7. Just In Time Compilation

Prof. O. Nierstrasz
Jan Kurs

UNIVERSITÄT BERN

Roadmap

- What is Just-In-Time Compilation (JIT)?
- History of JIT
- JIT Overhead
- Optimization Techniques in JIT

Roadmap

- **What is Just-In-Time Compilation (JIT)?**
- History of JIT
- JIT Overhead
- Optimization Techniques in JIT

Compilation vs Interpretation

Compilation

Pros

- Programs run faster

Cons

- Compilation overhead
- Programs are typically bigger
- Programs are not portable
- No run-time information

Interpretation

Pros

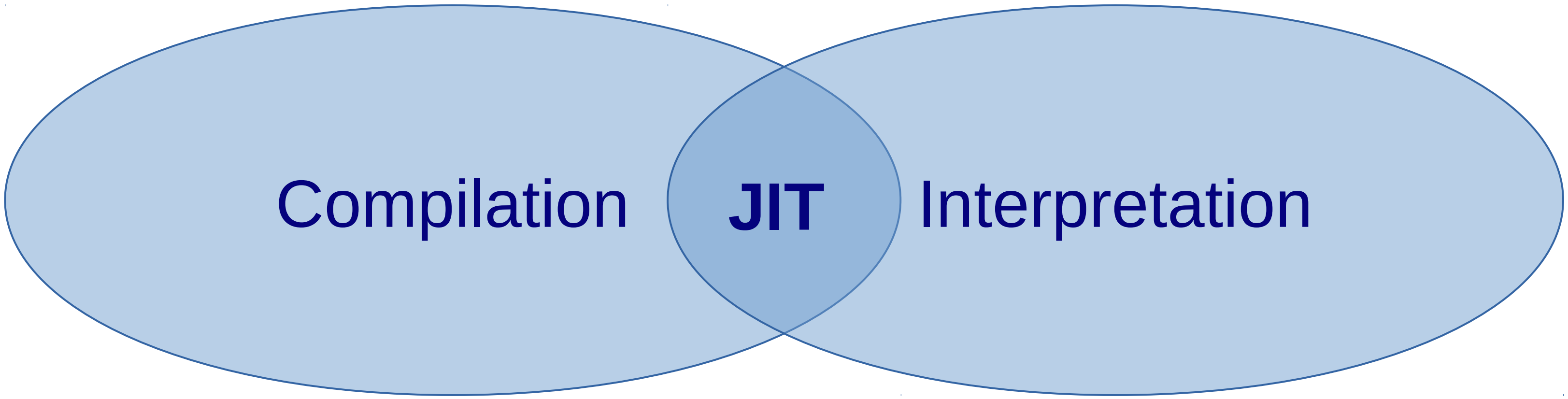
- Programs are typically smaller
- Programs tend to be more portable
- Access to run-time information

Cons

- Programs run slower

What is Just-In-Time Compilation?

Dynamic Translation: Compilation done during execution of a program – *at run time* – rather than prior to execution



What is Just-In-Time Compilation?

Is Just-In-Time

- dead code elimination during program execution?
- generation of native code during program execution?
- static analysis and subsequent optimization?
- compile-time generation of native code?
- Is JIT compile-time optimization based on previous program execution?

Why Just-In-Time Compilation?

Improve time and space efficiency of programs utilizing:

- portable and space-efficient byte-code
- run-time information → feedback directed optimizations
- speculative optimization

Why Just-In-Time Compilation?

```
kursjan@ludvik ~/doc/teaching/JIT/ws/Foo/bin $ time java -Xint  Qwark 100000000

real	0m9.499s
user	0m9.129s
sys	0m0.196s
kursjan@ludvik ~/doc/teaching/JIT/ws/Foo/bin $ time java  Qwark 100000000

real	0m0.143s
user	0m0.024s
sys	0m0.112s
kursjan@ludvik ~/doc/teaching/JIT/ws/Foo/bin $
```

Roadmap

- What is Just-In-Time Compilation (JIT)?
- **History of JIT**
- JIT Overhead
- Optimization Techniques in JIT

History of Just-In-time

First Just-In-Time

- 1960
- McCarthy's LISP paper about dynamic compilation

Fortran

- 1974
- Optimization of “hot spots”

Smalltalk

- 1980 – 1984
- Bytecode to native code translation
- First modern VM

History of Just-In-time

Self

- 1986 – 1994
- New Advanced VM techniques

Java

- 1995 – present
- First VM with mainstream market penetration

Android RunTime (ART)

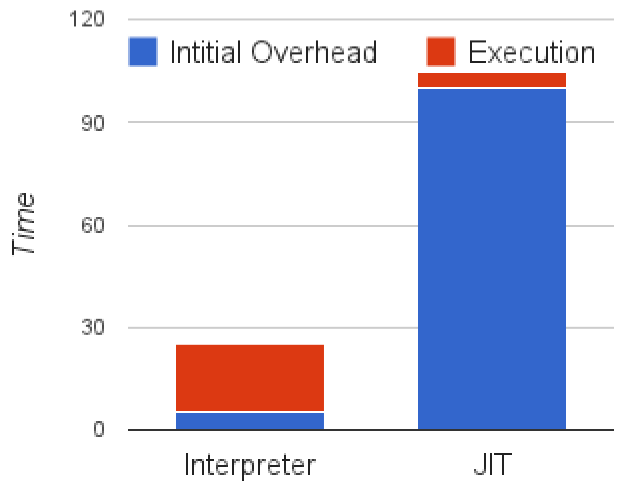
- 2014
- No JIT ;-)

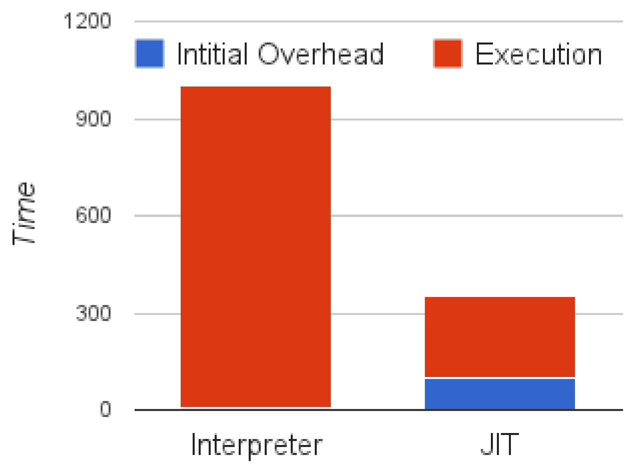
Roadmap

- What is Just-In-Time Compilation (JIT)?
- History of JIT
- **JIT Overhead**
- Optimization Techniques in JIT

Just-In-Time Overhead

JIT: 4x speedup, but 20x initial overhead





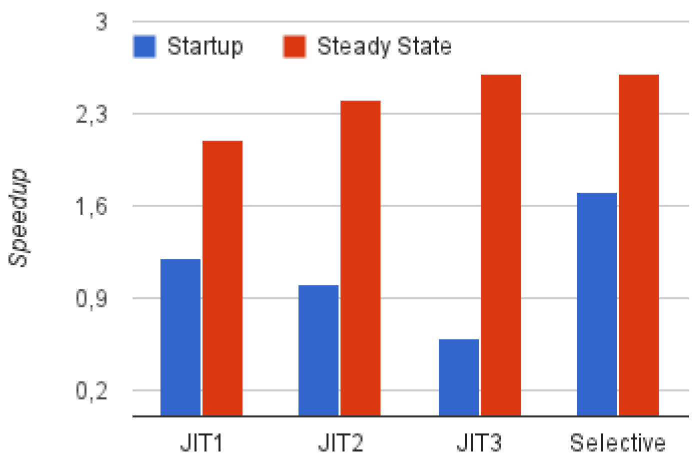
Matthew Arnold, Stephen Fink, David Grove, and Michael Hind, ACACES'06, 2006

Selective Optimization

- Start program in interpreted mode
- Find “hot spots”
- compile only hot spots

Selective Optimization

- **JIT1, JIT2 and JIT3**: the better startup, the worse steady state performance.
- **Selective optimization with JIT3:** reaches best startup and best steady state performance



Matthew Arnold, Stephen Fink, David Grove, and Michael Hind, ACACES'06, 2006

NB: Java Virtual Machine

- HotSpot
- server mode (-server)
  - aggressive and complex optimizations
  - slow startup
  - fast execution
- client mode (-client)
  - less optimizations
  - fast startup
  - slower execution

What To Optimize

- Method Counters
- Call Stack Sampling

What To Optimize: Method Counters

```
public void foo() {
    fooCounter++;
    if (fooCounter > threshold) {
        recompile();
    }
}
```

- Approximation of time spent in each method
- Popular
- Might have significant overhead

What To Optimize: Call Stack Sampling

- Call stack inspected in regular intervals as the program is running
- Approximation of time spent in each method
- Not deterministic

Roadmap

- What is Just-In-Time Compilation (JIT)?
- History of JIT
- JIT Overhead
- **Optimization Techniques in JIT**

Optimization Techniques

- Loop Unrolling
- Register Allocation
- Global Code Motion
- Machine Code Generation
- Inlining
- Code Positioning
- Multi-Versioning
- Dynamic Class Hierarchy Mutation

Standard Techniques Revised

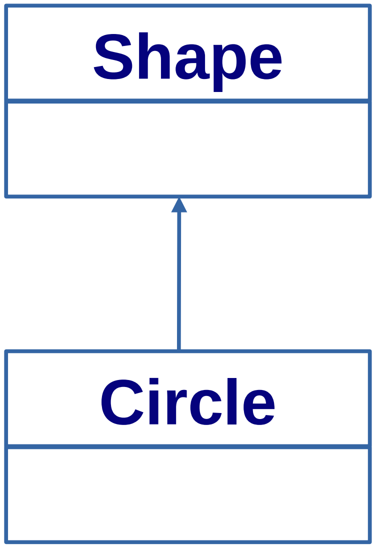
- Loop Unrolling
  — unroll "hot" loops only
- Register Allocation
  — assign register to "hot path" variables first
- Global Code Motion
  — move code from "hot" block
- Machine Code Generation
  — generate code for the particular architecture

Inlining (Pros & Cons)

- Pros
  - removes cost of a function call and return instruction
  - improves locality of code
  - once performed, additional optimizations can become possible
- Cons
  - may degrade performance (code size overflows cache)
  - increases code size

Speculative Inlining

```
for (Shape shape : shapes) {
    shape.computeArea();
}
```



- Inline `Circle.computeArea()`
- Monitor class hierarchy
- Recompile if `Shape` has more subclasses

On Stack Replacement (OSR)

Transfers execution from code A to code B even while code1 runs somewhere.

```
for (Shape shape : shapes) {
    area = ((Circle)shape).r() * pi^2;
}
```

Square appears in the `shapes`.

We cannot wait for loop to finish.

On Stack Replacement Applications

- Invalidation of speculative optimization
- De-optimization for debugging
- Runtime optimization of long-running activations

Multiversioning

- Multiple implementations of a code
- The best implementation is chosen at runtime

```
for (Shape shape : shapes) {
    area = shape.area();
}
```

heterogeneous

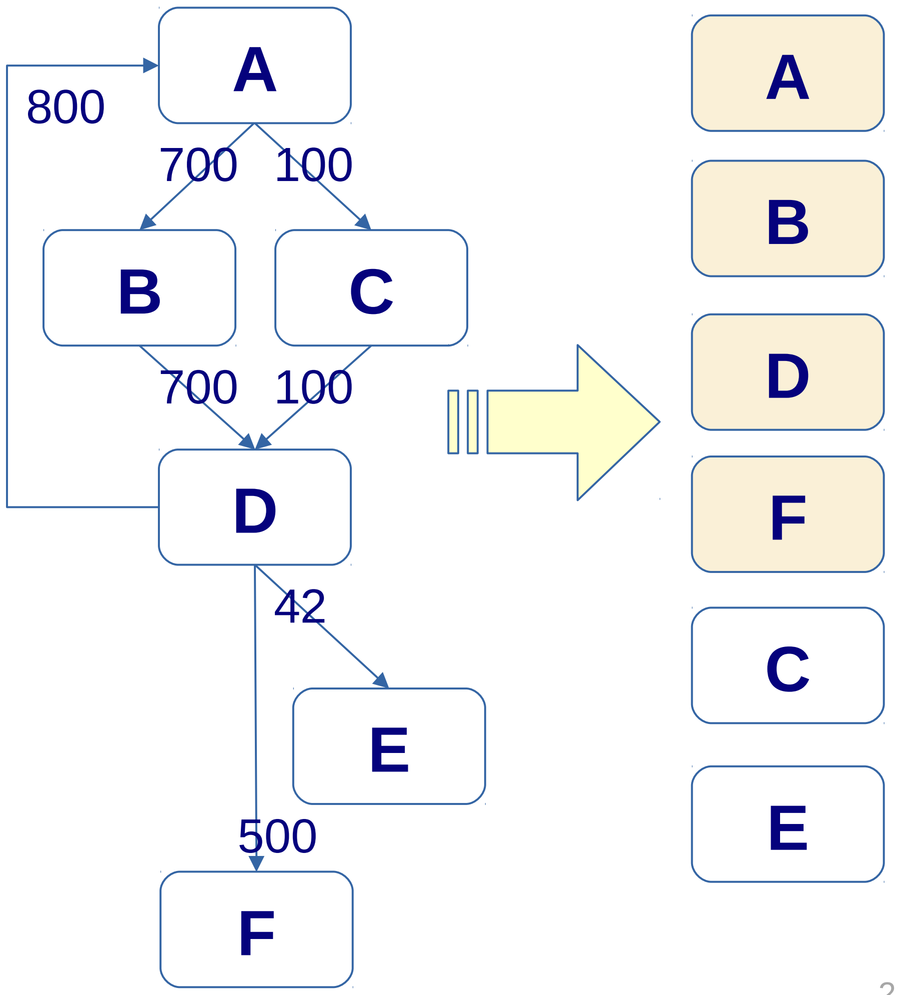
```
for (Shape shape : shapes) {
    area = shape.area();
}
```

```
for (Shape shape : shapes) {
    area = ((Circle)shape).r() * pi^2;
}
```

homogeneous

Code Positioning

- Linearizes the most common path
- Improves code locality
- Eliminates jumps
- Improves cache performance



Inline Caches (ILC)

- Improves performance by remembering the result of previous method lookup at the **call site**.

```
Object[] values =
    { 1, "a", 2, "b"};

values[0].toString();
values[1].toString();


for (val : values) {
    val.toString();
}
```

IC1

```
if (receiver.class == Integer)
    invoke #Integer.toString
else
    invokevirtual values[0] #toString
```

IC2

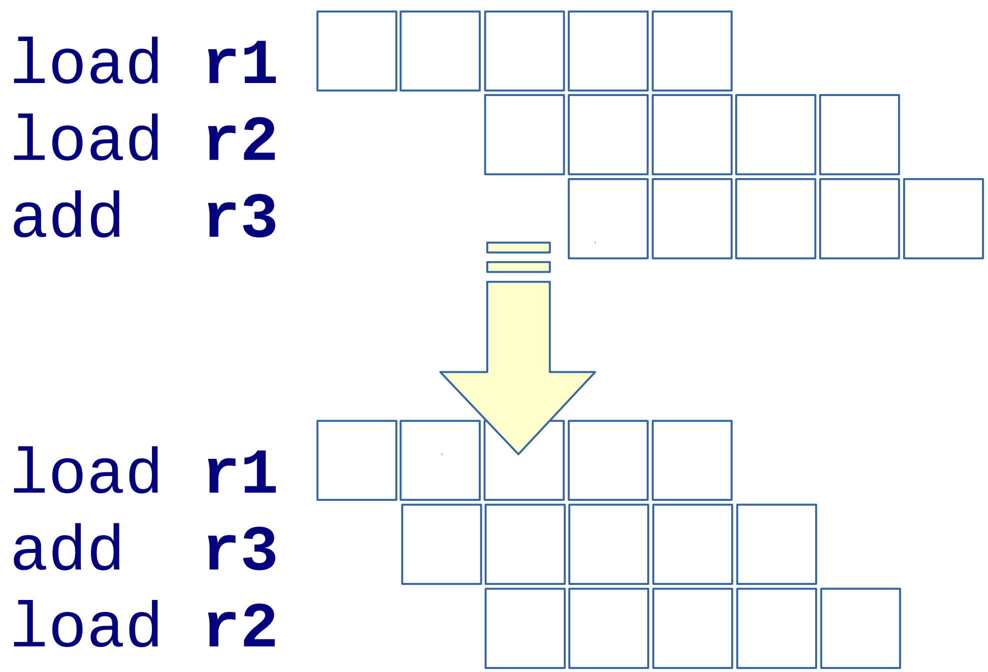
```
if (receiver.class == String)
    invoke #String.toString
else
    invokevirtual values[1] #toString
```

IC3

```
if (receiver.class == Integer)
    invoke #Integer.toString
else
    invokevirtual val #toString
```

Instruction Scheduling

- Improves Performance with instruction pipelines
- Heavily dependent on underlying architecture



What Should You Know!

- ✎ *What is and what is not Just-In-Time?*
- ✎ *What are advantages of JIT?*
- ✎ *What are drawbacks of JIT?*
- ✎ *What techniques can you use to reduce a JIT compilation overhead?*
- ✎ *What extra information does the JIT compiler have compared to static compiler?*
- ✎ *What is speculative inlining?*
- ✎ *What is code positioning?*
- ✎ *What is On Stack Replacement?*
- ✎ *What is Inline Cache?*

Can You Answer These Questions?

- ✎ *When would you prefer not to use a JIT compiler?*
- ✎ *Why can JIT compiler generate faster code than static compiler?*
- ✎ *How does code positioning improve performance?*
- ✎ *Why is OSR important for speculative optimizations?*
- ✎ *What happens if you dynamically load class in Java (from optimizations point of view)?*
- ✎ *What is is a time overhead of dynamic dispatch?*
- ✎ *What is the time overhead of dynamic dispatch with ILC?*

License