Read the Hotspot JVM's invokestatic and return instructions

Sep 10, 2019

Last time I checked the frame structure of Hotspot JVM , but I couldn't see the call of the method that uses it and the processing of the part that returns from it.

---

invokestatic

The following methods create native code for invokestatic.

```
// in src/hotspot/cpu/x86/templateTable_x86.cpp

void TemplateTable::invokestatic(int byte_no) {
  prepare_invoke(byte_no, rbx);  // get f1 Method*
  // do the call
  __ profile_call(rax);
  __ profile_arguments_type(rax, rbx, rbcp, false);
  __ jump_from_interpreted(rbx, rax);
}
```

If you ignore around profile (method data pointer)

- prepare_invoke
  - Save register information you want to save in frame
  - Prepare registers
  - Prepare return address
    - It becomes the jmp destination when interpret of the method of invoke target ends.
    - `-XX:+PrintInterpreter` Somewhere within the invoke return entry points of
- jump_from_interpreted
  - Set rsp to last sp of current frame
    - last sp is 0x00 when the frame is created, and is set at this timing
  - jmp

Such processing is performed.

prepare_invoke

Since it is difficult to follow the whole, only the important parts related to invokestatic are extracted.

```
// in src/hotspot/cpu/x86/templateTable_x86.cpp

void TemplateTable::prepare_invoke(int byte_no,
                                   Register method,  // linked method (or i-klass)
                                   Register index,   // itable index, MethodType, etc.
                                   Register recv,    // if caller wants to see it
                                   Register flags    // if caller wants to test it
                                   ) {
  ...
  if (flags == noreg)  flags = rdx;
  ...
  // save 'interpreter return address'
  __ save_bcp();
  load_invoke_cp_cache_entry(byte_no, method, index, flags, is_invokevirtual, false, is_invokedynamic);
  ...
  // compute return type
  __ shrl(flags, ConstantPoolCacheEntry::tos_state_shift);
  // Make sure we don't need to mask fjlags after the above shift
  ConstantPoolCacheEntry::verify_tos_state_shift();
  // load return address
```

```
  {
    const address table_addr = (address) Interpreter::invoke_return_entry_table_for(code);
    ExternalAddress table(table_addr);
    LP64_ONLY(__ lea(rscratch1, table));
    LP64_ONLY(__ movptr(flags, Address(rscratch1, flags, Address::times_ptr)));
  }

  // push return address
  __ push(flags);
  ...
}
```

The following processing seems to be performed.

- save_bcp
  - Save bcp register (r13 for x86_64) to bcp area in frame
  - Since only registers are updated during interpret, reflect the latest value here
- load_invoke_cp_cache_entry
  - `Method*`Set invoke target in method register (rbx for x86_64)
- compute return type
  - Extract information corresponding to tos from flags register
- load return address
- push return address
  - The return destination at the end of the method invoked by these two is calculated and pushed on the stack
  - The return destination is managed in a dedicated table and is determined by the type of invoke instruction (eg invokestatic), tos, etc.

jump_from_interpreted

```
void InterpreterMacroAssembler::prepare_to_jump_from_interpreted() {
  // set sender sp
  lea(_bcp_register, Address(rsp, wordSize));
  // record last_sp
  movptr(Address(rbp, frame::interpreter_frame_last_sp_offset * wordSize), _bcp_register);
}

// Jump to from_interpreted entry of a call unless single stepping is possible
// in this thread in which case we must call the i2i entry
void InterpreterMacroAssembler::jump_from_interpreted(Register method, Register temp) {
  prepare_to_jump_from_interpreted();
  ...
  jmp(Address(method, Method::from_interpreted_offset()));
}
```

- Method :: from_interpreted_offset contains, for example, the address of the entry point generated by generate_normal_entry of TemplateInterpreterGenerator
  - Interpret the method after a new frame is prepared here

Let's actually check the behavior so far with gdb. The following Java program is used as an example.

```
class InvokeStatic {
    public static void f(int a, int b) {
        int sum = a + b;
        System.out.println("sum: " + sum);
    }

    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        f(1 + 2, 2 + 2);
    }
}
```

```
$ javap -c -v InvokeStatic.class
...
```

```
  InvokeStatic();
    descriptor: ()V
    flags:
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public static void f(int, int);
    descriptor: (II)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=3, args_size=2
         0: iload_0
         1: iload_1
         2: iadd
         3: istore_2
         4: getstatic     #2     // Field java/lang/System.out:Ljava/io/PrintStream;
         7: new           #3     // class java/lang/StringBuilder
        10: dup
        11: invokespecial #4     // Method java/lang/StringBuilder."<init>":()V
        14: ldc           #5     // String sum:
        16: invokevirtual #6     // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/Strin
        19: iload_2
        20: invokevirtual #7     // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
        23: invokevirtual #8     // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
        26: invokevirtual #9     // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        29: return
      LineNumberTable:
        line 3: 0
        line 4: 4
        line 5: 29

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=3, args_size=1
         0: iconst_1
         1: istore_1
         2: iconst_2
         3: istore_2
         4: iconst_3
         5: iconst_4
         6: invokestatic  #10                 // Method f:(II)V
         9: return
      LineNumberTable:
        line 8: 0
        line 9: 2
        line 10: 4
        line 11: 9
...
```

The state of the stack when the invokestatic that calls the f method from the main method (when the invokestatic native code starts) is as follows. You can see the frame of the main method (sp = stack pointer, fp = frame pointer, bcp = byte code pointer).

```
# breakpoint at the beginning of invokestatic
(gdb) p $rbp
$1 = (void *) 0x7ffff59ed8d8
(gdb) p $rsp
$2 = (void *) 0x7ffff59ed880
(gdb) p ($rbp - $rsp)
$3 = 88
(gdb) x /128xb $rsp
0x7ffff59ed880: 0x04    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # operand stack <- rsp
```

```
0x7ffff59ed888: 0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # operand stack
0x7ffff59ed890: 0x90    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # expression stack
0x7ffff59ed898: 0x48    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00 # bcp
0x7ffff59ed8a0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # pointer to locals
0x7ffff59ed8a8: 0xc8    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00 # constant pool cache
0x7ffff59ed8b0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # methodData
0x7ffff59ed8b8: 0x18    0x54    0x6f    0x19    0x07    0x00    0x00    0x00 # mirror
0x7ffff59ed8c0: 0x60    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00 # Method*
0x7ffff59ed8c8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # last sp
0x7ffff59ed8d0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # old sp 1
0x7ffff59ed8d8: 0x60    0xd9    0x9e    0xf5    0xff    0x7f    0x00    0x00 # old fp 1 <- rbp
0x7ffff59ed8e0: 0xf3    0x09    0x00    0xe1    0xff    0x7f    0x00    0x00 # return address 1
0x7ffff59ed8e8: 0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # local
0x7ffff59ed8f0: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # local
0x7ffff59ed8f8: 0x60    0x59    0x6f    0x19    0x07    0x00    0x00    0x00 # param
```

Below is the state of the stack just before jmp to normal_entry.

```
# breakpoint at jmp (to method entrypoint) of invokestatic
(gdb) p $rbp
$4 = (void *) 0x7ffff59ed8d8
(gdb) p $rsp
$5 = (void *) 0x7ffff59ed878
(gdb) x /136xb $rsp
0x7ffff59ed878: 0xa7    0x9f    0x00    0xe1    0xff    0x7f    0x00    0x00 # return address 2 <- rsp
0x7ffff59ed880: 0x04    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # operand stack <- r13
0x7ffff59ed888: 0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # operand stack
0x7ffff59ed890: 0x90    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed898: 0x4e    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8a0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed8a8: 0xc8    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8b0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed8b8: 0x18    0x54    0x6f    0x19    0x07    0x00    0x00    0x00
0x7ffff59ed8c0: 0x60    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8c8: 0x80    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # last sp
0x7ffff59ed8d0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # old sp 1
0x7ffff59ed8d8: 0x60    0xd9    0x9e    0xf5    0xff    0x7f    0x00    0x00 # old fp 1 <- rbp
0x7ffff59ed8e0: 0xf3    0x09    0x00    0xe1    0xff    0x7f    0x00    0x00 # return address 1
0x7ffff59ed8e8: 0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed8f0: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed8f8: 0x60    0x59    0x6f    0x19    0x07    0x00    0x00    0x00
```

- rsp is set to last sp
    - Points to the address of rsp + word at this point
    - Same value in r13 register
- The calculated return address 2 is pushed on the stack

After this, a new frame is created with normal_entry, and the stack at the time when the f method interpret is started is as follows. In the figure, the area around memory addresses 0x7ffff59ed820-0x7ffff59ed878 is the new frame part.

```
# breakpoint at the beginning of interpretation of method 'f'
(gdb) p $rbp
$6 = (void *) 0x7ffff59ed868
(gdb) p $rsp
$7 = (void *) 0x7ffff59ed820
(gdb) p (((long) 0x7ffff59ed8f8) - ((long) $rsp))
$9 = 216
(gdb) x /216xb $rsp
(以下は f メソッドの frame)
0x7ffff59ed820: 0x20    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # <- rsp
0x7ffff59ed828: 0x80    0x53    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed830: 0x88    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed838: 0xc8    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed840: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed848: 0x18    0x54    0x6f    0x19    0x07    0x00    0x00    0x00
0x7ffff59ed850: 0xa8    0x53    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed858: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
0x7ffff59ed860: 0x80    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # old sp2
0x7ffff59ed868: 0xd8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # old fp2 <- rbp
0x7ffff59ed870: 0xa7    0x9f    0x00    0xe1    0xff    0x7f    0x00    0x00 # return address 2
0x7ffff59ed878: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # local
(以下は main メソッドの frame)
0x7ffff59ed880: 0x04    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # param (<- old sp2)
0x7ffff59ed888: 0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # param
0x7ffff59ed890: 0x90    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed898: 0x4e    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8a0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed8a8: 0xc8    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8b0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed8b8: 0x18    0x54    0x6f    0x19    0x07    0x00    0x00    0x00
0x7ffff59ed8c0: 0x60    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8c8: 0x80    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed8d0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed8d8: 0x60    0xd9    0x9e    0xf5    0xff    0x7f    0x00    0x00 # (<- old fp2)
0x7ffff59ed8e0: 0xf3    0x09    0x00    0xe1    0xff    0x7f    0x00    0x00
0x7ffff59ed8e8: 0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed8f0: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

return

The following methods create native code for return. Apparently `_return_register_finalizer`, if the bytecode is , there is additional processing, but we ignore it here.

```
// in src/hotspot/cpu/x86/templateTable_x86.cpp

void TemplateTable::_return(TosState state) {
  ...
  // Narrow result if state is itos but result type is smaller.
  // Need to narrow in the return bytecode rather than in generate_return_entry
  // since compiled code callers expect the result to already be narrowed.
  if (state == itos) {
    __ narrow(rax);
  }
  __ remove_activation(state, rbcp);
  __ jmp(rbcp);
}
```

- Release frame for method f
- Revert rbp to what it was when interpreting method main (old fp2)
- Revert rsp to old sp2

And so on.

After that, return address 2 set by invokestatic is set in the rbcp (r13) register, so jmp to that.

```
# breakpoint at the return address 2 (after jmp)
(gdb) p $rbp
$203 = (void *) 0x7ffff59ed8d8
(gdb) p $rsp
$204 = (void *) 0x7ffff59ed880
(gdb) x /128xb $rsp
(main メソッドの frame)
0x7ffff59ed880: 0x04    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # <- rsp
0x7ffff59ed888: 0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed890: 0x90    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed898: 0x4e    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8a0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed8a8: 0xc8    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8b0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed8b8: 0x18    0x54    0x6f    0x19    0x07    0x00    0x00    0x00
0x7ffff59ed8c0: 0x60    0x54    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed8c8: 0x80    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed8d0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00
```

```
0x7ffff59ed8d8: 0x60    0xd9    0x9e    0xf5    0xff    0x7f    0x00    0x00 # <- rbp
0x7ffff59ed8e0: 0xf3    0x09    0x00    0xe1    0xff    0x7f    0x00    0x00
0x7ffff59ed8e8: 0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed8f0: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff59ed8f8: 0x60    0x59    0x6f    0x19    0x07    0x00    0x00    0x00
```

In the processing within the return entry points after jmp, return the value from the frame to the register (eg bcp_registers, local_registers), and then restart the interpreter from the Java bytecode following the invokestatic in main with dispath_next.

Summary

```
invokestatic (Java byte code)
↓
normal entry
↓
Java byte codes in the invoked method
↓
return (Java byte code)
↓
return entry points
↓
continue to interpret of the caller (who executed invokestatic) method
```

It was that kind of feeling.