Performance Tuning in Production

DEVOXX United Kingdom

James Gough
Sadiq Jaffer
Kirk Pepperdine
Richard Warburton

@Jim__Gough, @opsian, @RichardWarburto, @kcpeppe

Session Overview

- Optimizing Java: a brief tour of the JVM
- Moving to G1GC
- Production Profiling: What, Why and How

Optimizing Java: A JVM Tour

DEVOXX United Kingdom

James Gough

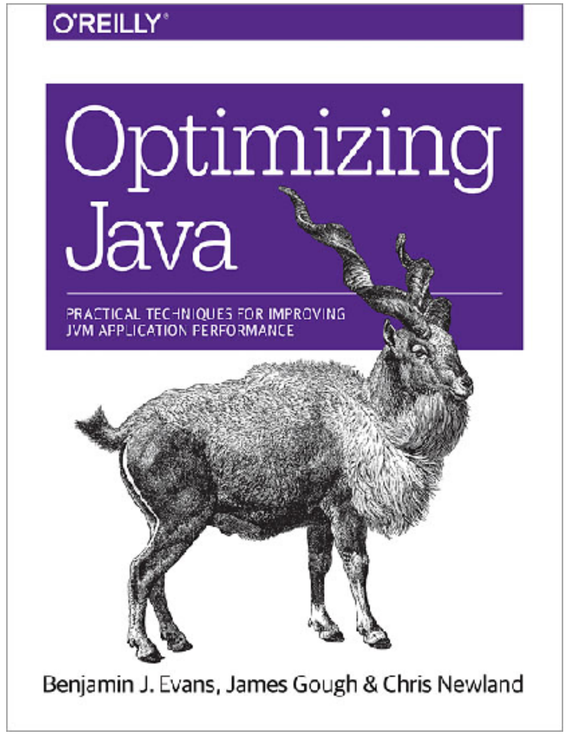
@Jim__Gough
http://jamesgough.net

This Talk

- Who Am I
- Creating Bytecode
- Classloading
- Profiling Code
- Runtime Optimisations
- JITWatch

About Me

- Started programming BASIC on the C64
- Worked as a Java and Web Developer
- Helped to design and test JSR 310
- Spent 4 years training Java and C++
- Written a book called Optimizing Java
- Work at Morgan Stanley
  - Building Client Facing Technology

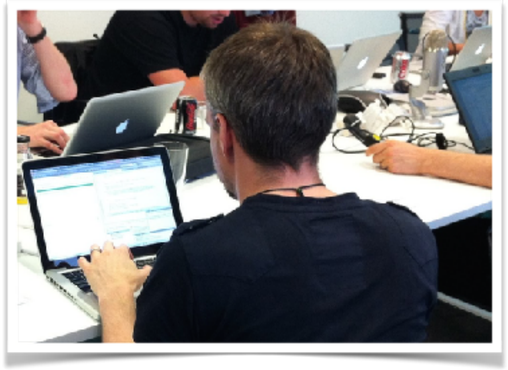


JMANC

30.06.2018

Creating Bytecode

+

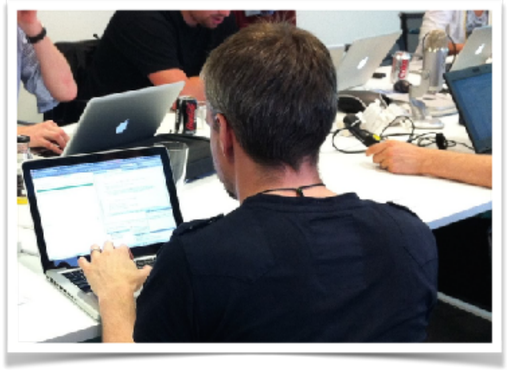


=

JVM

Creating Bytecode

+

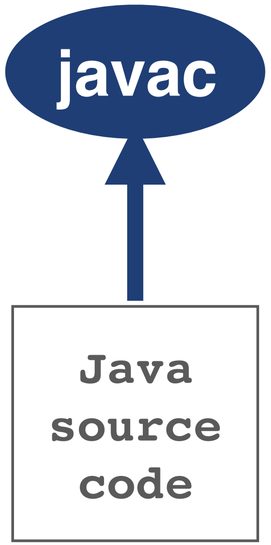


=

```
Java
source
code
```

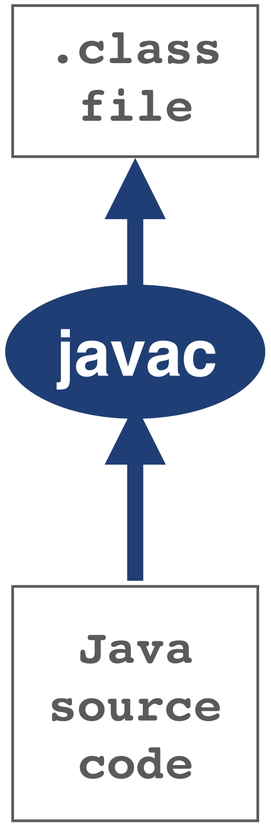
JVM

Creating Bytecode



Class file creation

Creating Bytecode



Class file creation

JVM

The anatomy of a classfile

Magic Number	0xCAFEBABE
Version of Class File Format	The minor and major versions of the class file
Constant Pool	Pool of constants for the class
Access Flags	For example whether the class is abstract, static, etc.
This Class	The name of the current class
Super Class	The name of the super class
Interfaces	Any interfaces in the class
Fields	Any fields in the class
Methods	Any methods in the class
Attributes	Any attributes of the class (e.g. name of the sourcefile, etc.)

The anatomy of a classfile

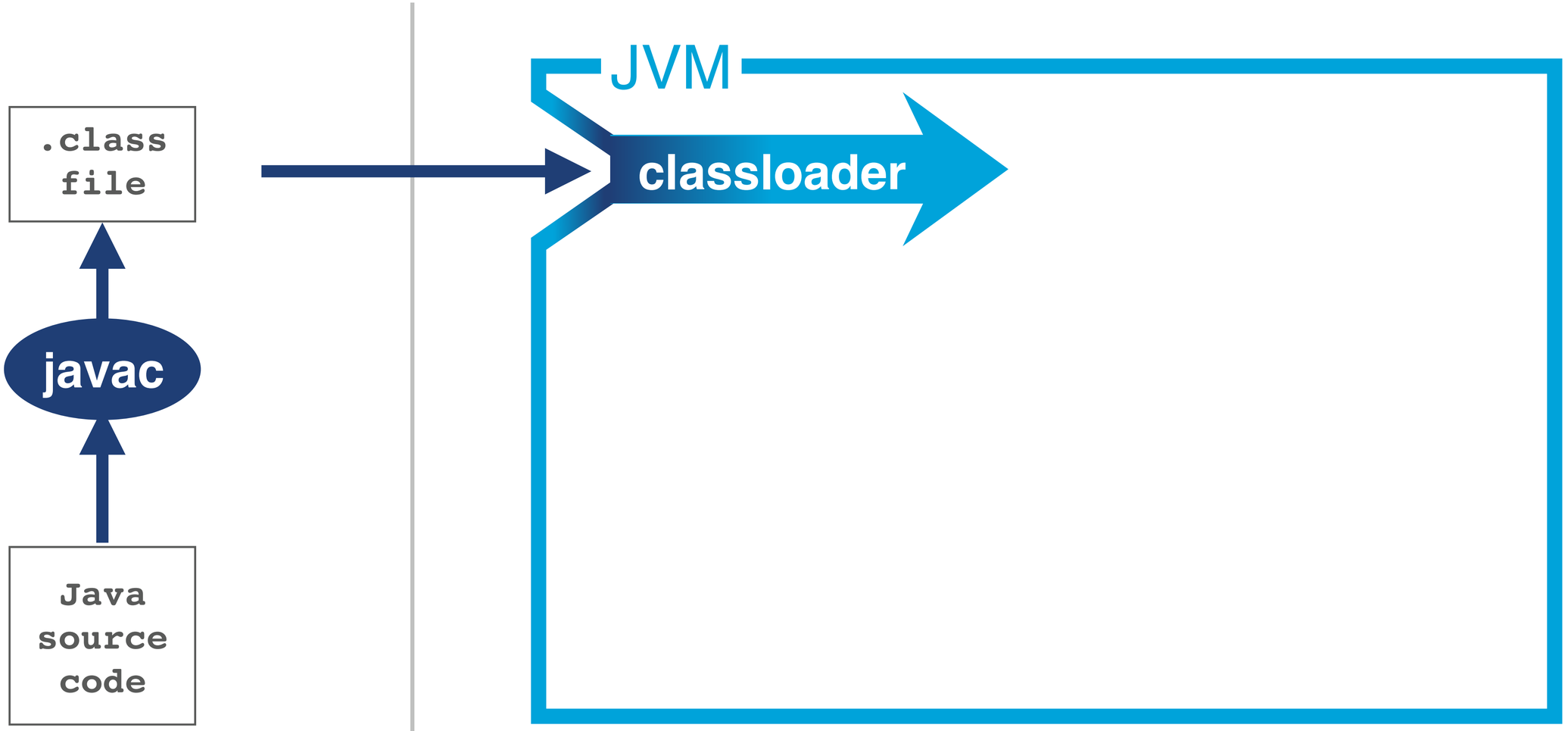
My	Very	Cute	Animal	Turns	Savage	In	Full	Moon	Areas
M	**V**	**C**	**A**	**T**	**S**	**I**	**F**	**M**	**A**
Magic	Version	Constant	Access	This	Super	Interfaces	Fields	Methods	Attributes

Type Descriptors

- Describe signatures
- Common in `javap` output
- E.g.
  - `()Ljava/lang/String;`
  - `(I)V`
  - `(Ljava/lang/String;I)J`

Descriptor	Type
`B`	`byte`
`C`	`char`
`D`	`double`
`F`	`float`
`I`	`int`
`J`	`long`
`L<type>;`	Reference type
`S`	`short`
`Z`	`boolean`
`[`	Array-of

How Bytecode is executed



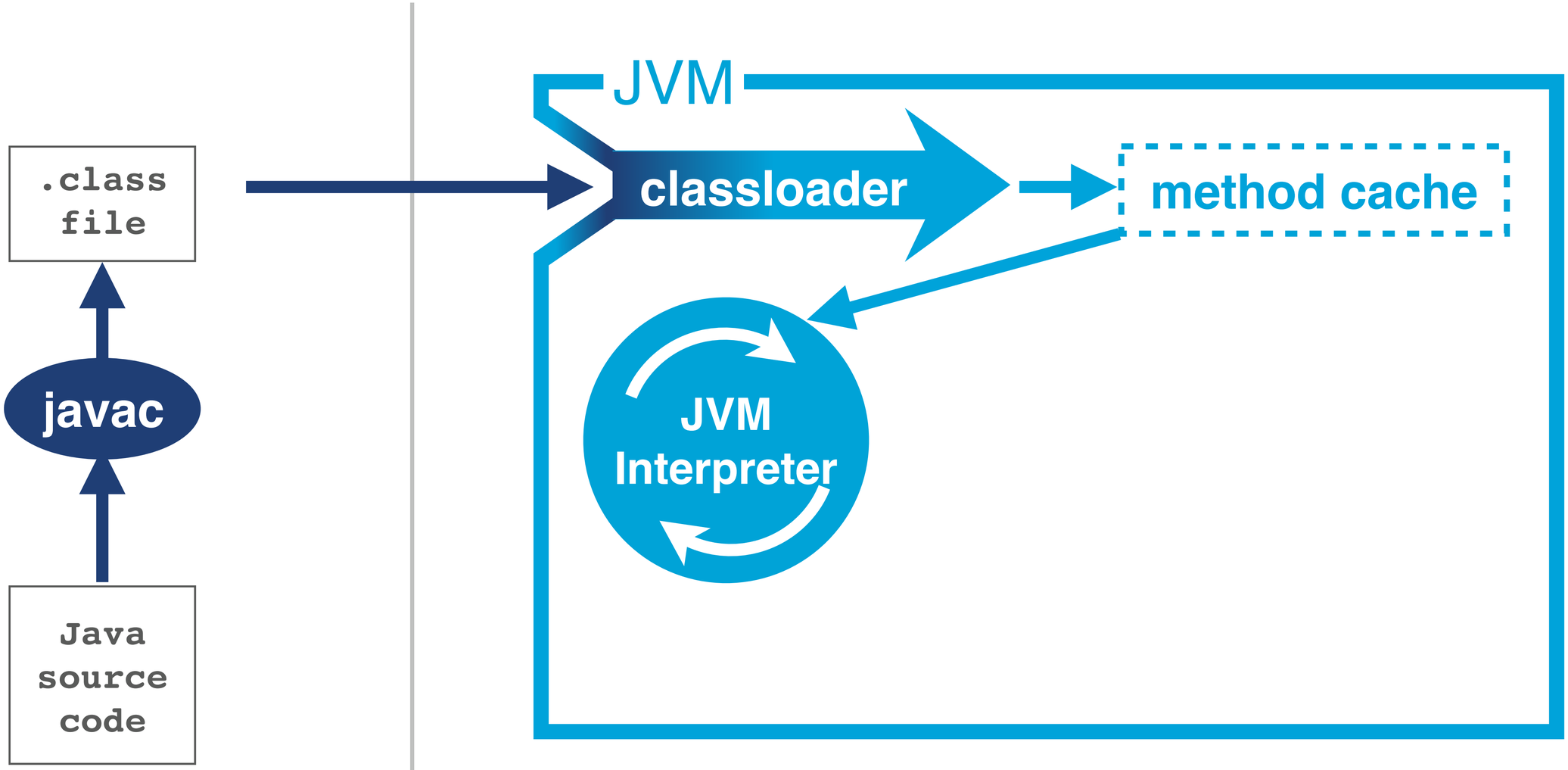
Class file creation

Continuous (Just In Time) compilation

Classloaders

- Classes are loaded just before they are needed
  - proven by the painful ClassNotFoundException
- Loads classfile into the Class object
  - mechanism for representing classes in the VM
- Example used in **watching-classloader**
  - https://github.com/jpgough/watching-classloader

Interpreting Bytecode



Class file creation

Continuous (Just In Time) compilation

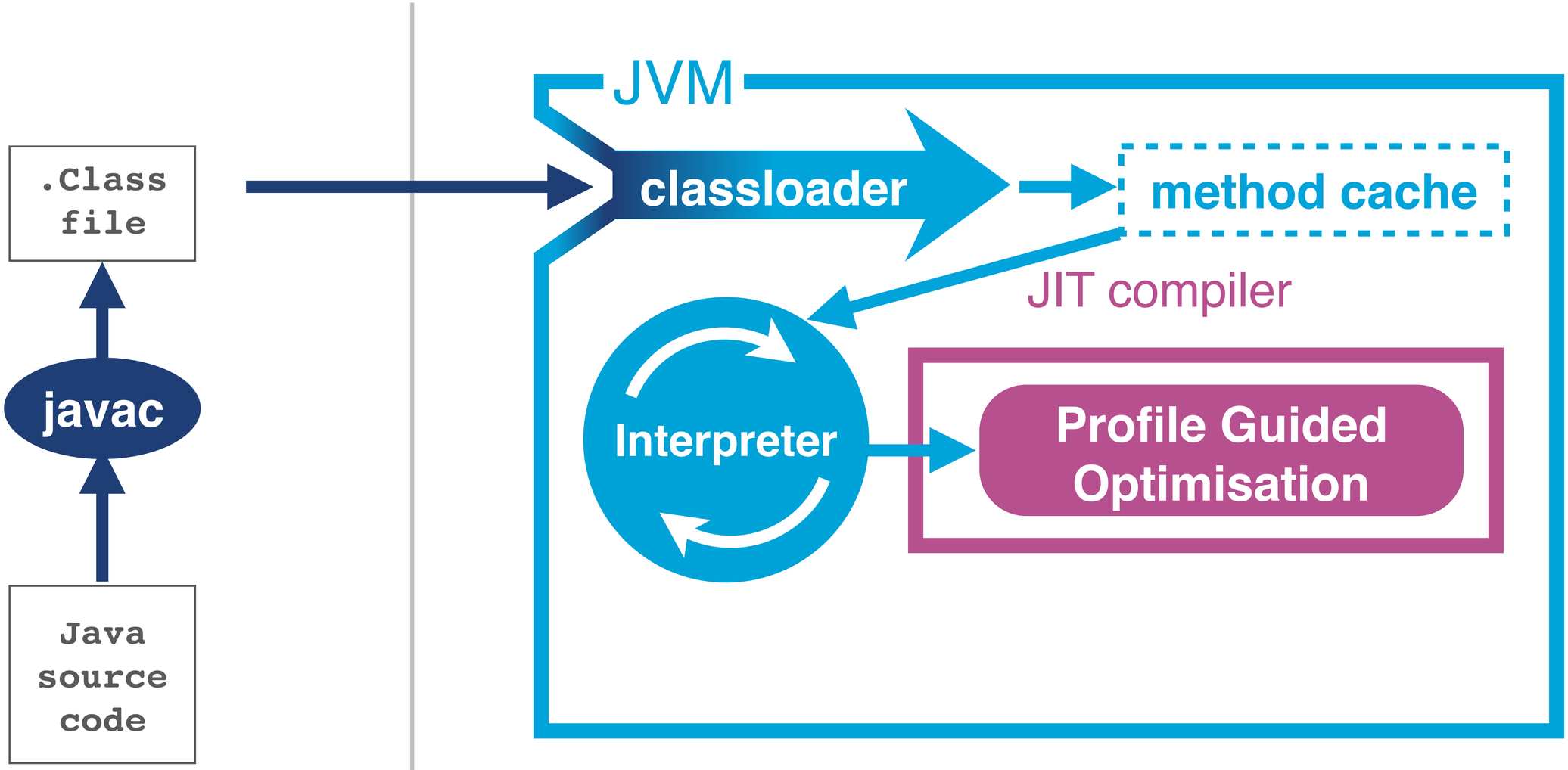
Interpreting Bytecode

- Bytecode initially fully interpreted
- Conversion of each instruction to machine instruction
- Time not spent compiling code that is only used once

How does Interpreting Help?

- Provides the opportunity to observe code execution paths
  - may not be the same for each execution of the app
- The profiler observes the execution and looks for the best optimisations
- Code is compiled after hitting a threshold
  - Configurable
  - JVM can revert optimisation decisions

Profiling Code



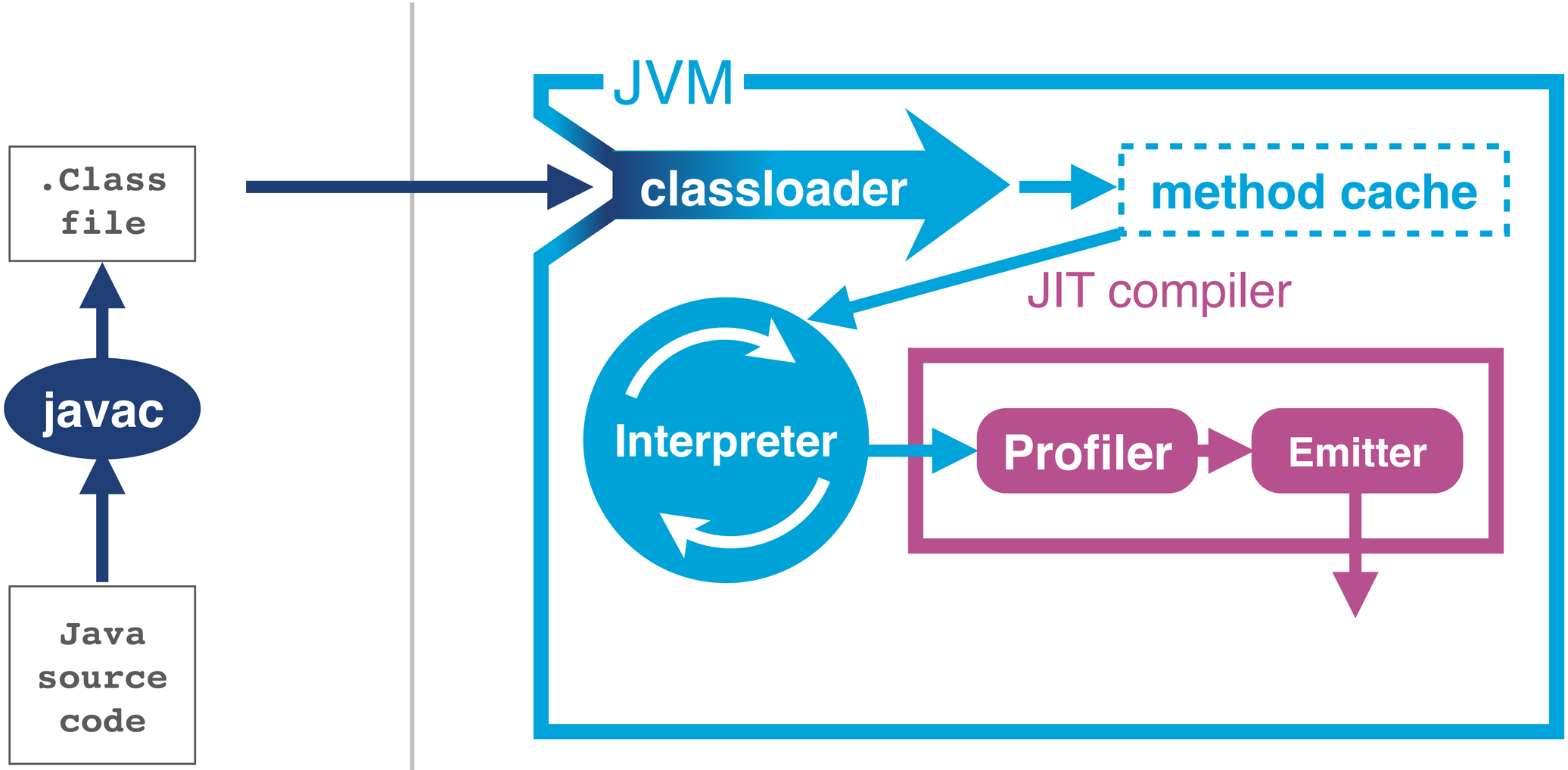
Class file creation

Continuous (Just In Time) compilation

Profiling Code

- Looking for loops or frequent execution of code blocks
- Barometer used to count the number of executions
- Threshold is reached and mode changes to tracing
- Tracing follows the execution path involving that method
  - proactively looking for optimisation opportunities
  - often stored as an intermediate representation
  - traces are used in the code generation phase

The Hotspot JVM



Class file creation

Continuous (Just In Time) compilation

Viewing Code Compilation

```
java -XX:-TieredCompilation -XX:+PrintCompilation HelloWorld 2> /dev/null
```

Time Offset	Task		Method Name (size of compiled code)
321	40		sun.nio.cs.StreamEncoder::isOpen (5 bytes)
322	41		sun.nio.cs.StreamEncoder::implFlushBuffer (15 bytes)
327	42		sun.nio.cs.StreamEncoder::writeBytes (132 bytes)
331	43	!	java.io.PrintStream::write (69 bytes)
335	44	s	java.io.BufferedOutputStream::write (67 bytes)
337	46		java.nio.Buffer::clear (20 bytes)
337	47		java.lang.String::indexOf (7 bytes)
338	48	!	java.io.PrintStream::println (24 bytes)
338	49		java.io.PrintStream::print (13 bytes)
343	50	!	java.io.PrintStream::write (83 bytes)
346	51	!	java.io.PrintStream::newLine (73 bytes)
347	52		java.io.BufferedWriter::newLine (9 bytes)
347	53	%	HelloWorld::main @ 2 (23 bytes)

- `!` method has exception handler(s)
- `s` method declared `synchronized`
- `n` native method (no compilation, generate wrapper)
- `%` on-stack replacement used

Inlining

- Calling a method has an overhead
  - creation of a new stack frame
  - copying values required to the stack frame
  - returning from the stack frame post execution
- Consider a method call in a for loop

```
public class HelloWorld {
       public static void main(String[] args) {
                for(int i=0; i < 100_000; i++) {
                        System.err.println("Hello World");
                }
        }
}
```

Inlining

```
java -XX:+UnlockDiagnosticVMOptions  -XX:+PrintInlining  HelloWorld 2> /dev/null
```

```
 @ 40   java.io.BufferedOutputStream::flush (12 bytes)   inline (hot)
                  \-> TypeProfile (19272/19272 counts) = java/io/BufferedOutputStream
                   @ 1   java.io.BufferedOutputStream::flushBuffer (29 bytes)   inline (hot)
                     @ 20   java.io.FileOutputStream::write (12 bytes)   inline (hot)
                      \-> TypeProfile (4696/4696 counts) = java/io/FileOutputStream
                       @ 8   java.io.FileOutputStream::writeBytes (0 bytes)   native method
                   @ 8   java.io.OutputStream::flush (1 bytes)   inline (hot)
                    \-> TypeProfile (7047/7047 counts) = java/io/FileOutputStream
!m             @ 13   java.io.PrintStream::println (24 bytes)
                 @ 6   java.io.PrintStream::print (13 bytes)
!m               @ 9   java.io.PrintStream::write (83 bytes)   callee is too large
!m             @ 10   java.io.PrintStream::newLine (73 bytes)   callee is too large
!m             @ 13   java.io.PrintStream::println (24 bytes)
                 @ 6   java.io.PrintStream::print (13 bytes)
!m               @ 9   java.io.PrintStream::write (83 bytes)   callee is too large
!m             @ 10   java.io.PrintStream::newLine (73 bytes)   callee is too large
!m             @ 13   java.io.PrintStream::println (24 bytes)   already compiled into a big method
```

Constant Subexpression Elimination

- Compiler hunts through code for common expressions
  - if results analyses replacement with a single variable
- Relies on data flow analysis of the program
  - which is done during the profiling and tracing part

Dead Code Elimination

- Removes code that is never executed
  - shrinks the size of the program
  - avoid executing irrelevant operations
- Dynamic dead code elimination
  - eliminated base on possible set of values
  - determined at runtime

Register Allocation

- Identification of variables suitable for registers
  - to avoid cache misses
  - improve execution speed of the program
- Uses data from the trace to make informed decision

Loop-Invariant Code Motion

- Involves removal of code from loops
  - for code that doesn't impact the outcome of the loop
  - moved above the loop to avoid unnecessary execution
- Hoisted code can now be cached in a register
  - improving performance of the loop execution

Escape Analysis

- Introduced in later versions of Java 6
- Analyses code to assert if an object reference
  - returns or leaves the scope of the method
  - stored in global variables
- Allocates unescaped objects on the stack
  - avoids the cost of garbage collection
  - prevents workload pressures on Eden
  - beneficial effects to counter high infant mortality GC impact

Loop Unrolling

```
private static final String[] RESPONSES =
                              { "Yes", "No", "Maybe" };

public void processResponses () {
  for ( String response: RESPONSES ) {
    process(response);
  }
}
```

```
private static final String[] RESPONSES =
                              { "Yes", "No", "Maybe" };

public void processResponses () {
  process(RESPONSES[0]);
  process(RESPONSES[1]);
  process(RESPONSES[2]);
}
```

Loop Unrolling

```
@Benchmark
public long intStride1()
{
        long sum = 0;
        for (int i = 0; i < MAX; i++)
        {
                sum += data[i];
        }
        return sum;
}

@Benchmark
public long longStride1()
{
        long sum = 0;
        for (long l = 0; l < MAX; l++)
        {
                sum += data[(int) l];
        }
        return sum;
}
```

Benchmark	Mode	Cnt	Score	Error	Units
LoopUnrollingCounter.intStride1	thrpt	200	**2423.818**	± 2.547	ops/s
LoopUnrollingCounter.longStride1	thrpt	200	**1469.833**	± 0.721	ops/s

Excerpt From: Benjamin J. Evans, James Gough, and Chris Newland. “Optimizing Java.” iBooks.

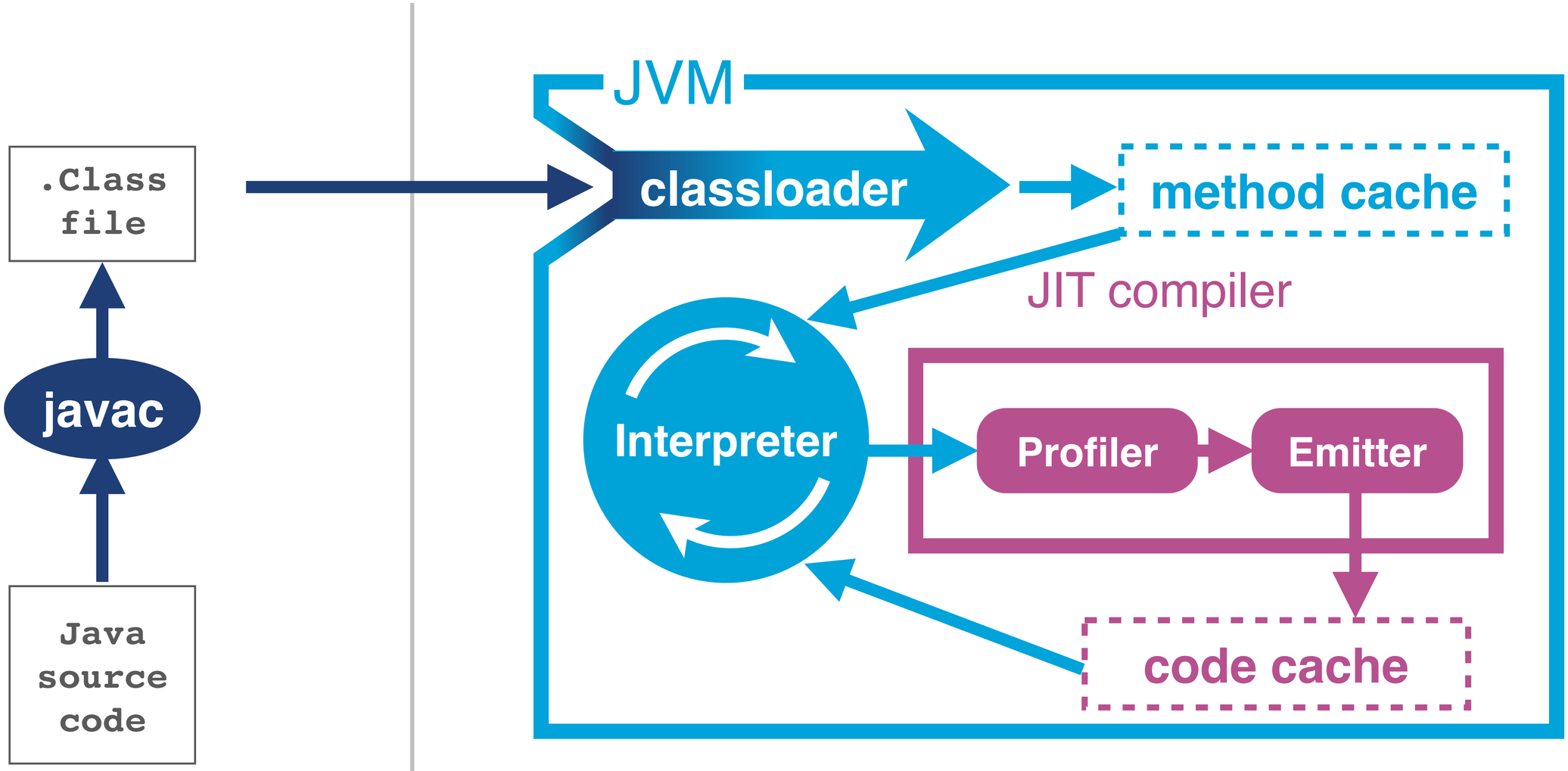
Loop Unrolling

- Can unroll int, char and short loops
- Can remove safe point checks
- Removes back branches and branch prediction cost
- Reduces the work needed by each “iteration”

Monomorphic Dispatch

- When HotSpot encounters a virtual call site, often only one type will ever be seen there
  - e.g. There's only one implementing class for an interface
- Hotspot can optimize vtable lookup
  - Subclasses have the same vtable structure as their parent
  - Hotspot can collapse the child into the parent
- Classloading tricks can invalidate monomorphic dispatch
  - The class word in the header is checked
  - If changed then this optimisation is backed out

Code Cache



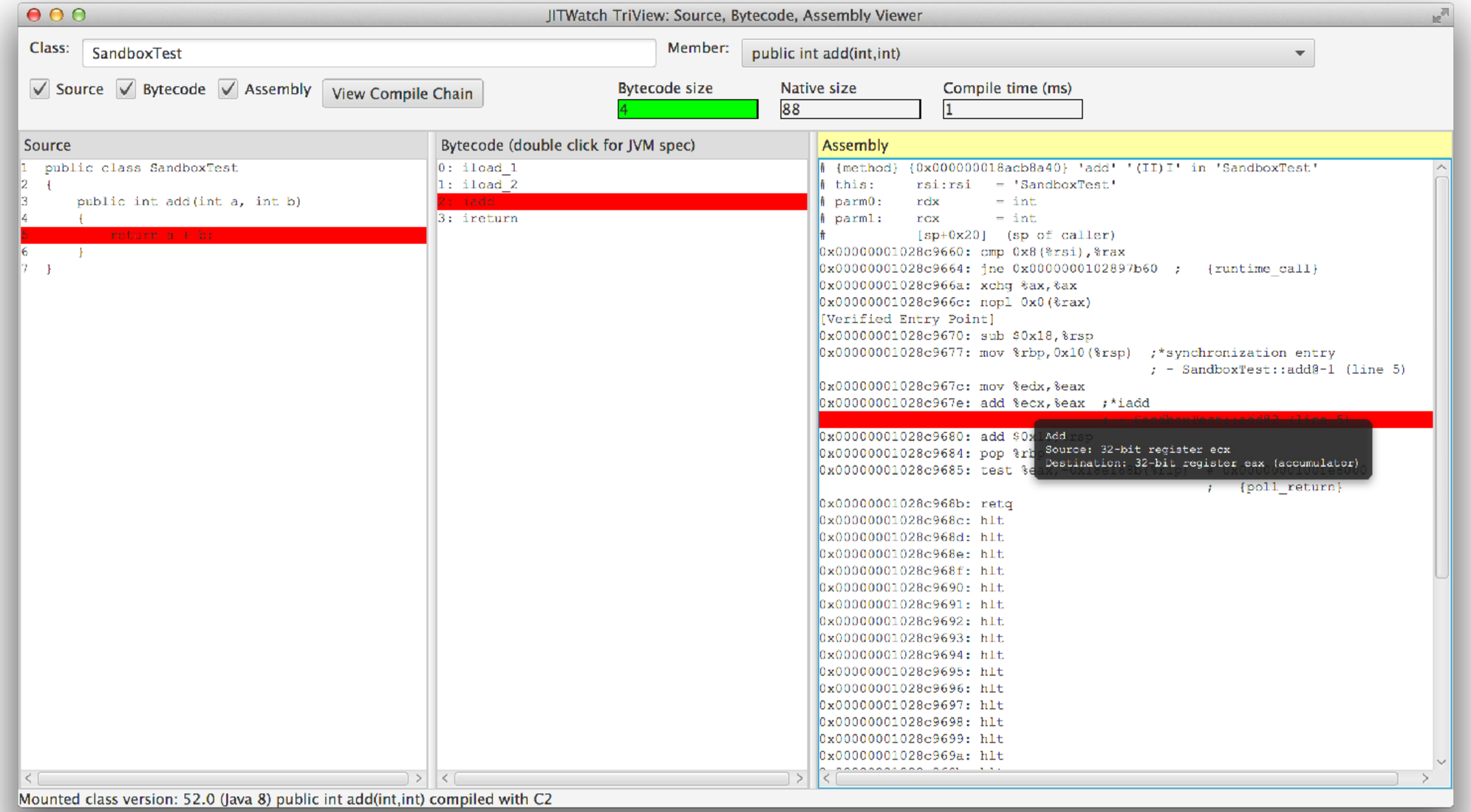
Class file creation

Continuous (Just In Time) compilation

Code Cache

- The code cache contains the JIT native compiled code
- Code is JIT'd on a per method basis
  - 1. This occurs when an entry counter is exceeded
  - 2. Internal Representation (IR) is built
  - 3. Optimisations are applied
  - 4. JIT turns IR into native code
- Pointers are swizzled to use the native code
  - native code is executed on the next call

Introduction to JITWatch



Summary

- Java has carried a brand name of being slow
- Java can emit instructions comparable to C++
- javac doesn’t do much optimisation
- We can make better decisions from profiling at runtime
- JITWatch makes life easier

Performance Landscape