Efficient Interpretation of Java Bytecodes

Kasper Lund

Who am I?

- Kasper Lund, virtual machine addict
  - 2000-2002: CLDC HI (Sun Microsystems)
  - 2002-2006: Resilient (OOVM)
  - 2006-2009: V8 (Google)
- Primary areas of interests
  - Dynamic programming languages
  - Interpretation and dynamic code generation
  - Method dispatching

Where is interpretation used?

- Virtual machines with mixed-mode execution
  - Methods are interpreted until they have been determined to be hot spots
  - Efficient interpretation leads to faster startup and less dynamic compilation
- Virtual machines for embedded devices
  - Dynamic compilation is too slow
  - Dynamically generated code takes up too much space

Why is interpretation slow?

- Decoding bytecode operands
- Dispatching based on opcode
- Memory traffic due to lack of register allocation

Decoding operands

- Bytecodes are often parameterized with constants, indexes, or offsets
  - bipush: 0x10 <constant>
  - iload: 0x15 <local index>
  - getfield: 0xb4 <constant $index_1$> <constant $index_2$>
  - if_acmpeq: 0xa5 <branch $offset_1$> <branch $offset_2$>
- Interpreter has to load the operands from the bytecode stream while executing the bytecodes
  - Byte-ordering: Big endian vs little endian
  - Have to deal with unaligned memory access

Decoding operands (cont.)

```
if_acmpeq:
  movsxb eax, [esi + 1]
  shl eax, 8
  movzxb ebx, [esi + 2]
  or eax, ebx
  add esi, eax
  ...
```

```
aload_<n>:
  push [edi + <n> * 4]
  ...
```

Dynamic bytecode customization

- Some bytecodes are often executed with specific operand value ranges
  - Check the operands on first bytecode execution
  - Rewrite the opcode to a customized opcode if the operands match the expected range

```
if_acmpeq_short:
  movsxb ebx, [esi + 2]
  add esi, ebx
  ...
```

Dispatching based on opcode

```
...
while (true) {
  switch (*bcp) {
    case op0: {
      ...
      bcp += 2;
      break;
    }

    case op1: {
      ...
      bcp += 1;
      break;
    }
  }
}
```

```
  ...
dispatch:
  movzxb ebx, [esi]
  jmp [ebx * 4 + table]

op0:
  ...
  add esi, 2
  jmp dispatch

op1:
  ...
  inc esi
  jmp dispatch
```

Indirect threaded interpretation

```
...
static const void* table =
    { &&op0, &&op1, ... };
goto *table[*bcp];

op0:
...
bcp += 2;
goto *table[*bcp];

op1:
...
bcp += 1;
goto *table[*bcp];
```

```
...
movzxb ebx, [esi]
jmp [ebx * 4 + table]

op0:
...
movzxb ebx, [esi + 2]
add esi, 2
jmp [ebx * 4 + table]

op1:
...
movzxb ebx, [esi + 1]
inc esi
jmp [ebx * 4 + table]
```

Loop examples (one is 34% slower)

```
start:
   iload_1
   iconst_1
   iadd
   istore_1
   iinc 3 1
   iload_3
   bipush 100
   if_icmple start
```

```
start:
   iinc 1 1
   iinc 3 1
   iload_3
   bipush 100
   if_icmple start
```

Results from study on Smalltalk

- We measured dispatch overhead by doubling the dispatch cost by going through an extra dispatch table
  - $T = B + D$ and $T_2 = B + 2 \times D$
  - $B = 2 \times T - T_2$
- Non-threaded interpreters spend **65-85%** of the total execution time dispatching between bytecodes
- Threading the interpreter reduces this fraction slightly to **56-78%** - cutting **21-32%** off the running times

Super bytecodes

- Combine multiple bytecodes into a single super bytecode
  - Look for common pairs in dynamic bytecode traces
  - Iteratively introduce new pair bytecodes allowing both bytecodes to be pair bytecodes themselves
- Reduces the amount of dispatching by executing less bytecodes (each doing more work)

Super bytecodes (cont.)

- Replacing bytecode sequences with the corresponding super bytecode is somewhat tricky
  - Must make sure nobody jumps into the sequence
  - Branch offsets change
- Solution: Leave the bytecodes there and just change the opcode of the first bytecode in the sequence
  - The original bytecodes (including opcodes) are left as the operands of the new super bytecode

aload_0 ; getfield #3	aload_0_getfield #3
0x2a 0xb4 0x00 0x03	**0xcb** 0xb4 0x00 0x03

Top-of-stack caching

```
iload_0
   push local_0
iload_1
   push local_1
iadd
   t_1 = pop
   t_0 = pop
   push t_0 + t_1
istore_0
   pop local_0
```

```
iload_0
   tos = local_0
iload_1
   push tos
   tos = local_1
iadd
   t_0 = pop
   tos = t_0 + tos
istore_0
   local_0 = tos
```

Multiple interpreter states

- Is the top-of-stack cache full or empty?
  - Keep state in separate variable or register
  - Check state before operating on **tos** register
    - Using tos register may require popping (empty)
    - Updating tos register may require pushing (full)
- Example on previous slide didn't take this into account

```
iload_0:
  if full: push tos
  tos = local_0
```

```
istore_0:
  if empty: pop tos
  local_0 = tos
```

Multiple interpreter states (cont.)

- Use multiple dispatch tables and have multiple version of the bytecodes depending on state
- Faster than explicitly checking state but blows up the size of the interpreter

```
iload_0_empty:
  tos = local_0
  jmp through full dispatch table

iload_0_full:
  push tos
  tos = local_0
  jmp through full dispatch table
```

```
istore_0_empty:
  pop tos
  local_0 = tos
  jmp through empty dispatch table

istore_0_full:
  local_0 = tos
  jmp through empty dispatch table
```

Top-of-stack caching (cont.)

- Can we stay in the full state (and only have one state)?
  - Requires popping tos when leaving in empty state
  - What happens on method entry when stack is empty?
- Conclusions: Modern hardware architectures do a pretty good job at optimizing access to the stack
  - Speed-up due to simple tos-caching is less than **5%**
  - Some micro-benchmarks show greater improvements
  - Make sure to measure if it's really worthwhile

Loop examples (cont.)

```
start:
   iinc 1 1
   iinc 3 1
   iload_3
   bipush 100
   if_icmple start
```

```
start:
   iinc 1 1
   iload_3
   iinc 3 1
   bipush 100
   if_icmple start
```

References

The Structure and Performance of Efficient Interpreters, M. Anton Ertl and David Gregg, 2003

Threaded Code, James R. Bell, Comm. of ACM, 1973

Revolutionizing Embedded Software, Kasper Lund and Jakob Andersen, Master's Thesis at University of Aarhus, 2003

Questions?