Compiling for Java

Jeremy Singer

classic compilation model

- ***ahead-of-time*** compilation
- programmer writes high-level source code
  - e.g. C or C++
- programmer compiles to executable binary code
- programmer distributes executable binary code
- user installs and runs executable binary code

does AOT compile time matter?

- not really, since the user never sees (or knows) how long compilation takes
- because the programmer incurs the cost of compile time

- Long compile times are bad for edit/compile/debug cycle
- Otherwise, not a serious problem

alternative compilation model

- ***just-in-time*** compilation

- programmer writes high-level source code
  - e.g. C# or Java
- programmer compiles to platform-neutral bytecode
- programmer distributes bytecode

- user installs and runs bytecode
- system converts bytecode into native machine code

does JIT compile time matter?

- Yes, since compile time is now part of the application execution time, which the user experiences

- Compilation mustn't take too long!

What are the benefits of JIT?

- programmer writes code once, and it should run on any machine
  - 'write once, run anywhere'
  - a single distributable version of the code
- JIT compiler can generate highly specialized code for the target processor/platform

Key message

- A JIT compiler must balance compilation ***overhead*** with execution ***speedup***.

- Only optimize ***hot*** methods

Hot methods

- In general, programs spend **90%** of their time executing only **10%** of their code
- hot code might be inner loops of data processing methods, etc
- cold code might be exception handling methods, etc
- The ‘Pareto Principle’

Focus compilation effort

- identify hot code
- concentrate the expensive compilation on this code, for maximum benefit

Code generators

- for a modern JIT compiler, there will be several code generators:

- interpreter – directly execute the JVM bytecodes
- fast compiler – generate native code without optimizations
- opt compiler – generate native code with optimizations

Example table of costs / times

	code generation time (COST)	execution time (BENEFIT)
interpreter	0	t
fast compiler	$k*n$	$0.3*t$
opt compiler	$5k*n$	$0.1*t$

Code generation time is for single method containing n bytecode instructions

Assume this method executes with average time t in interpreter

k measures compilation cost– in seconds per bytecode instruction

Notice that …

- execution time ***decreases*** with compilation level
- But code generation time ***increases*** with compilation level

Balancing costs and benefits

- Initially, interpret all code
  - (quick to generate code, slow to execute it)
- Identify hot methods
  - (by profiling)
- for hot methods, determine whether

$$time\_to\_recompile + improved\_execution\_time < execution\_time\_without\_recompile$$

Example

- Suppose method `m()` has 20 bytecode instructions and takes 75 time units to execute

- Suppose compilation cost is 10 time units per instruction for fast compiler, and 50 time units per instruction for opt compiler.

- time to compile `m()` with fast compiler:
  - 20 * 10 = 200 time units
- with the opt compiler:
  - 20 * 50 = 1000 time units

But how do we know whether it's worth it?

- Assume the amount of time spent executing `m()` so far is equal to the amount of time that will be spent executing `m()` in future
- Simply, we always assume we are *half-way* through the program execution
- So – if we have executed `m()` 5 times so far, assume we will execute it another 5 times

recall …

- for hot methods, determine whether

time_to_recompile + *improved_execution_time*

<

execution_time_without_recompile

Now we can see whether the compilation effort is justified

- time to recompile `m()` with fast compiler is:
  - 200 time units (see earlier slide)
- from table previously, method execution will be 30% of original time – time to execute recompiled `m()` is:
  0.3 * 75 * 5 = 113 time units

- time to execute `m()` if we do not recompile is:
  mean execution time for `m()` * expected number of invocations of `m()`:
  75 * 5 = 375 time units

- Since (200+113) < 375 – the recompilation effort is beneficial in this case!

Implementation Issues

- How do we identify hot methods?
  - counters, stack sampling, profiling
- Do we only recompile to the 'next' level of optimization, or do we consider all levels as possibilities?
- Should we ever reverse an optimization if it is not beneficial?

Exercises

- Please download the Java Compilation worksheet from the http://wolearn.org site and go through the exercises *before* class.

- Also think about questions to ask on this topic.