [method] {0x000000012139d480} 'init' 'IV' in 'java/lang/Object')}

{metadata(method data for [method] {0x000000012139d480} 'init' 'IV' in 'java/lang/Object')}

{metadata(method data for [method] {0x000000012139d480} 'init' 'IV' in 'java/lang/Object')}

RAX.value = EBX (retVal)
```
final cleanup and return, RAX contains return value (pointer to Integer instance)

» **Ordinary Object Pointer (Oop)** – flexible reference to an object

» **safepoint** – Oops in perfectly described state by OopMap (GCmaps)
  - Oop can be *safely manipulated externally* while thread is suspended
  - in interpreted mode – between any 2 byte codes
  - in C1/C2 compiled – end of all methods (not in-lined), non-counted loop back edge, during JVM run-time call
  - parked, blocked on IO, monitor or lock
  - while running JNI (do not need thread suspension)
  - **global safepoint (all threads)** – stop the world
    - GC, print threads, thread dumps, heap dump, get all stack trace
    - enableBiasedLocking, RevokeBias
    - class redefinition (e.g. instrumentation), debug
  - **local safepoint (just executing thread)**
    - de-optimization, enable/revoke bias locking, OSR
JVM – Time To Safe Point

» **Time To Safe Point** (TTSP) – how long it takes to enter safepoint

-XX:+PrintSafepointStatistics -XX:+PrintGCApplicationStoppedTime -XX:PrintSafepointStatisticsCount=1

TTSP overhead in profiler while calling GetStackTrace example with 5 threads:
JVM – Example 2 – daysInMonth Assembly Code – Tier 4

tier 4 – C2 compiler – no profile counters
because: count="5376" iicount="5376" hot_count="5376"

stack initialization, use lookup table jump for table switch

```
[Entry Point]

# {method} {0x00000012169d568} 'daysInMonth' '(II)java/lang/Integer;' in 'SwitchTest'
# parm0: rsi = int
# parm1: rdx = int
# [esp+0x20] (sp of caller)
0x000000108a97020: mov %eax, -0x14000(%rsp); {no_reloc}
0x000000108a97027: push %rbp
0x000000108a97028: sub $0x10,%rsp
; *synchronization entry
; - SwitchTest::daysInMonth@1 (line 7)
0x000000108a9702c: mov %esi,%rll1d
0x000000108a9702f: dec %rll1d
0x000000108a97032: cmp $0xc,%rll1d
0x000000108a97036: jae 0x000000108a9704a
0x000000108a97038: movslq %esi,%r10
0x000000108a9703b: movabs $0x108ad96f0,%r11
0x000000108a97045: jmpq *(0x8(0x11,0x10,8)); *tablesswitch
; - SwitchTest::daysInMonth@1 (line 7)
```
target for month=4

Integer.<init>, Number.<init>, Object.<init> - iicount="5376" -> Inline (hot) optimized branching, inlined TLAB allocation, inlined constructors, no nulling, caching optimization

```
0x0000000108a9708a: mov $0x1e,%ebp ;*goto EBP=30 is retVal
    ; - SwitchTest::daysInMonth@105 (line 26)

0x0000000108a9708f: mov 0x60(%r15),%rax
0x0000000108a97093: mov %rax,%r10
0x0000000108a97096: add $0x10,%r10
0x0000000108a9709a: cmp 0x70(%r15),%r10
0x0000000108a9709e: jae 0x0000000108a97124
0x0000000108a970a4: mov %r10,0x60(%r15)
0x0000000108a970a8: prefetchnta 0xc0(%r10)
0x0000000108a970b0: mov $0xf8002264,%r10d ; {metadata('java/lang/Integer')}
0x0000000108a970b6: shl $0x3,%r10
0x0000000108a970ba: mov %xa8(%r10),%r10
0x0000000108a970c1: mov %r10,(%rax)
0x0000000108a970c4: movl $0xf8002264,0x8(%rax) ;*new compressed OOP to Integer class
    ; - SwitchTest::daysInMonth@135 (line 30)
    ; {metadata('java/lang/Integer')}
0x0000000108a970cb: mov %ebp,0xc(%rax) ;*synchronization entry RAX.value = EBX (retVal)
    ; - SwitchTest::daysInMonth@-1 (line 7)

0x0000000108a970ce: add $0x10,%rsp
0x0000000108a970d2: pop %rbp
0x0000000108a970d3: test %eax,-0x214e0d9(%rip) # 0x0000000106949000
    ; {poll_return}

0x0000000108a970d9: retq
```

TLAB Integer object allocation, ref in RAX

cache optimization 3 cache lines ahead

MarkWord fetch from class and then store compressed OOP to Integer class

RAX.contains return value (pointer to Integer instance)
target for default

class IllegalArgumentException no profile -> uncommon -> reinterpret

remap inputs, return back to reinterpreter

```
x00000000108a9704a: mov  %esi,%ebp
x00000000108a9704c: mov  $0x2,%esi
x00000000108a97051: xchg %ax,%ax
x00000000108a97053: callq 0x0000000010898b1a0 ; OopMap{off=56}
    ;*new ; - SwitchTest::daysInMonth@108 (line 28)
    ;{runtime_call}

x00000000108a97058: callq 0x00000000107e7e33c ;*new
    ; - SwitchTest::daysInMonth@108 (line 28)
    ;{runtime_call}
```

then tier 3 code version is discarded

138 17 3 SwitchTest::daysInMonth (144 bytes) made not entrant
OSR @10 – On Stack Replacement at bytecode 10

**tier 4 – C2** (before there was tier 3 OSR @10 because 60416 loops and tier 3)
because: backedge_count="101376" hot_count="101376"

copy 4 locals on stack from tier3 OSR @10 to regs

```java
private static int compute() {
    int month = 4;
    int year = 2000;
    int o = 0;
    for (int i = 0; i < 1_000_000; i++) {
        o += daysInMonth(month, year);
    }
    o = daysInMonth(month, year);
    return o;
}
```

RSI compiled stack of tier 3 OSR @10
then there is **inlined** tier 4 `daysOfMonth` (lookup jump) because the call is **hot**
ending with addition into accumulator `o`

reinterpret on end of cycle jump (unstable if bytecode), save 3 locals to stack
JVM – Example 2 – compute Assembly Code – Tier 4

tier 4 – C2

because: count="2" backedge_count="150528"

use combination of **full inline**, **dead code elimination**, **object escape**, **loop invariant hoisting**, **strength reduction**

```
157 23 b 4 SwitchTest::compute (35 bytes)
Decoding compiled method 0x0000000108a97f90:
Code:
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} {0x000000012169d638} 'compute' '()' in 'SwitchTest'
  # [sp+0x20] (sp of caller)
  0x0000000108a980c0: sub $0x18,%rsp
  0x0000000108a980c7: mov %rbp,0x10(%rsp) ;*synchronization entry
    ; - SwitchTest::compute@-1 (line 34)

  0x0000000108a980cc: mov $0x1c9c380,%eax ——— 30 000 000
  0x0000000108a980d1: add $0x10,%rsp
  0x0000000108a980d5: pop %rbp
  0x0000000108a980d6: test %eax,-0x214f0dc(%rip) # 0x0000000106949000
    ; {poll_return}
  0x0000000108a980dc: retq RAX contains **return value** (primitive int)
```

25th February 2019
Java Virtual Machine – Performance

– requires warm-up to utilize benefits of C2 (or C1)
– compilers cannot do all magic -> write better algorithms

– 32-bit vs 64 bits JVMs
  • 32-bit (max ~3GB heap)
    – smaller memory footprint
    – slower long & double operations
  • 64-bit max 32GB virtual memory (with default ObjectAlignmentInBytes)
    - faster performance for long&double
    – slight increase of memory footprint
    – compressed OOPs are slightly slower for references upon usage
    – compressed OOPs less memory -> less frequent GC -> faster program
  • 64-bit >32GB virtual memory (large heap)
    – fast reference usage
    – wasting a lot of memory (48GB ~32GB with compressed OOPs)
Java Virtual Machine – CPU and Memory Profiling

» profiling
  • CPU – time spent in methods
  • memory – usage, allocations

» modes
  • sampling
    – periodic sampling of stacks of running threads
    – no invocation counts, no 100% accuracy (various sampling errors)
    – no bytecode (& assembly code) modifications
    – 1-2% impact to standard performance (TTSP, thread dumps, analysis)
  • tracing (instrumentation) - method entry, exit, traceObjAllocations
    – instrumented bytecode -> affected performance -> affected compiler optimizations

» jvisualvm
  • JVM monitoring, troubleshooting and profiling tool
  • included in all JDKs
JVM – Example 2 – CPU Tracing of daysOfMonth

assembly code of tier 4 – C2 (before there was very complex tier 3)

inlined daysInMonth rootMethodEntry tracking

```
# [method] {0x0000000012489e38} 'daysInMonth' '([II)java/lang/Integer;' in 'SwitchTest'
# parm0:  rsi = int
# parm1:  rdx = int
#
#                   (sp of caller)
0x000000010c08aa80: mov %eax,-0x14000(%rsp) ; {no_reloc}
0x000000010c08aa87: push %rbp
0x000000010c08aa88: sub $0x60,%rsp ;*synchronization entry
                     - SwitchTest::daysInMonth@-1 (line 7)
0x000000010c08aa8c: mov %edx,0x4(%rsp)
0x000000010c08aa90: mov %esi,(%rsp)
0x000000010c08aa93: movabs $0x76c73c180,0x10
0x000000010c08aa9d: movzb 0x82(0x10),0x11d
                ;getstatic recursiveInstrumentationDisabled
                     - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUTime::rootMethodEntry@0 (line 189)
                     - SwitchTest::daysInMonth@3 (line 7)
0x000000010c08aa95: test %r11d,%r11d
0x000000010c08aa98: jne 0x000000010c08b075 ;*ifeq
                     ;- org.netbeans.lib.profiler.server.ProfilerRuntimeCPUTime::rootMethodEntry@3 (line 189)
                     ;- SwitchTest::daysInMonth@3 (line 7)
0x000000010c08aaae: movabs $0x76c73e220,0x10
0x000000010c08aab8: mov %ebx,0x10
                ;getstatic lastThreadInfo
                     - org.netbeans.lib.profiler.server.ThreadInfo::getThreadInfo@84 (line 244)
                     - org.netbeans.lib.profiler.server.ProfilerRuntimeCPUTime::rootMethodEntry@7 (line 193)
                     - SwitchTest::daysInMonth@3 (line 7)
0x000000010c08aabc: mov 0x40(%r12,%r8,8),%ebp ;getfield thread
                     ;- org.netbeans.lib.profiler.server.ThreadInfo::getThreadInfo@89 (line 246)
                     ;- org.netbeans.lib.profiler.server.ProfilerRuntimeCPUTime::rootMethodEntry@7 (line 193)
```

749 Bytes of assembly code for each rootMethodEntry
additional **rootMethodEntry** and **rootMethodExit** trackings for `Integer::<init>` and `Number::<init>`

inlined rootMethodExit after Integer instance.value = retVal

313 Bytes of assembly code for each **rootMethodEntry**
JVM – Example 2 – CPU Tracing Outcome

![Image of JVM Profiling with VisualVM]

**Profiler Snapshot**

- **Call Tree – Method**
  - RMI TCP Connection(idle) [83,120 ms (100%)]
  - RMI TCP Connection(idle) [38,035 ms (100%)]
  - main [8,444 ms (100%)]
  - SwitchTest.compute() [8,444 ms (100%)]
  - SwitchTest.daysInMonth(int, int) [5,059 ms (59.9%)]
  - java.lang.Integer.<init>(int) [2,808 ms (33.3%)]
  - java.lang.Number.<init>() [1,750 ms (20.7%)]
  - SwitchTest.waitForAnyInputLine() [0.000 ms (0%)]

**Method Name Filter (Contains)**

**25th February 2019**

**ESW – Lecture 2**
CPU tracing of compute results into much slower code
- no object escape from daysInMonth call
- no invariant hoisting
- no strength reduction (full loop remains there)

Object allocation tracing is similar with traceObjAlloc injected calls

**recommended approach**
- do sampling first
- identify performance bottlenecks (where most time is spent)
  - it could be outside of JVM (e.g. latency of external DB, file system)
- focus with tracing just to identified parts
jmc – JRockit JVM, included in commercial JDKs, sampling in Flight recorder
Approach to Performance Testing

» test real application – ideally the way it is used
  • microbenchmarks – measure very small units
    – warm-up – to measure real code, not compilers itself, biased locks
    • keep in mind caching
    – beware of compilers – use results, reordering of operations
    – synchronization – multi-threaded benchmarks
    – vary pre-calculated parameters affecting complexity – different optimization in reality
  • macrobenchmarks – measure application input/output
    – least performing component affects the whole application

» understand throughput, elapsed and response time
  • outliers can occur – e.g. GC
  • use existing generators than writing own
Approach to Performance Testing

» understand variability – changes over time
  • internal state
  • background effects – load, network
  • probabilistic analysis – works with uncertainty

» test early, test often – ideally part of development cycle
  • ideally some properly repeated mesobenchmarking
  • automate tests – scripted
  • proper test coverage of functionality and inputs
  • test on target system – different code on different systems