

Falcon

a new JIT compiler in Zing JVM

Artur Pilipenko
apilipenko@azul.com
[@arturpilipenko](#)

Ivan Krylov
ivan@azul.com
[@JohnWings](#)



Agenda

- What is Falcon? Why do we need a new compiler?
- Why did we decide to use LLVM?
- What does it take to make a Java JIT from LLVM?
- How does it fit with exiting technology, like ReadyNow?

Zing JVM

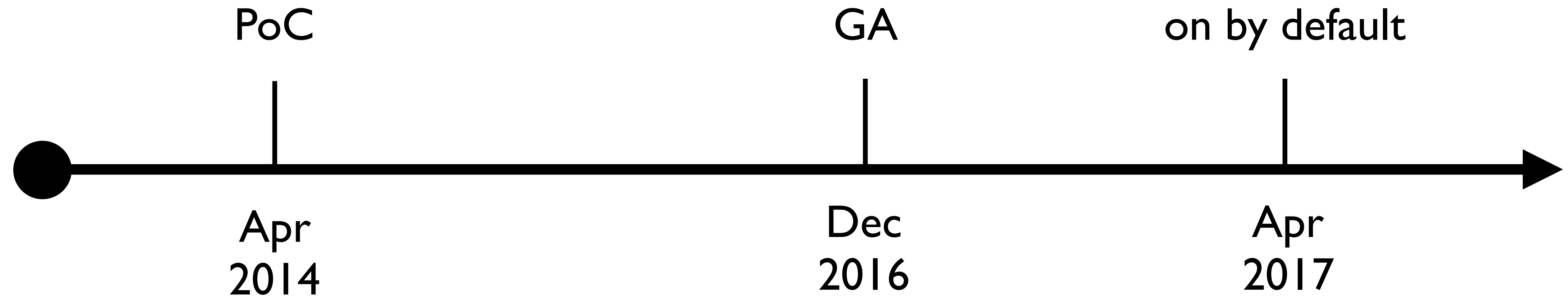
Zing: A better JVM

- Commercial server JVM
- Key features
 - C4 GC, ReadyNow!, **Falcon**

What is Falcon?

- Top tier JIT compiler in Zing JVM
- Replacement for C2 compiler
- Based on LLVM

Development Timeline



**Estimated resource investment
~20 man-years (team of 4-6 people)**

Why do we need a new compiler?

- Zing used to have C2 like OpenJDK
- C2 is aging poorly — difficult to maintain and evolve
- Looking for competitive advantage over competition

Alternatives

- Writing compiler from scratch? Too hard
- Open-source compilers
 - GCC, Open64, Graal, LLVM...

Alternatives

- Writing compiler from scratch? Too hard
- Open-source compilers short list
 - Graal vs LLVM

**“The LLVM Project
is a collection of
modular and
reusable compiler
and toolchain
technologies”**

– llvm.org



Where LLVM is used?

- C/C++/Objective C



- Swift



- Haskell



RubyMotion

- Rust



- ...

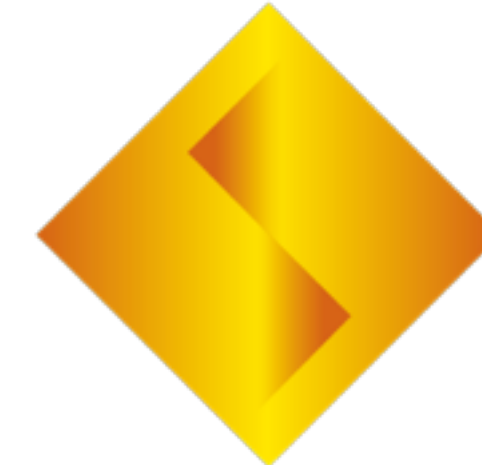


OpenCL

Who makes LLVM?



SONY



COMPUTER
ENTERTAINMENT®



NVIDIA®



More than 500 developers



LLVM

- Started in 2000
- Stable and mature
- Proven performance for C/C++ code

LLVM IR

General-purpose high-level assembler

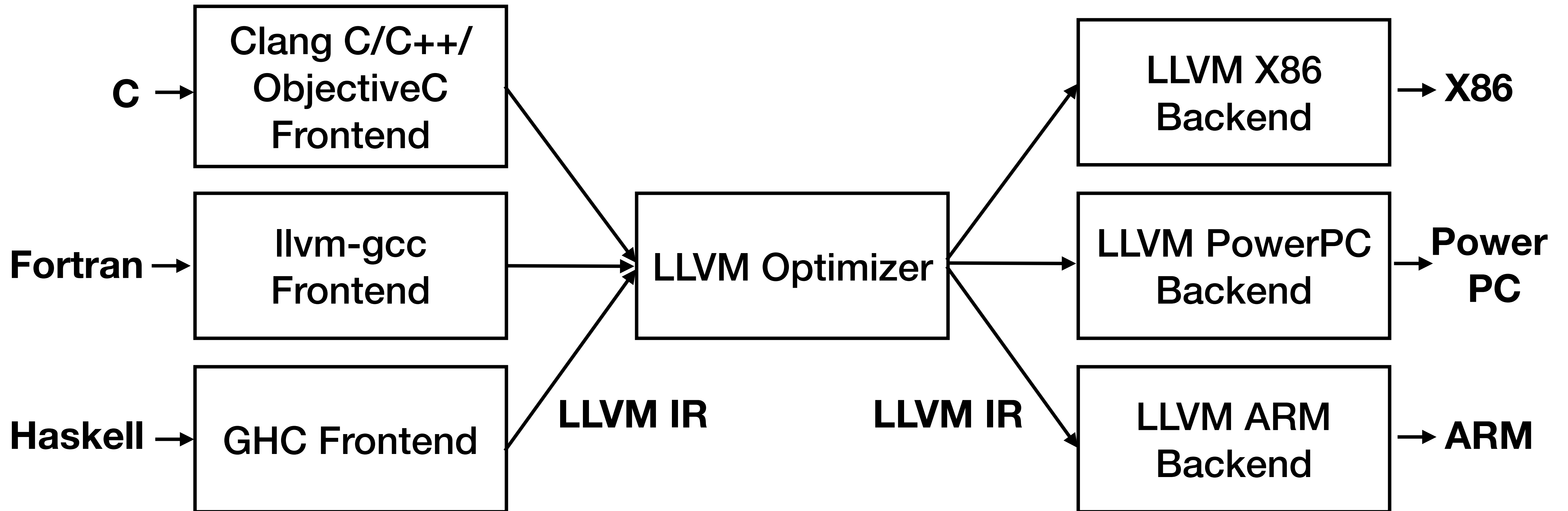
```
int mul_add(int x, int y, int z) {  
    return x * y + z;  
}
```

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {  
entry:  
    %tmp = mul i32 %x, %y  
    %tmp2 = add i32 %tmp, %z  
    ret i32 %tmp2  
}
```

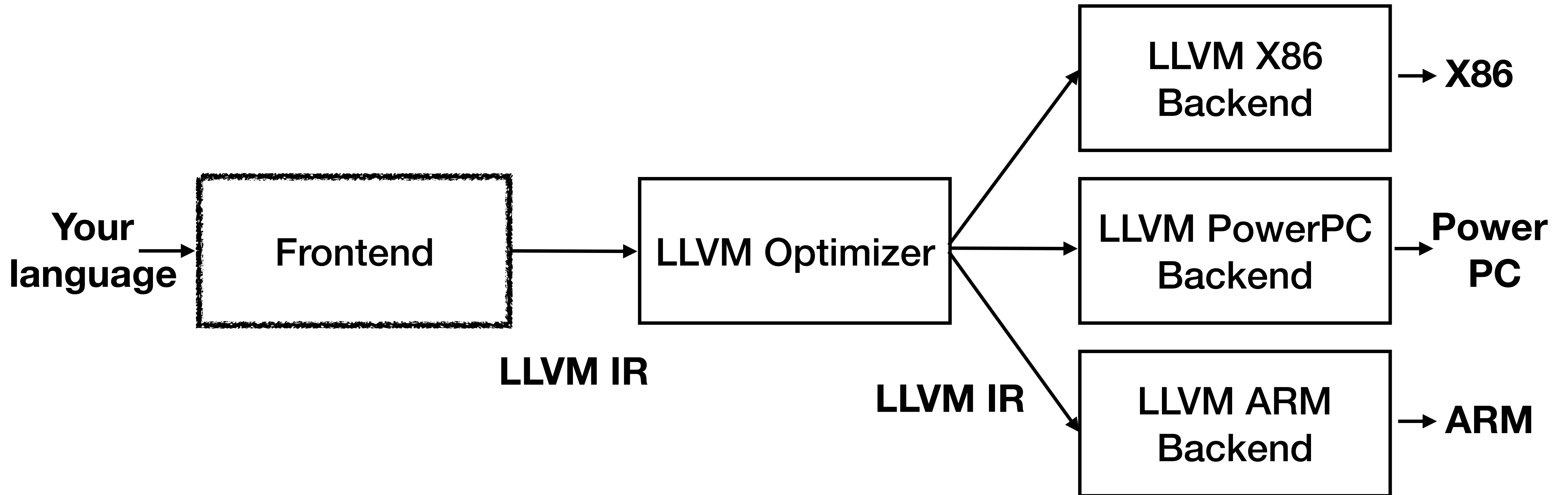
Infrastructure provides

- Analysis, transformations
 - LLVM IR $\Rightarrow$ LLVM IR
- Backends for all major CPU architectures
 - LLVM IR $\Rightarrow$ Native
- Tools

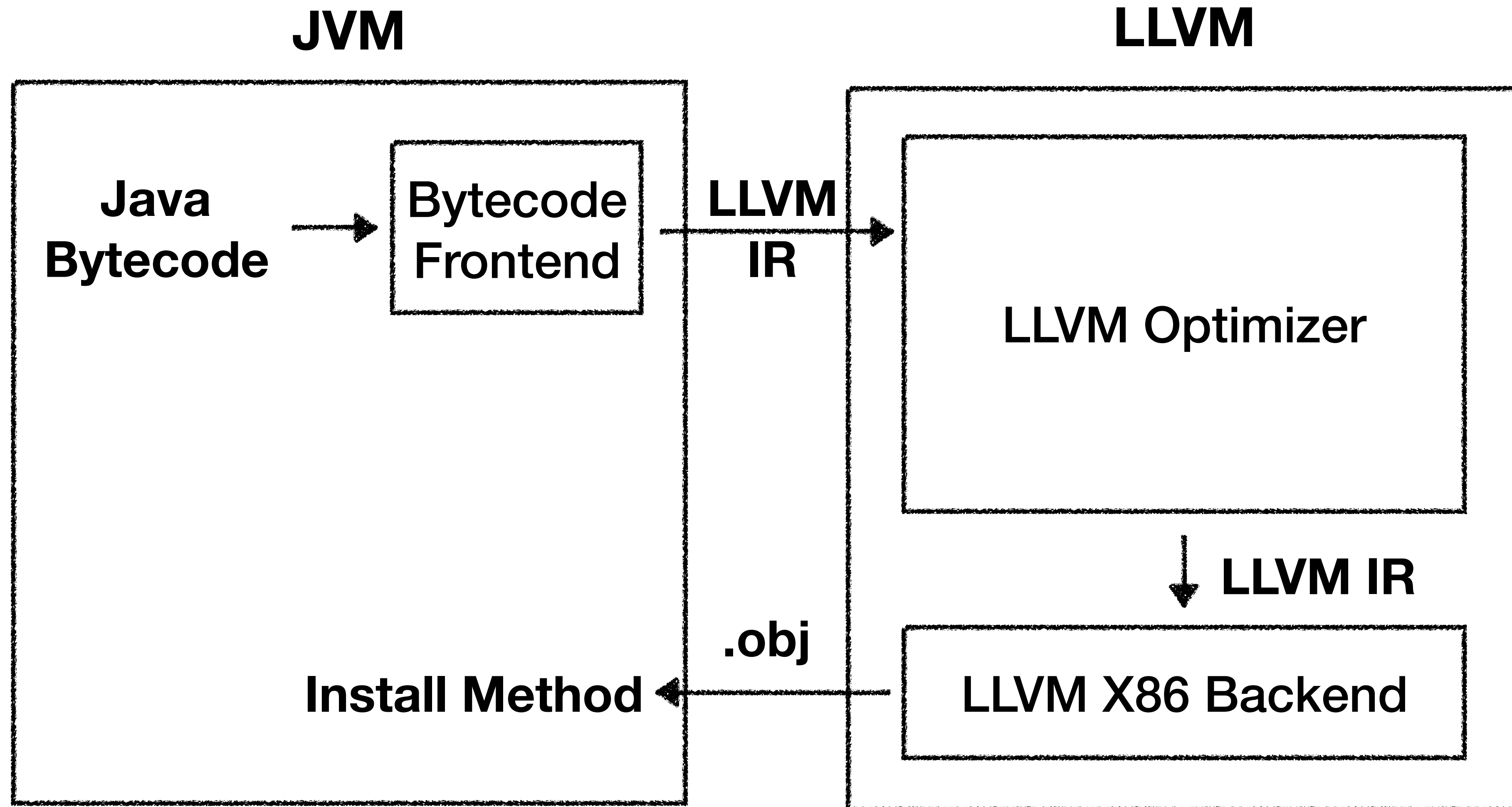
How to write a compiler?



How to write a compiler?



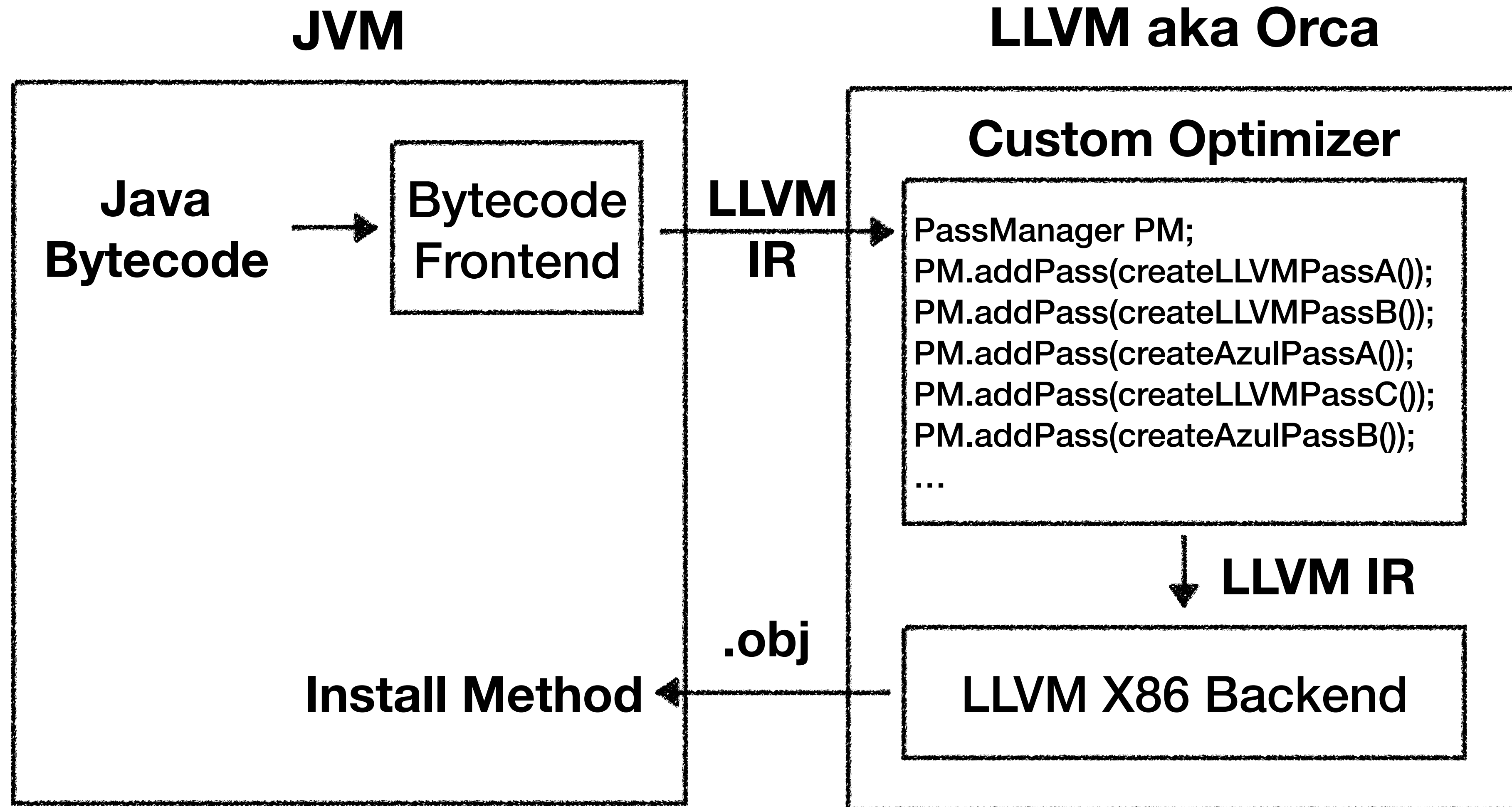
Falcon



Orca

- Our fork of LLVM
- LLVM-based Java JIT compiler
- Can be used in other VM
- Or outside of any VM

Falcon

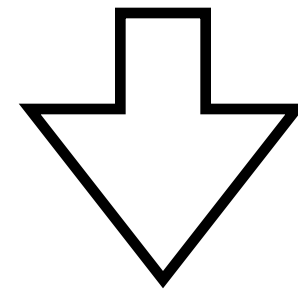


VM Callbacks

```
static final Object X = new Integer(42);  
bool foo() {  
    return X instanceof Number;  
}
```

VM Callbacks

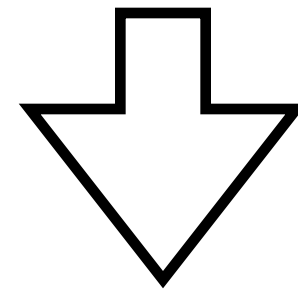
```
static final Object X = new Integer(42);  
bool foo() {  
    return X instanceof Number;  
}
```



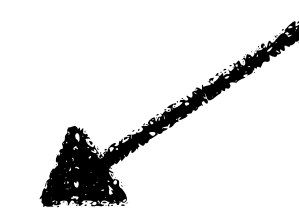
```
define i1 @foo() {  
    %X = load i8*, i8** <Xaddress>  
    %kid = call i32 @get_class(%X)  
    %res = call i1 @is_subtype_of(<Number>, %kid)  
    ret i1 %res  
}
```

VM Callbacks

```
static final Object X = new Integer(42);  
bool foo() {  
    return X instanceof Number;  
}
```



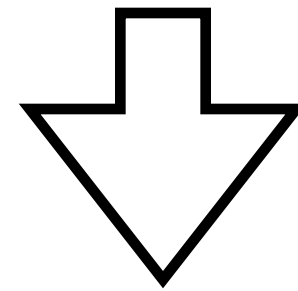
VM, what is at
<Xaddress>?



```
define i1 @foo() {  
    %X = load i8*, i8** <Xaddress>  
    %kid = call i32 @get_class(%X)  
    %res = call i1 @is_subtype_of(<Number>, %kid)  
    ret i1 %res  
}
```

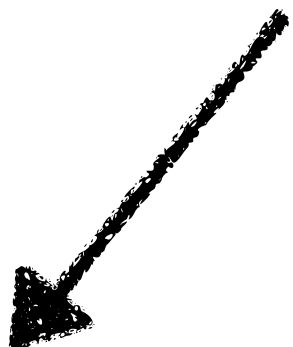
VM Callbacks

```
static final Object X = new Integer(42);  
bool foo() {  
    return X instanceof Number;  
}
```



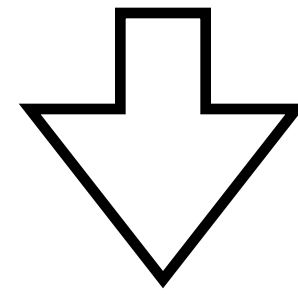
```
define i1 @foo() {  
    %X = load i8*, i8** <Xaddress>  
    %kid = call i32 @get_class(%X)  
    %res = call i1 @is_subtype_of(<Number>, %kid)  
    ret i1 %res  
}
```

**VM, what is the
type of this
object?**



VM Callbacks

```
static final Object X = new Integer(42);  
bool foo() {  
    return X instanceof Number;  
}
```

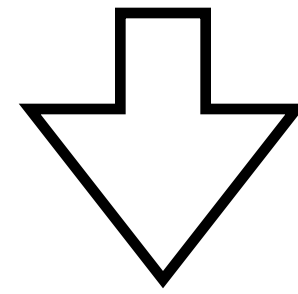


java.lang.Integer

```
define i1 @foo() {  
    %X = load i8*, i8** <Xaddress>  
    %kid = <Integer>  
    %res = call i1 @is_subtype_of(<Number>, %kid)  
    ret i1 %res  
}
```

VM Callbacks

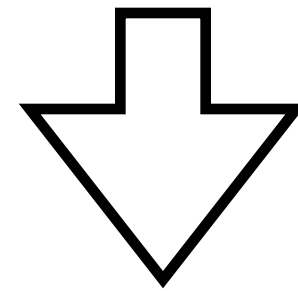
```
static final Object X = new Integer(42);  
bool foo() {  
    return X instanceof Number;  
}
```



```
define i1 @foo() {  
    %res = call i1  
    @is_subtype_of(<Number>, <Integer>)  
    ret i1 %res  
}
```

VM Callbacks

```
static final Object X = new Integer(42);  
bool foo() {  
    return X instanceof Number;  
}
```



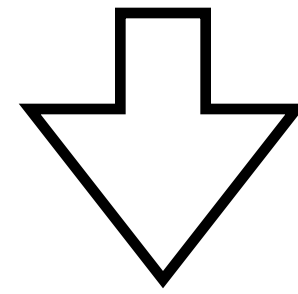
```
define i1 @foo() {  
    %res = call i1  
    @is_subtype_of(<Number>, <Integer>)  
    ret i1 %res  
}
```

**VM, is Integer a
subtype of
Number?**



VM Callbacks

```
static final Object X = new Integer(42);  
bool foo() {  
    return X instanceof Number;  
}
```

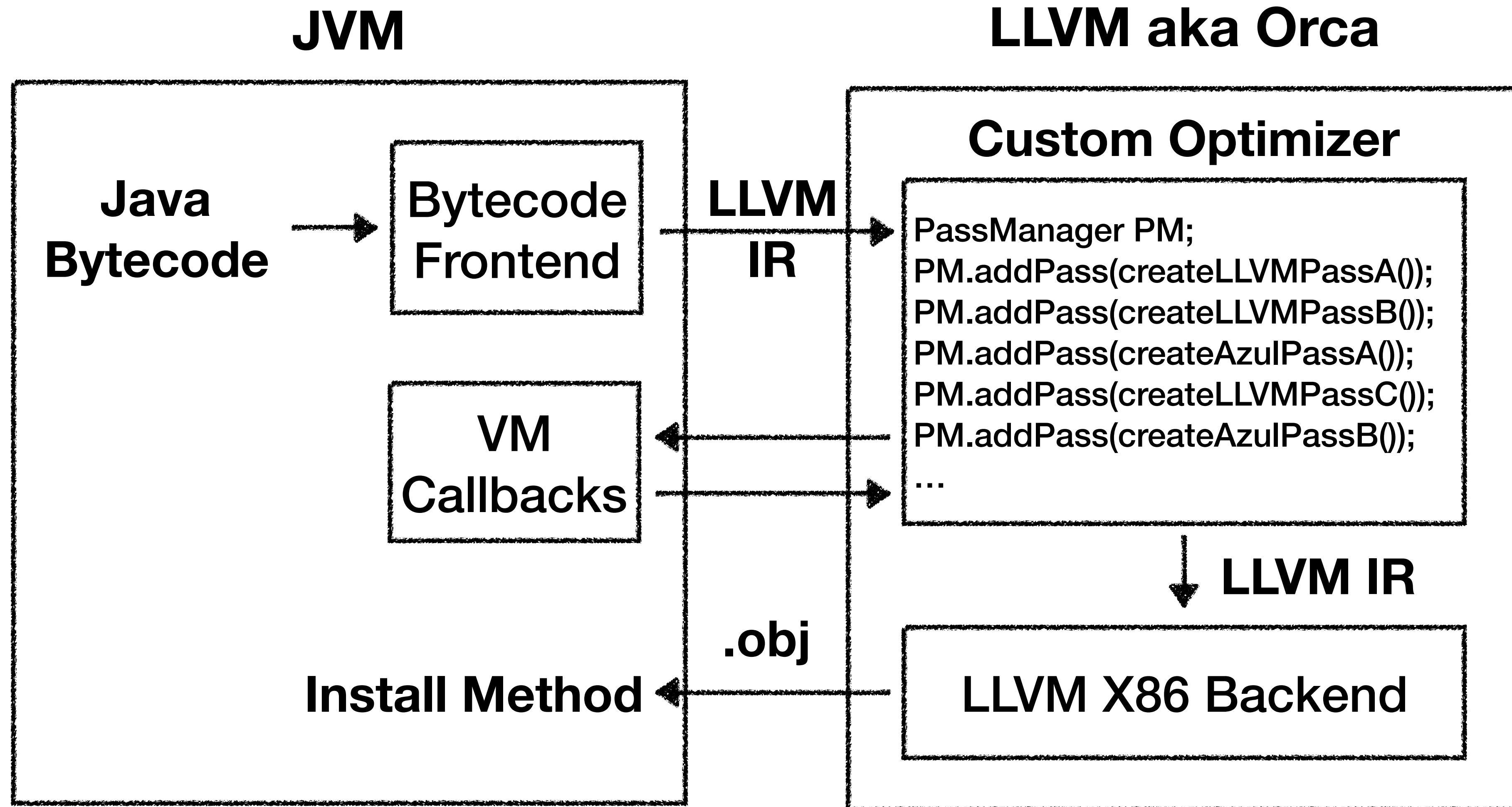


```
define i1 @foo() {  
    ret i1 true  
}
```

VM Callbacks

- Is class A a subclass of class B?
- Can you devirtualize a call for the given receiver type?
- Generate LLVM IR for the given method
- ...

Falcon



**What does it take to make
a Java JIT from LLVM?**

LLVM as a JIT

- LLVM is mostly used for static compilation
 - Clang — the main user
- A lot of experiments to use LLVM as a JIT for a managed language
- Shark — (was) an experimental LLVM-based JIT compiler in Open JDK

What was missing?

- Precise Garbage Collection
- Speculative optimizations and deoptimizations
- Java-specific optimizations

GC-Compiler interaction

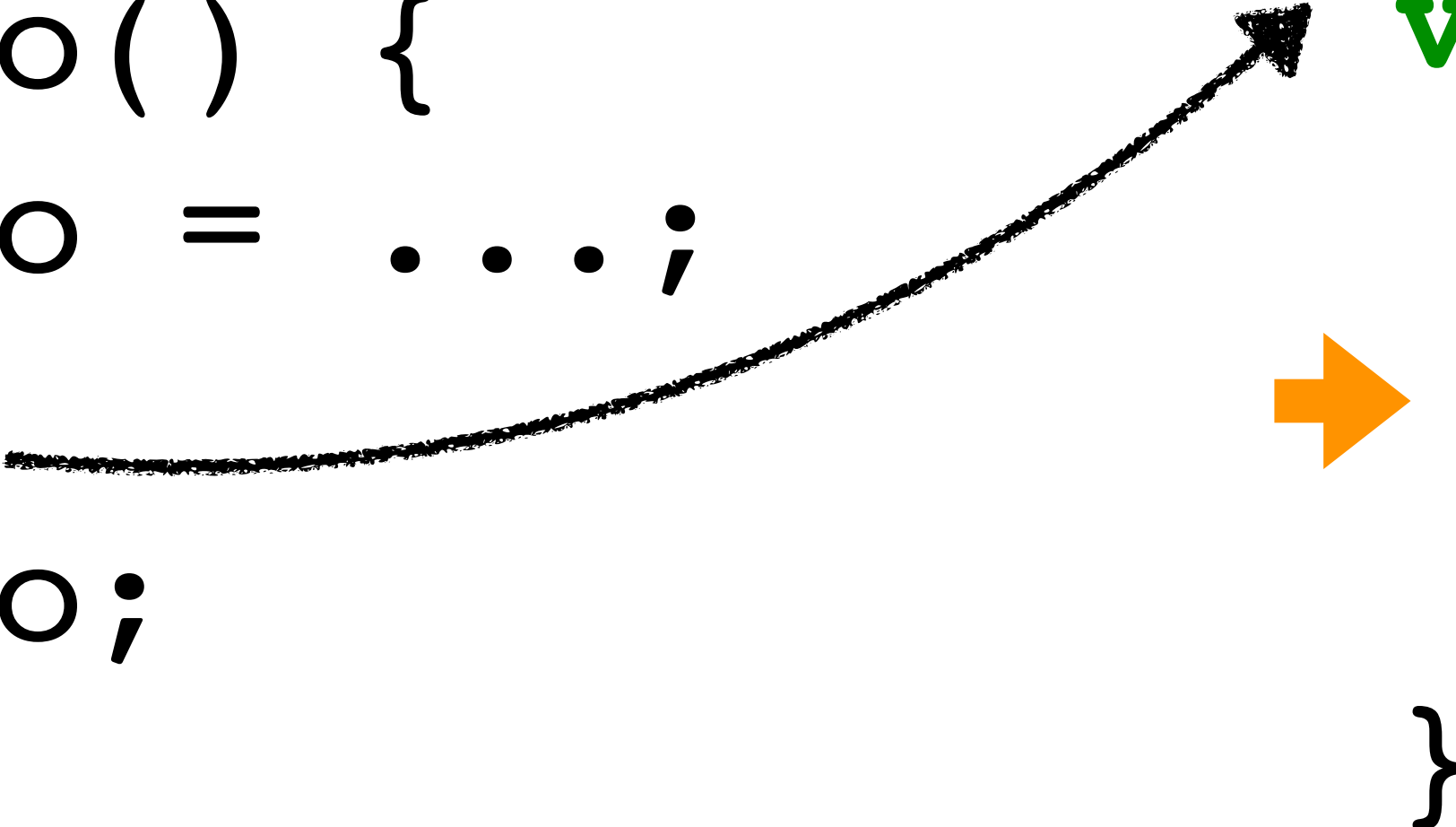
- How does the GC find the references?
- Stack, registers

GC-Compiler interaction

- How does the GC find the references?
 - Stack, registers
- GC safepoints — the GC can parse the stack at safepoints
- GC can ask a thread to pause at a safepoint

Thread is at a Safepoint

```
Object foo() {  
    Object o = ...;  
    bar();  
    return o;  
}  
  
void bar() {  
    ...  
    <<Wait for GC>>  
    ...  
}
```



GC should be able to parse the stack

GC Relocations

```
Object foo() {  
    Object o = ...;  
    bar();  
    return o;  
}
```

**The compiler should expect o to
be relocated**

GC Relocations

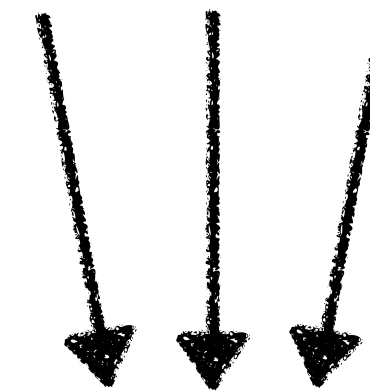
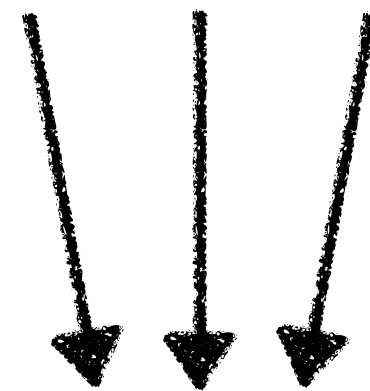
```
Object foo() {  
    Object o = ...;  
    _o = safepoint(bar(), o);  
    return _o;  
}
```

**The compiler should expect o to
be relocated**

Statepoint

function args...

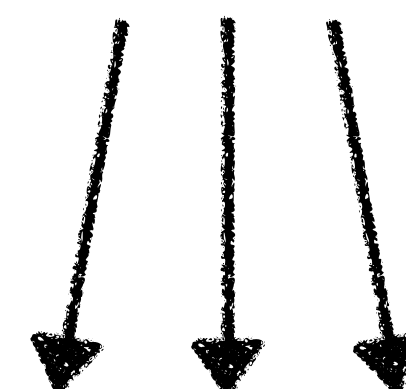
live pointers...



call gc.statepoint(<target function>)



return value



relocated pointers...

Statepoint Intrinsic

<http://llvm.org/docs/Statepoints.html>

```
declare token  
  @llvm.experimental.gc.statepoint(func_type <target>,  
    i64 <#call args>, ... (call parameters),  
    i64 <#deopt args>, ... (deopt parameters),  
    ... (gc parameters))
```

Statepoint Intrinsic

<http://llvm.org/docs/Statepoints.html>

```
declare token  
  @llvm.experimental.gc.statepoint(func_type <target>,  
    i64 <#call args>, ... (call parameters),  
    i64 <#deopt args>, ... (deopt parameters),  
    ... (gc parameters))
```

Lowered to a call to <target> + Stack Map

GC parses the stack using the Stack Map

Safepoints and optimizations

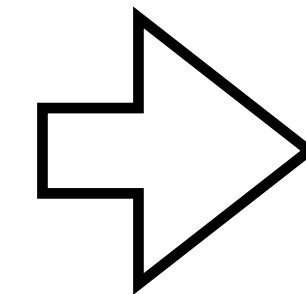
```
void foo(Object o) {  
    if (o instanceof String)  
        return;  
    _o = safepoint(bar(), o);  
    if (_o instanceof String)  
        return;  
    ...  
}
```

**The optimizer
doesn't know
anything about _o**



Safepoints and optimizations

```
void foo(Object o) {  
    if (o instanceof String)  
        return;  
    bar();  
    if (o instanceof String)  
        return;  
    ...  
}
```



```
void foo(Object o) {  
    if (o instanceof String)  
        return;  
    bar();  
    ...  
}
```

We don't use explicit relocations for optimizations

RS4GC

**Implicit relocations,
strong gc pointers**

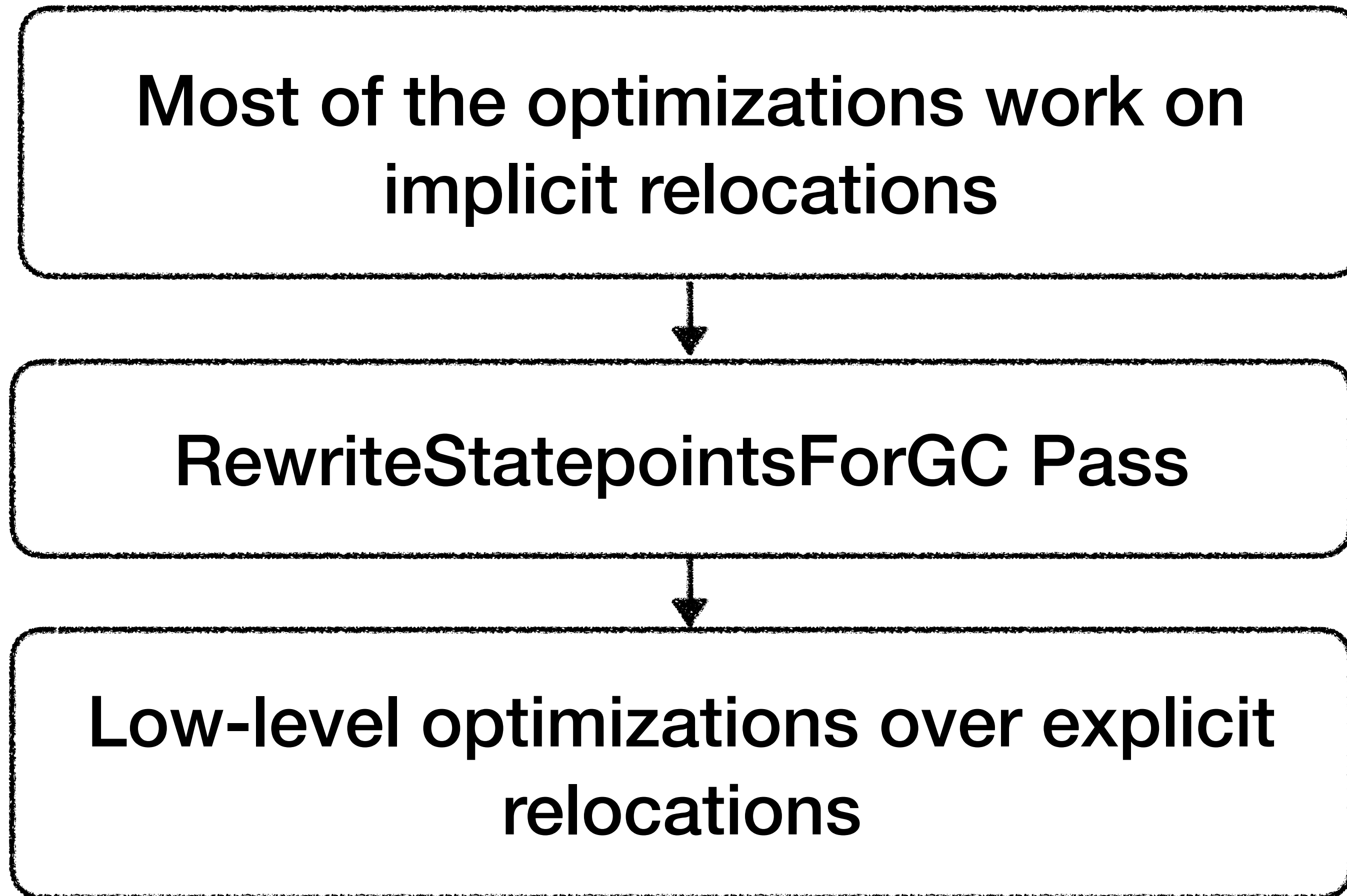


RewriteStatepointsForGC Pass



**Explicit relocations,
integral pointers**

Optimization Pipeline



What was missing?

- ✓ Precise Garbage Collection
- Speculative optimizations and deoptimization
- Java-specific optimizations

Uncommon Traps

Speculation example

```
if (cond)
    foo();
else
    bar();
```

Uncommon Traps

Profile...

```
if (cond)
    foo();    // 0
else
    bar();    // 10000
```

foo was never called!

Uncommon Traps

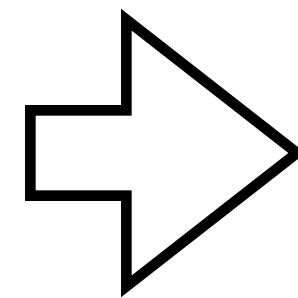
Will not compile the foo call!

```
if (cond)
    foo();
else
    bar();
```

Uncommon Traps

Will not compile the foo call!

```
if (cond)
    foo();
else
    bar();
```



```
if (cond)
    <uncommon_trap>
else
    bar();
```

Uncommon Traps

How to express an uncommon trap in
LLVM IR?

```
declare type  
  @llvm.experimental.deoptimize(...)
```

Deoptimization

```
final int f = 5;  
boolean foo() {  
    int f_ = f;  
    bar();  
    return f == f_;  
}
```

**Assume that
final f never
changes**

Deoptimization

```
final int f = 5;  
boolean foo() {  
    int f_ = f;  
    bar();  
    return f == _f;  
}
```

**Assume that
final f never
changes**

Deoptimization

```
final int f = 5;  
boolean foo() {  
    int f_ = f;  
    bar();  
    return true;  
}
```

**Assume that
final f never
changes**

Deoptimization

```
final int f = 5;
boolean foo() {
    int f_ = f;
    bar();
    return true;
}

void bar() {
    // abuse field
    // finality!
    // f = 1
}
```

Where do we return?

Deoptimization

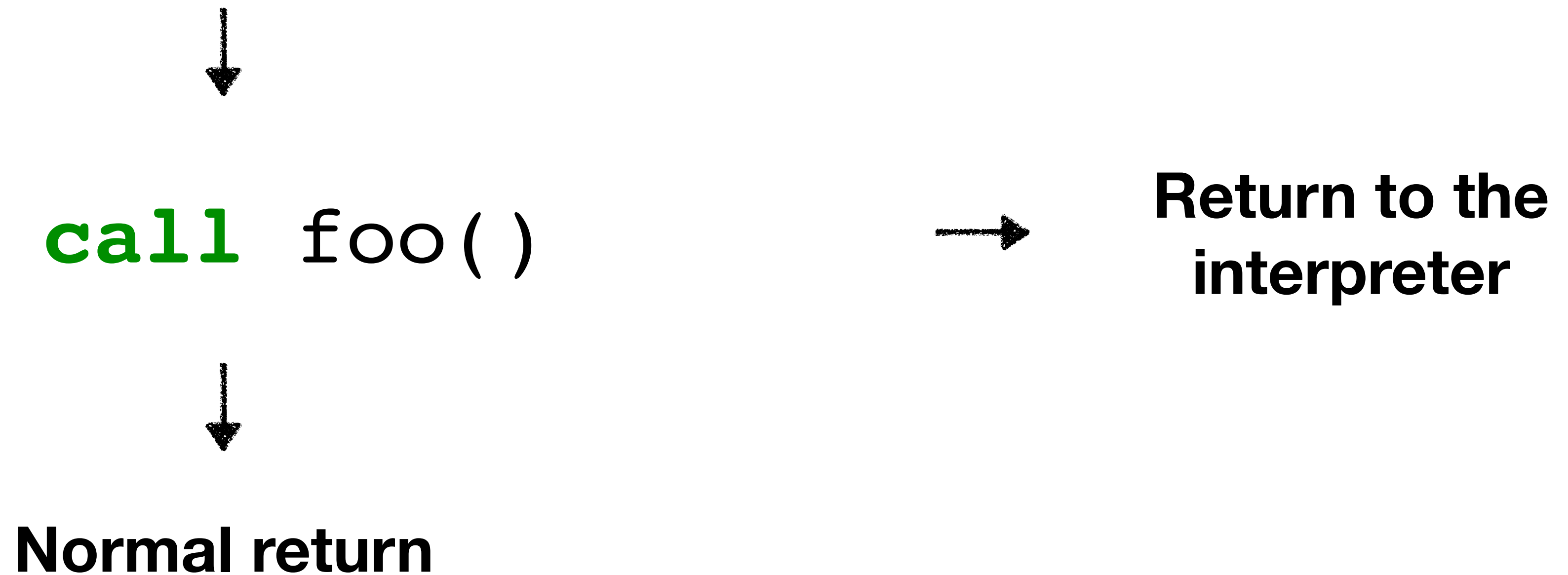
```
final int f = 5;
boolean foo() {
    int f_ = f;
    bar();
    return true;
}

void bar() {
    // abuse field
    // finality!
    // f = 1
}
```

To the interpreter!

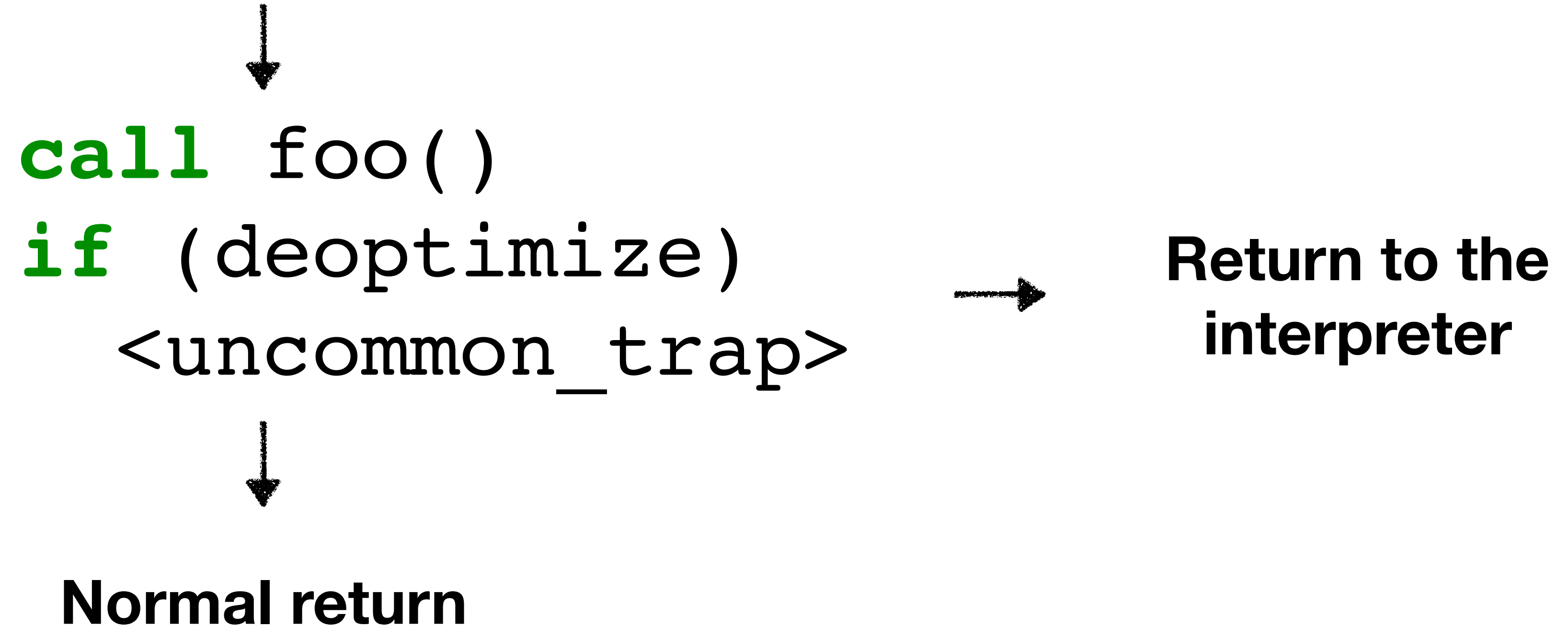
Return from a call

Any call has two execution continuations



Return from a call

Any call has two execution continuations



Interpreter State

```
boolean foo() {  
    int f_ = f;  
    bar();  
    if (deoptimize)  
        <uncommon_trap>  
    return true;  
}
```



```
0:  iconst_1  
1:  istore_1  
2:  aload_0  
3:  invokevirtual bar()  
6:  iconst_1  
7:  iload_1  
8:  if_icmpne 15  
11: iconst_1  
12: goto 16  
15: iconst_0  
16: ireturn
```

Interpreter State

```
boolean foo() {  
    int f_ = f;  
    bar();  
    if (deoptimize)  
        <uncommon_trap>  
        [bci=6, local_1=f_]  
    return true;  
}
```

```
0:  iconst_1  
1:  istore_1  
2:  aload_0  
3:  invokevirtual bar()  
6:  iconst_1  
7:  iload_1  
8:  if_icmpne 15  
11: iconst_1  
12: goto 16  
15: iconst_0  
16: ireturn
```

Interpreter State

```
boolean foo() {  
    int f_ = f;  
    bar();  
    if (deoptimize)  
        <uncommon_trap>  
        [bci=6, local_1=f_]  
    return true;  
}
```

```
0:  iconst_1  
1:  istore_1  
2:  aload_0  
3:  invokevirtual bar()  
6:  iconst_1  
7:  iload_1  
8:  if_icmpne 15  
11: iconst_1  
12: goto 16  
15: iconst_0  
16: ireturn
```

Interpreter State

- Interpreter state consists of:
 - BCI, locals, stack, monitors
- Runtime should know where to get these values

Interpreter State

How to express this in LLVM IR?

```
call void @f(i32 %arg)  
[ "deopt"(i32 0, i8* %a, i32* null) ]
```

Operand Bundles

<http://llvm.org/docs/LangRef.html#operand-bundles>

- Generic mechanism to add a set of values to a call
- Use Deopt Bundle to describe the interpreter state

Deopt Bundle Lowering

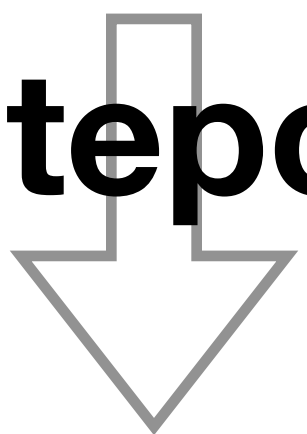
How does the runtime find out these values?

```
call void @f(i32 %arg)
  [ "deopt"(i32 0, i8* %a, i32* null) ]
```

Deopt Bundle Lowering

```
call void @f(i32 %arg)
  [ "deopt"(i32 0, i8* %a, i32* null) ]
```

RewriteStatepointsForGC



```
call token
  @llvm.experimental.gc.statepoint(void @f(i32),
    i64 1, i32 %arg,
    i64 3, i32 0, i8* %a, i32* null,
    ... (gc parameters))
```

Deopt Bundle Lowering

```
call token  
  @llvm.experimental.gc.statepoint(void @f(i32),  
    i64 1, i32 %arg,  
    i64 3, i32 0, i8* %a, i32* null,  
    ... (gc parameters))
```

Stack Map is generated

Runtime uses the Stack Map to find the values

What was missing?

- ✓ Precise Garbage Collection
- ✓ Speculative optimizations and deoptimization
- Java-specific optimizations

High-level semantics

- High-level semantics should be taken into account for optimizations
- E.g. devirtualization
- LLVM IR — low level representation
- LLVM doesn't know anything about Java semantics

High Level Embedded IR

- High level IR embedded in LLVM IR
- Attributes and metadata for type annotations
- Abstractions — high level intrinsics
- Functions with the semantics known by the optimizer

High Level Embedded IR Example

```
int foo(Object o) {  
    if (o instanceof Integer)  
        if (o instanceof Number)  
            return o.hashCode();  
    return 0;  
}
```

High Level Embedded IR

Example

Attributes

```
int foo(int* "java-type-class"="Object" o) {  
    o_class = @get_class(o);  
    if (@is_subtype_of(Integer, o_class))  
        if (@is_subtype_of(Number, o_class)) {  
            target =  
                @resolve_virtual(o, hashCode);  
            return target(o);  
        }  
    return 0;  
}
```

High Level Embedded IR Example

Abstractions

```
int foo(int* "java-type-class"="Object" o) {  
    o_class = @get_class(o);  
    if (@is_subtype_of(Integer, o_class))  
        if (@is_subtype_of(Number, o_class)) {  
            target =  
                @resolve_virtual(o, hashCode);  
            return target(o);  
        }  
    return 0;  
}
```

High Level Embedded IR Optimizations

```
int foo(i8* "java-type-class"="Object" o) {  
    o_class = @get_class(o);  
    if (@is_subtype_of(Integer, o_class))  
        if (@is_subtype_of(Number, o_class)) {  
            target =  
                @resolve_virtual(o, hashCode);  
            return target(o);  
        }  
    return 0;  
}
```

After the check
o_class is Integer!



High Level Embedded IR Optimizations

```
int foo(i8* "java-type-class"="Object" o) {  
    o_class = @get_class(o);  
    if (@is_subtype_of(Integer, o_class))  
        if (@is_subtype_of(Number, o_class)) {  
            target =  
                @resolve_virtual(o, hashCode);  
            return target(o);  
        }  
    return 0;  
}
```

Integer is a
subtype of
Number



High Level Embedded IR Optimizations

resolve_virtual for
Integer is
Integer::hashCode

```
int foo(i8* "java-type-class"="Object" o) {  
    o_class = @get_class(o);  
    if (@is_subtype_of(Integer, o_class)) {  
        target =  
        → @resolve_virtual(o, hashCode);  
        return target(o);  
    }  
    return 0;  
}
```

High Level Embedded IR Optimizations

```
int foo(i8* "java-type-class"="Object" o) {  
    o_class = @get_class(o);  
    if (@is_subtype_of(Integer, o_class))  
        return Integer::hashCode(o);  
    return 0;  
}
```

Abstraction Lowering

- VM provides implementations for the abstractions
- Abstractions' bodies are inlined after high-level optimizations
- After that low-level optimizations can be applied

What was missing?

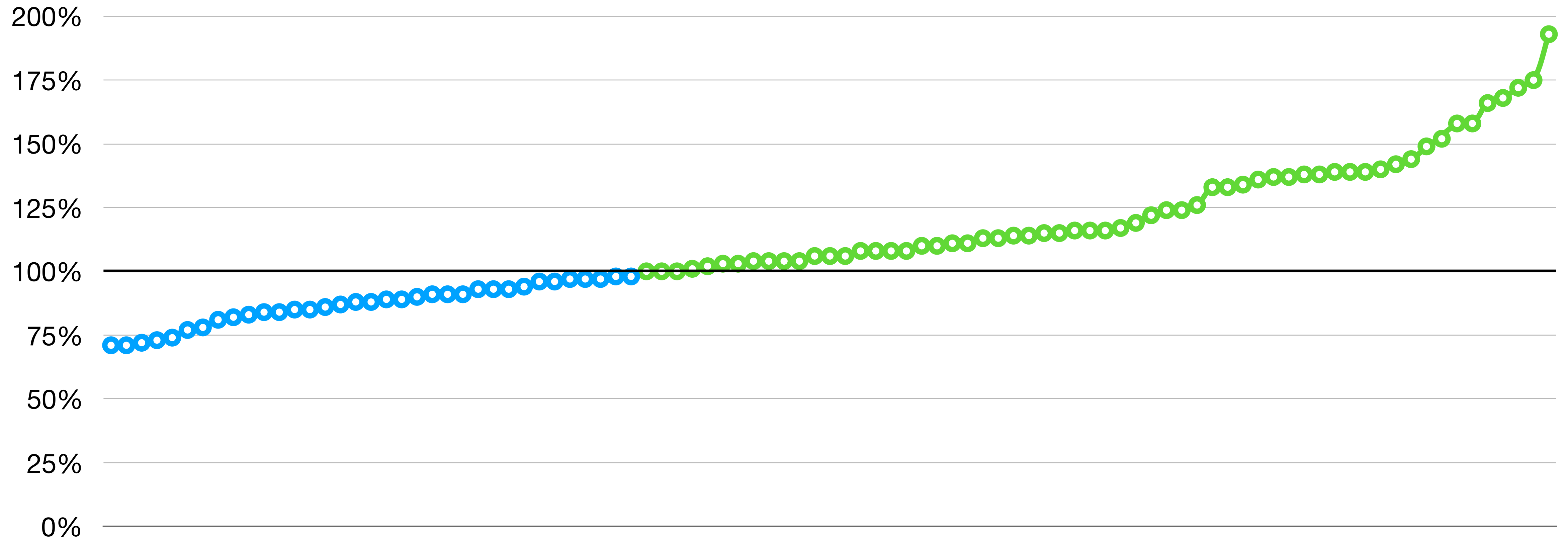
- ✓ Precise Garbage Collection
- ✓ Speculative optimizations and deoptimization
- ✓ **Java-specific optimizations**

Our interaction with LLVM Upstream

- Most of the code is upstreamed
- GC infrastructure
- Speculative optimizations infrastructure
- Optimizations improvements

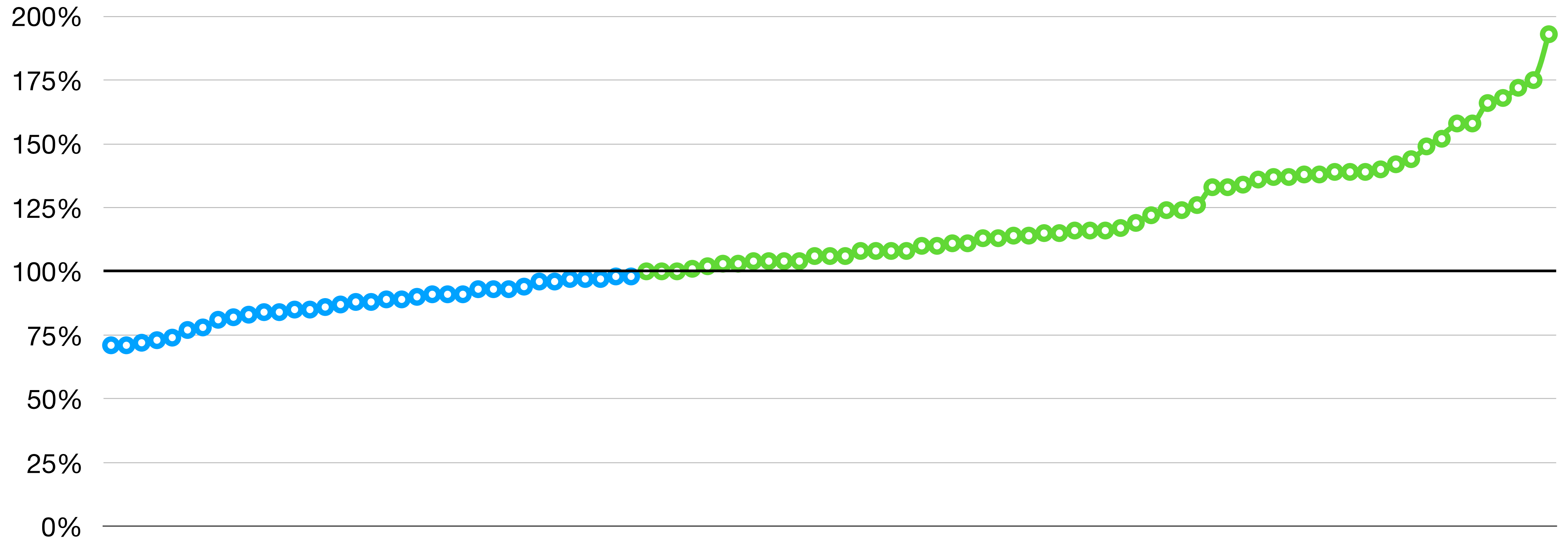
**How do we stand
against competition?**

Zing 18.02 vs Oracle 8u152



SPECjvm + Dacapo + application benchmarks
skylake + haswell

Zing 18.02 vs Oracle 8u152



63% win rate

Falcon vs C2

Robustness

- Fuzzing testing, in 100 000 tests
- Zing Falcon — 1-2 crashes
- OpenJDK C2 — ~10 crashes

Any downsides?

A vector arithmetic microbenchmark

```
double eachIteration(double[] input, int phase) {
```

```
    if (phase == 1) {
        return Case1.case1(input);
    } else {
        if (phase == 2) {
            return Case2.case2(input);
        } else {
            if (phase == 3) {
                return Case3.case3(input);
            } else {
                return Case4.case4(input);
            }
        }
    }
}
```

```
static double case1(double[] input) {
    double result = 0;
    for (int i = 0; i < input.length; i++) {
        result += Math.sin(input[i])
                / Math.cos(input[i]);
    }
    return result;
}
```

```
}
```

Iteration times

OpenJDK
8u152

Zing C2

Falcon

Score (lower is better)

712 us/op

438 us/op

223 us/op

Iteration times / compile times

Zing C2

Falcon

Score (lower is better)

458 us/op

228 us/op

Total compile time

49 ms

467 ms

Wait time in a compiler queue

0 methods over 10 ms

8 methods over 10 ms

Reducing compile time

or combat higher CPU usage

- Reduce number of compilations
 - ReadyNow
- Reuse previous compilations
- Compile Stashing

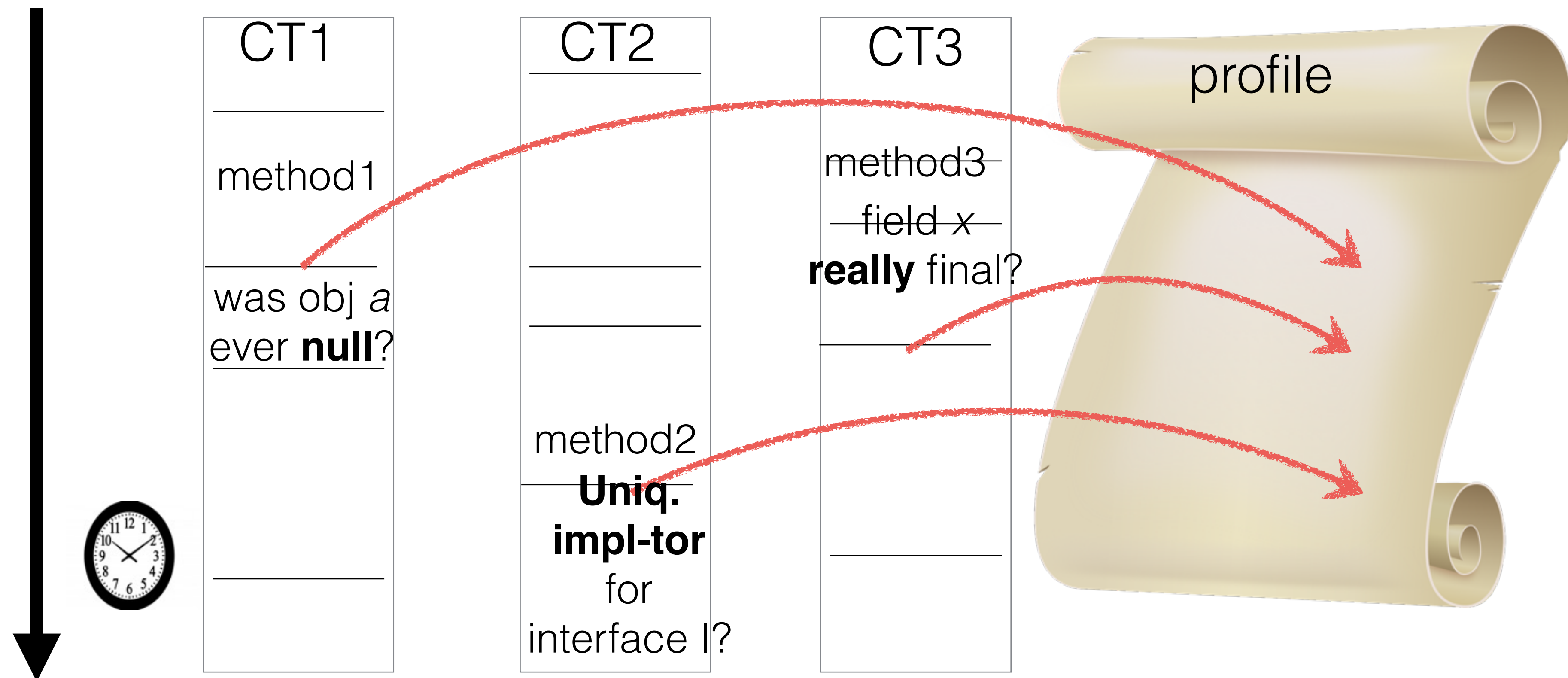


ReadyNow

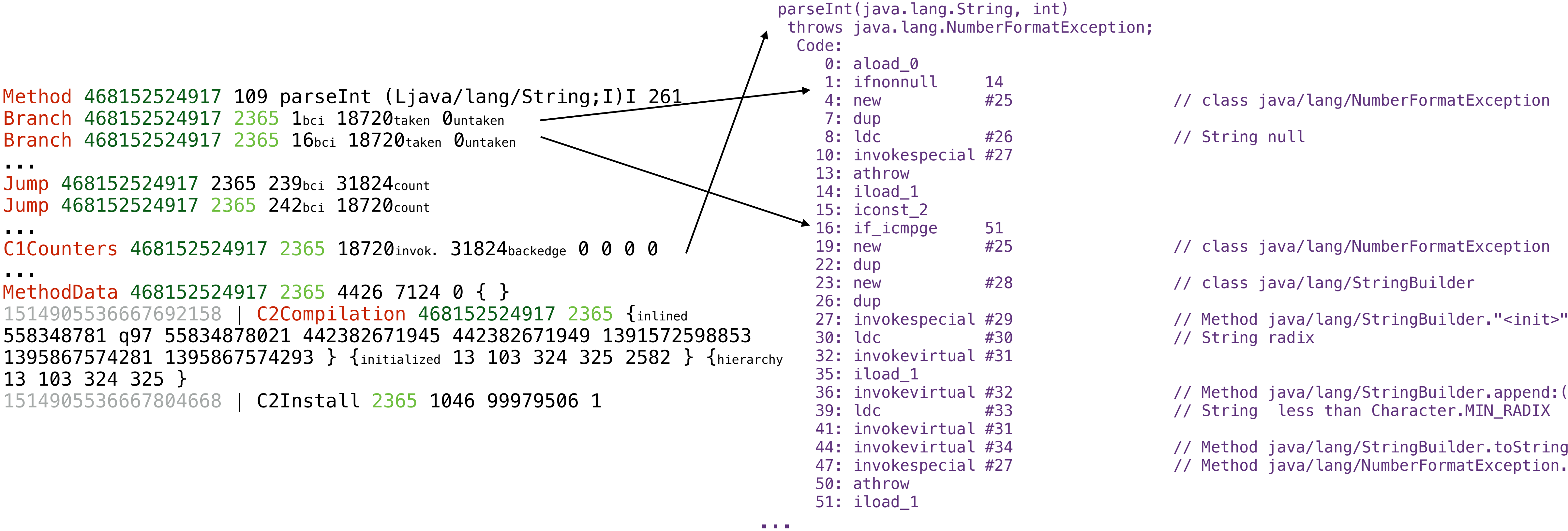
- Persistent profile driven compilation
- Similar to PGO in C++ and other compilers
- Reduces warmup times and number of deoptimizations



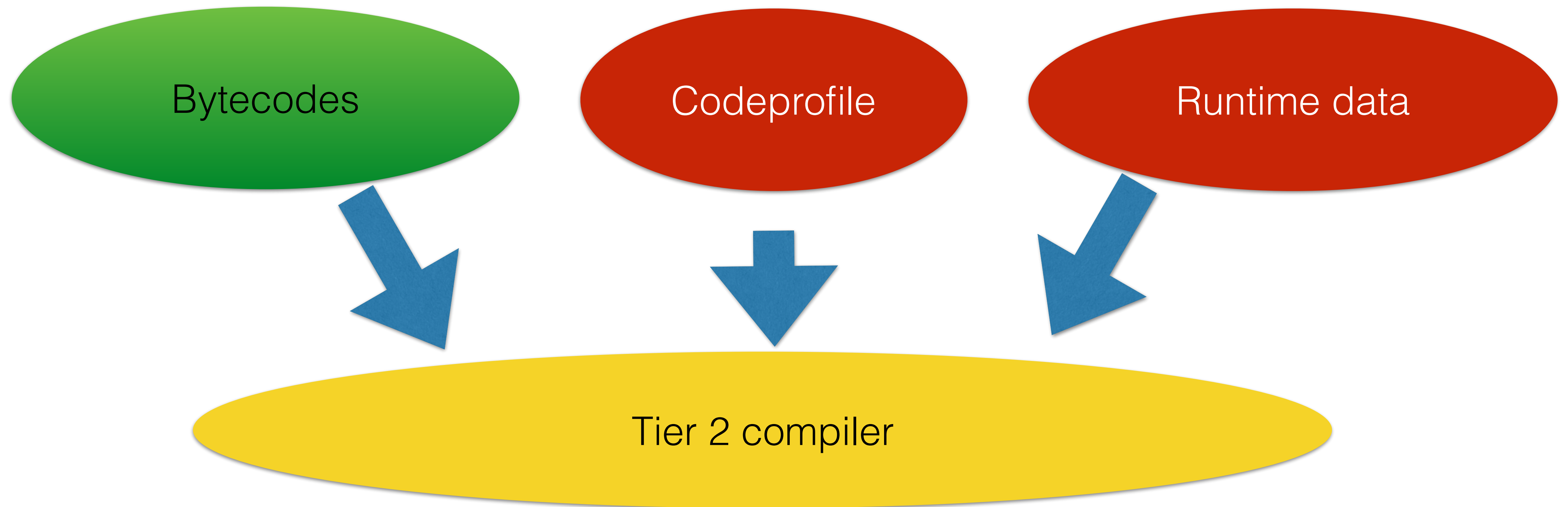
Recording a profile



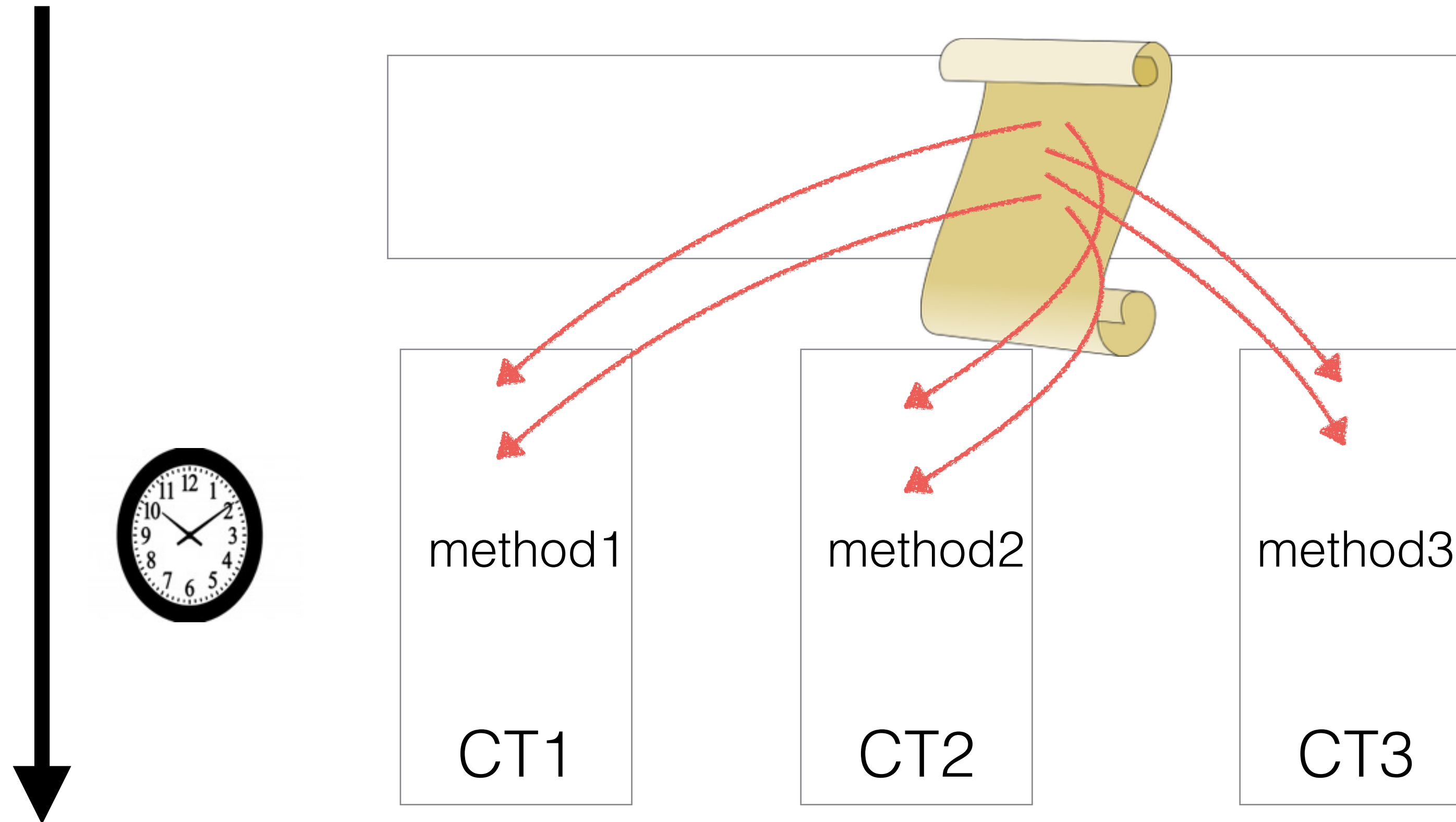
Serialized codeprofile



Inputs to compile



Using a profile



Deoptimizations without/with profile

3 total

0 total

unloaded

3

0

of top tier compiles

9

6

Same test with ReadyNow

Falcon

Falcon + ReadyNow

Score (lower is better)

228 us/op

226 us/op

Time the top tier compiler

467 ms

284 ms

Worse op time

2111 us

514 us

Reducing compile time

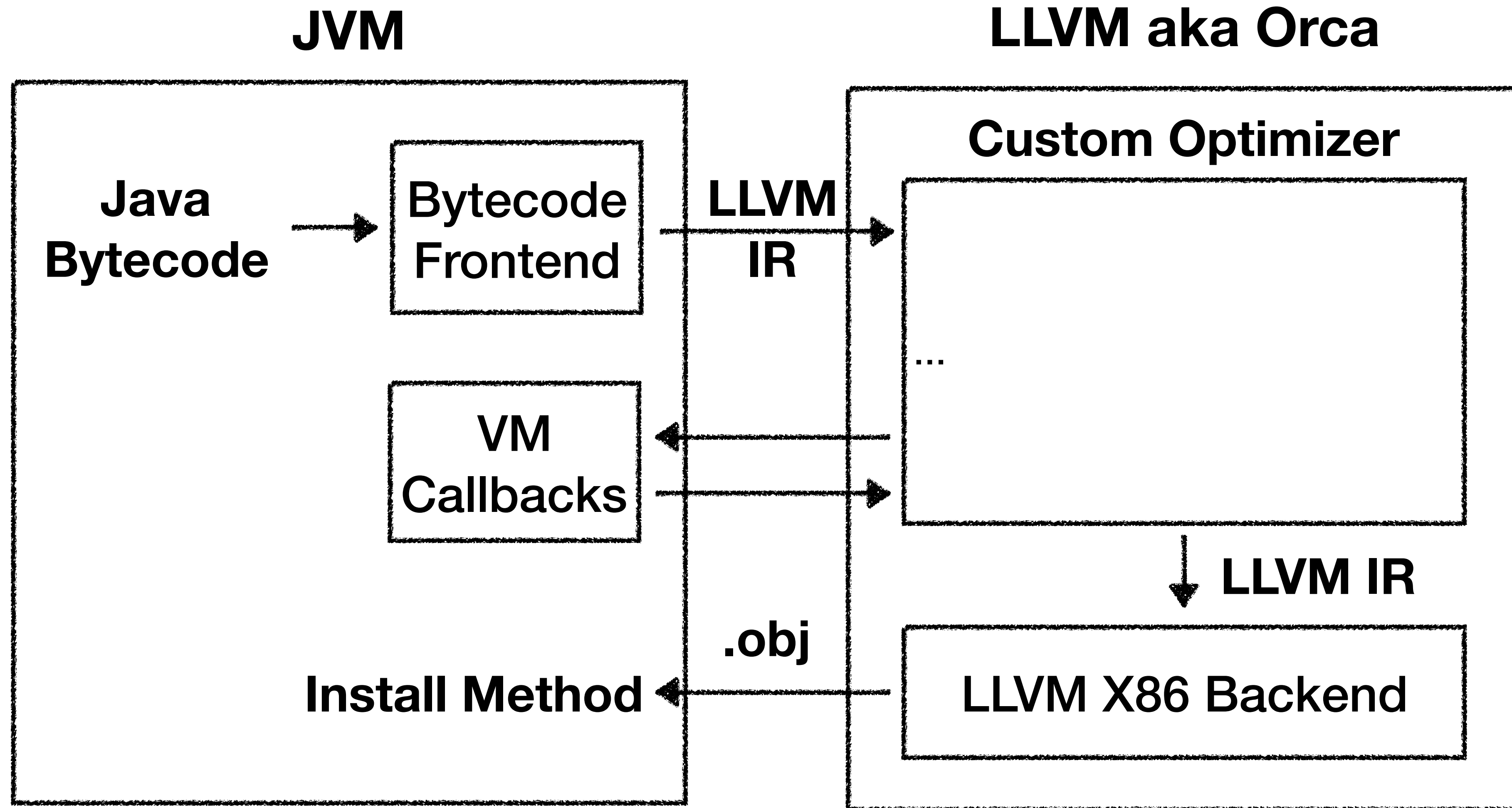
- ✓ Reduce number of compilations caused by late deoptimizations
- ✓ ReadyNow
- Reuse previous compilations
- Compile Stashing

Compile Stashing

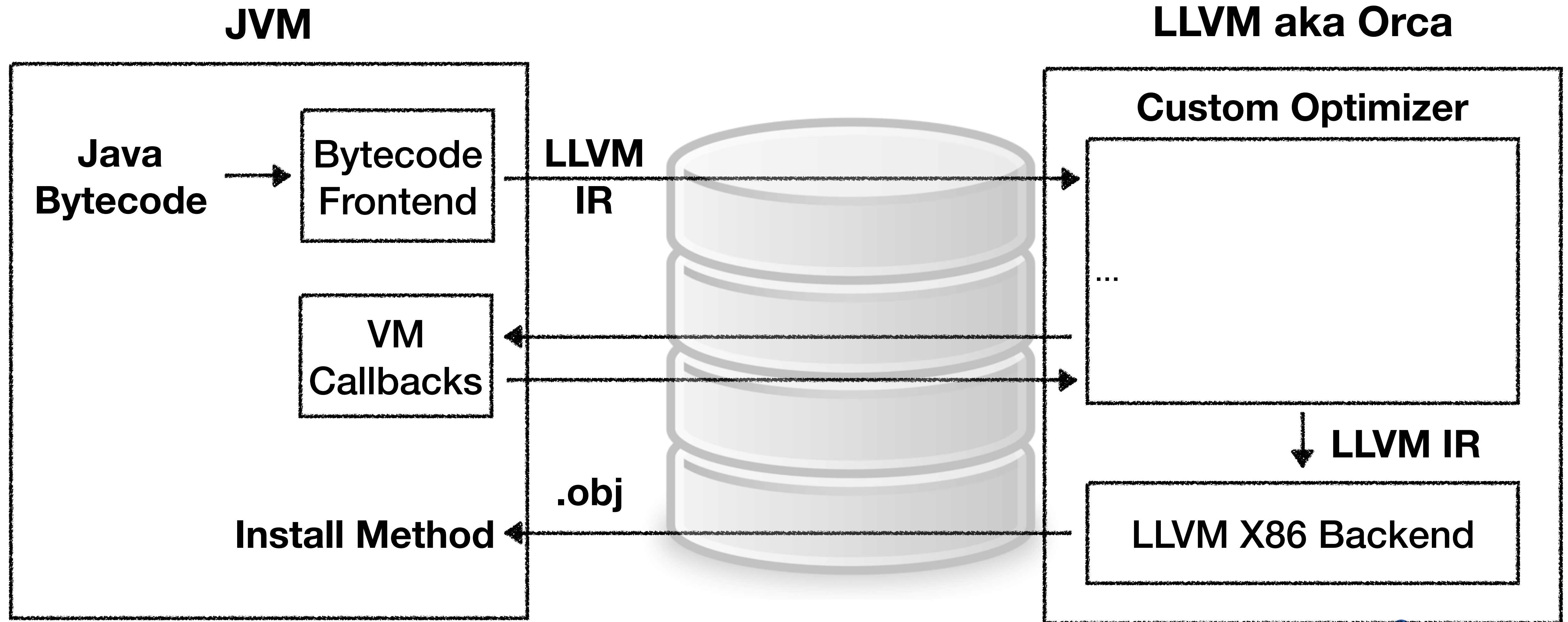
- Reuse top-tier compilation
- Applied to JDK and user code
- Different from JAOT
- Challenges are the same as for JOAT



Compile Stashing - Write

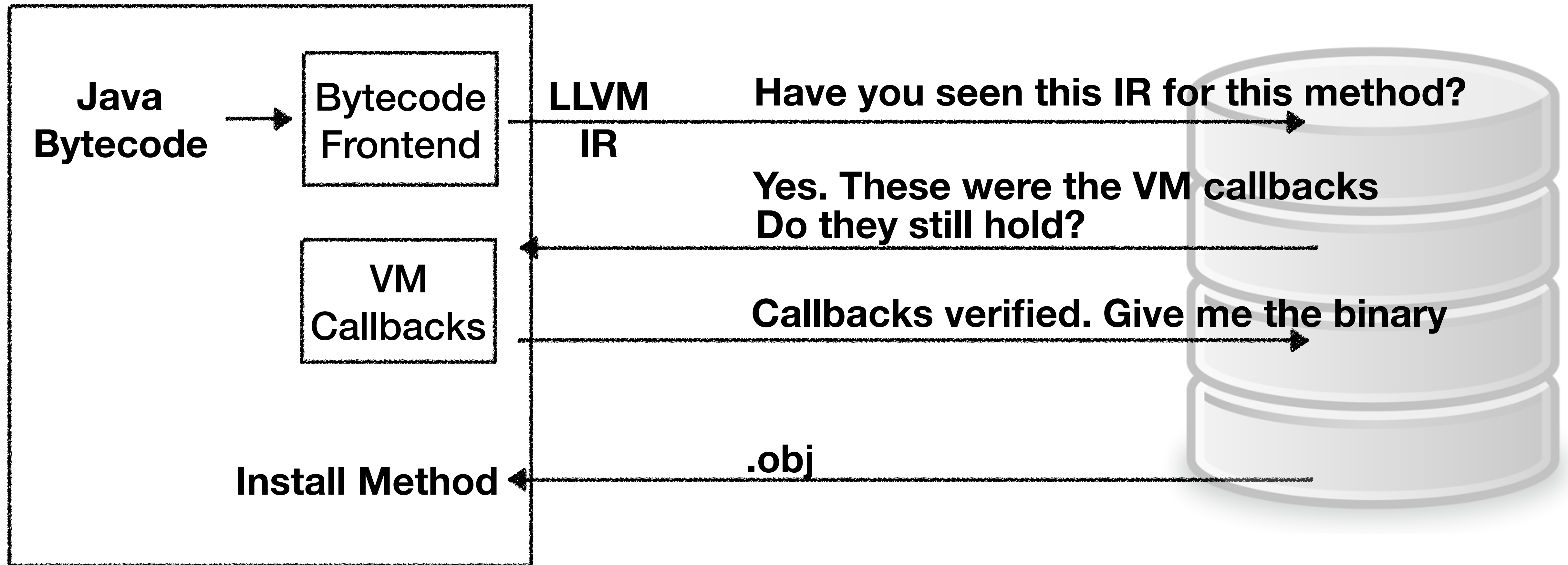


Compile Stashing - Write



Compile Stashing - Use

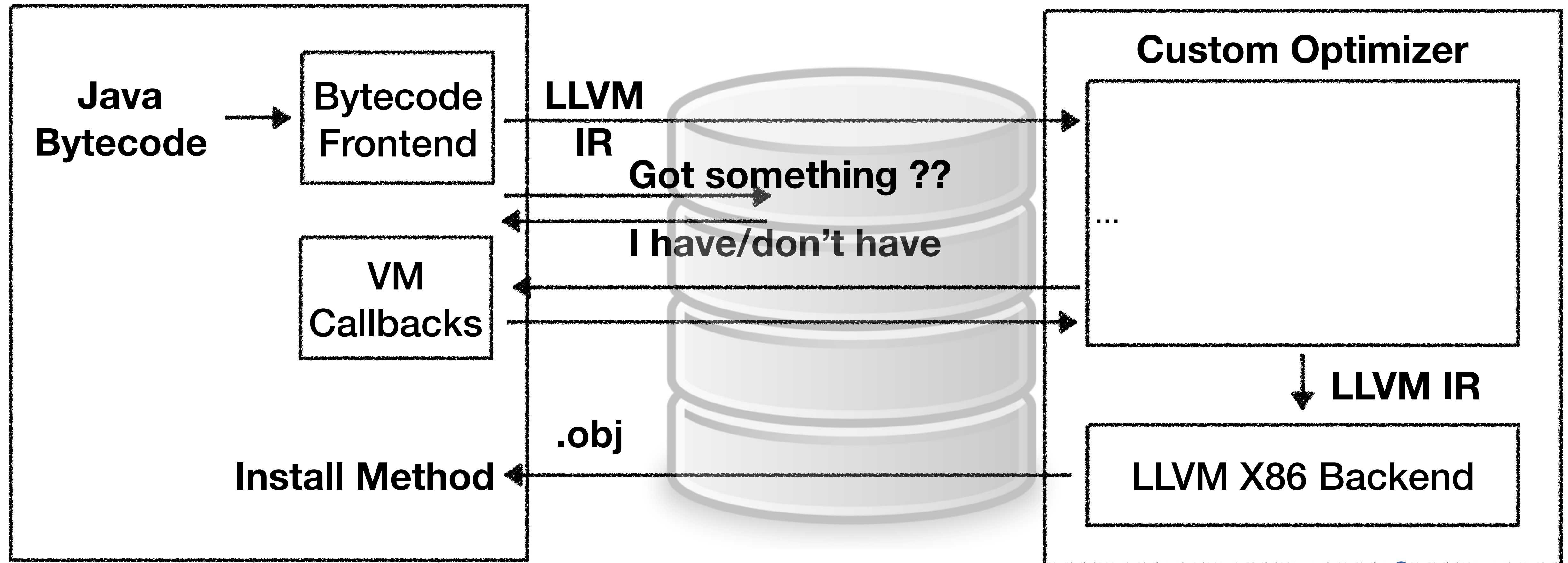
JVM



Compile Stashing - Use and write

JVM

LLVM aka Orca



A vector arithmetic microbenchmark

Falcon + RN

Falcon + RN + Stashing

Time in the compiler

284 ms

53 ms/op

Score (lower is better)

226 us/op

228 us/op

Problem solved? ..

- Real-world match rate is 10% - 50 %
- (But we are improving every month)
- Some due to well-known problems
 - Un-identifiable classloaders
 - Instrumenting and other transforming agents
 - Generated classes

Can I try it?

- ReadyNow
 - In the product since 2015
- Falcon
 - Since December 2016
- Compile Stashing
 - Experimental feature
 - Since 17.12



Conclusion

- Falcon - an LLVM-based JIT
 - in production, now the default compiler
 - delivers well performing binaries
 - easier to maintain and advance
- Made a lot of work building infrastructure in LLVM upstream
- Technologies to re-use compilations from previous runs



Questions?

How to try?

<http://www.azul.com/zingtrial/>

Artur Pilipenko
apilipenko@azul.com
@arturpilipenko

Ivan Krylov
ivan@azul.com
@JohnWings