8 JIT Compilation

JIT - "Just in Time"
- possible for any language
- made popular by Java

- compile controlled by engine
  "virtual machine" (JVM)

- write program once - run everywhere
- produce intermediate code
  JIT compiles intermediate code
  executes on all platforms
  90s - many architectures
  00s - one arch, left x85
  10s - smartphones, tablets -
  diversity

write once, debug everywhere
- new optimization opportunities
- use runtime information
  "rear view mirror"

**JIT compile**

- Java: compiler
  - full program translation
  - circle all libraries!

```
modular compilation is a must
```

```
Java  -->  intermediate code (byte code)
          -->  native code
```

"front end compiler"

- JIT compile
  - execute on host
    (all libraries etc.)

ahead of time
Compilation is another service of the runtime system

- VM is compiler service provider
- memory management
- garbage collection
- thread management
- profiling infrastructure
- include interpreter for intermediate code (not mandatory, but often done)
Setup: Compiler in cooperation with interpreter

Issue: mixed mode execution

Intermediate language: does not know about:

- (byte code)
- handle local resources
- registers not known

foo, recursive function
foo -> bar -> foo

+ many exceptions

foo candidate

for (int)

compiler

// optimized main of foo
At run time

Stack

foo
bar
foo

foo

bar

foo (optimized/compiled)

bar

foo (compiled code/register)

interface to unp

continue with

interpreted code
Inputs ... slow code
100x slower than compiled code

\[ \Rightarrow \]
JIT compile should be invoked if execution time matters

- Code block(s), method(s) are executed frequently

\[ \Rightarrow \] compilation takes time must be recovered by program speedup
Variant:

"shift" compile once

- start with baseline compile
- when useful / winning strategy?

fast compile to have a compilation time of the same order of magnitude as the time taken to load program

- transfer from local storage
- transfer via the network

Compile cannot include expensive analysis

faster loading makes life more difficult
Fast compiler must be almost linear

(Intel adopted this approach)

pass — global register allocation — code generation — code emission

Linear time processing of the byte code

- simple CFG
- keep track of stack depth
- identify points where GC can be invoked

(to identify variables that contain reference,
(prohibit / delay) GC until safe point is reached)
propose identifier candidate for register allocation
* immediate values are good candidate
  no need to spill value

code generator
  * common subexpression elimination based on bitvector
    reasonably fast - (128 entries are handled)
8.1 Partial compilation

Compile code from different methods together
(To be the same optimization level)

- O-O programs contain small methods
  Scope of optimization limited
- O-O programs contain exception handling
  Code, less-frequently executed code
  Optimizing compiler is expensive — only to be used on frequently executed parts.
Example (Java, Sketch)

```java
void read_data(String fn) {
    int a;
    int b;
    int acc;
    try {
        FileInputStream sif = new FileInputStream(fn);
        n = sif.length;
        buffer = new byte[n];
        while (b = sif.read(buffer)) {
            acc += acc + b;
        }
        sif.close();
        if (...) { // handle errors...
            catch (IOException) { // handle IO problems
        }
    }
```
Inlining of read not a good idea
contains try-catch

That part of the program contains code from "read_db" as well as code from "read".

Abandon method as compilation/optimization unit
- maintain exec. counts for basic blocks
- combine blocks from multiple methods into one compilation unit.
read - db

if (...) {
  buffer = ...
  while (...) {
    setup read
    Call
    read
    catch
  }
}
Book Keeping is more complicated than in "conventional" trace scheduling... doable