

Byte codes and interpreters

*Design of Virtual Machines for
Object-Oriented Languages*
Fall 2008

Outline of today's lecture

- Byte codes and interpreters
 - Byte code sets and encoding
 - Byte code dispatching
 - Optimizations
- Discuss exercise from last week
- Exercise for next week



Chromium: New open-source browser

- First beta release on September 2, 2008
- Uses WebKit as its rendering engine
 - Fast rendering
 - Relatively simple code base
 - Also used by Safari
- Contains a brand new JavaScript engine: V8
 - Replaces JavaScriptCore (SquirrelFish)
 - Improves performance of pure JavaScript code
 - Goal: Raise the performance bar for JavaScript

V8: Google's JavaScript engine

- Developed from scratch primarily in Aarhus
- Optimized for property access and function calls
 - Compiles JavaScript to native code
 - Uses behind-the-scenes maps to enable inline caching
 - Employs a precise, generational garbage collector
- Open-source under BSD license
 - <http://code.google.com/p/v8>
 - `svn co` <http://v8.googlecode.com/svn/trunk>

V8: Google's JavaScript engine (2)

- Reasonably fast on benchmarks
 - Rotating shuttle
 - Benchmark suite
- Get more technical details about V8 next week!

Let's get back to the lecture

Byte code sets

- Instruction set of the virtual machine
 - Maybe an internal representation only
 - Serialization format - Java class files, Python .pyc files
- Instructions usually have
 - Opcode which encodes the operation
 - Zero or more arguments that parameterize the operation
- CISC / RISC
 - Java virtual machines: 190+ opcodes (I think)
 - SELF: 7 opcodes

Java Bytecode

- Many, many specialized opcodes
- Encodes primitive type information
 - `aload_0` - load an object reference from local 0
 - `iret` - returns int from top of expression stack
- Operates on an expression stack
 - Result of evaluating an expression is pushed to the stack
 - Data operands are implicitly taken from top of stack

Java Bytecode example

```
int fib(int n) {  
    if (n < 2) return n;  
    return fib(n - 1) + fib(n - 2);  
}
```

```
0:  iload_1  
1:  iconst_2  
2:  if_icmpge 7  
5:  iload_1  
6:  ireturn  
7:  aload_0  
8:  iload_1  
9:  iconst_1  
10: isub  
11: invokevirtual fib:(I)I  
14: aload_0  
15: iload_1  
16: iconst_2  
17: isub  
18: invokevirtual fib:(I)I  
21: iadd  
22: ireturn
```

SELF opcodes

- Only seven different opcodes
 - SELF
 - LITERAL <value index>
 - SEND <message name index>
 - SELF SEND <message name index>
 - DELEGATEE <parent name index>
 - NON-LOCAL RETURN
 - INDEX-EXTENSION <index extension>
- Uniformly encoded in bytes
 - 3 bits for opcode and 5 bits for argument

SELF opcode example

x print.
' , ' print.
y print.

SELF SEND #0 (x)
SEND #1 (print)
LITERAL #2 (,)
SEND #1 (print)
SELF SEND #3 (y)
SEND #1 (print)

Compact encoding of arguments

- Java: Multiple variants of the same bytecode
 - `iload_n`: argument encoded in opcode ($n < 4$)
 - `0x1a | n`
 - `iload`: unsigned one byte argument
 - `0x15 <ub>`
- Java: Wide prefixes
 - wide `iload`: unsigned two byte argument
 - `0xc4 0x15 <ub> <ub>`
- SELF: Uniform encoding with extension prefixes
 - LITERAL #37 : INDEX-EXTENSION #1; LITERAL #5

Decoding arguments: Java wide

- Cannot look at previous opcode
 - Variable length encoding
 - Don't know if the last byte is an opcode or an argument
- Must deal with wide bytecode when we execute it
 - Jump through (sparse) separate dispatch table
 - Put wide dispatch table entries before non-wide variants

Decoding arguments: SELF extensions

- Possible to look back due to uniform encoding
 - Expensive because we need to check for each opcode
 - Have to deal with the start of a bytecode sequence
- Prefetch the arguments before dispatching
 - Read the argument bits into a hard-coded register
 - Let the INDEX EXTENSION opcode shift the current argument register and add the argument bits
 - No need for different variants of the code for interpreting a single opcode

Stack-based or register-based?

- Java and SELF bytecodes operate a lot on the stack
 - Easy to generate using recursion
 - Too much memory traffic
- Register-based alternatives
 - Lua, SquirrelFish, Parrot
 - Byte codes must explicitly encode their data operands
 - Spend more time decoding
 - Less compact representation
 - May be a bit faster to interpret

Byte code dispatching

- Dispatching is transferring control between the different parts of the interpreter routine that handles the specific opcodes
- Measurements on interpreted Smalltalk shows
 - A fast interpreter may spend 65-85% time dispatching
 - Indirect threading reduces this to 56-78%
 - ... but it's still a lot ...

```
movzb ebx, [edi + <n>]
add edi, <n>
jmp [ebx * 4 + <dispatch table offset>]
```

Threaded interpretation

- Widely used in its indirect table-based form
 - Allows compact opcode encoding
 - Allows changing the interpreter behavior dynamically
 - Break out of long running loops without explicit checks
 - Only check for break points when actively debugging
 - Reasonably efficient on most platforms

```
void* table[] = { &&..., &&... };  
goto *table[opcode];
```

Exercise for week 36

- Great work - keep it up!
- Notes about hand-ins that consist mostly of code
 - Make sure it's documented
 - Please put in a note of anything unexpected
 - Make it as easy to reproduce as possible
 - Include relevant information about OS, processor, compiler, compilation flags, etc.
 - Only include relevant code
 - Nobody wants to read through the generated machine code for ReadFile - not even me
 - Look at the code before you hand it in
 - Did the code get optimized as you expected?

Why is threading faster?

- Branch prediction is the number one reason
 - 1111 - easy to predict
 - 11221122 - two branch sites with two possible targets
 - 1234 - should be easy to predict for threaded version
 - 13142134 - three branch sites with two possible targets
- Unfortunately branch prediction varies a lot between different version of the CPUs

Pentium 4 performance issues?

- Possible causes
 - Long pipeline leads to expensive mispredictions
 - Shouldn't this hurt the switch-based one even more?
- How about code caches?
 - Pentium 4 and Pentium M very different code caches
 - Switch code takes up ~64 bytes
 - Threaded code takes up ~100 bytes
- Maybe the switch-based version is fast because it's small?

Optimization techniques

- Write or generate the interpreter machine code
 - Compilers often have a hard time optimizing interpreters
 - Allows using the native execution stack
- Cache the top-of-stack in a register
 - Some values may only fit in certain registers
 - May need multiple interpreter states / dispatch tables
- Reduce dispatching overhead by combining byte codes
 - Static rewriting - also suggested by group 15 (in colors)
 - Dynamic rewriting

Exercises for week 37

- Read articles
 - An Efficient Implementation of SELF (...)
 - Uniprocessor Garbage Collection Techniques
- Hand in written report (2-3 pages) with a discussion of the pros and cons of using maps in a virtual machine for an object-oriented language without classes
 - How does maps impact memory consumption?
 - Does it enable certain optimizations? Disable others?
 - How would you add it to an existing implementation?
 - Give examples of operations you need to change
 - How complex is it to implement?