

# “Just In Time” to understand

(An introduction to how *JIT compilers* work under the hood)

# Why this topic?

I am really curious about computer works under the hood. My curiosity lead me to try and find out how the JIT compilers function.

- How is the code compiled during the run-time?
- How is it possible to compile code into the memory and then execute it?
- Are JIT compilers horrible monsters? (Spoiler: *maybe not* :D)

I hope these questions will be answered during the presentation.

“Great code is efficient code. But before you can write truly efficient code, you must understand how computer systems execute programs and how abstractions in programming languages map to the machine's low-level hardware.” - **Randall Hyde** (*Write Great Code, No Starch Press*)

# Minimum Requirements

To fully understand the presentation, you will need a *bit* of knowledge about:

1. *Computer Architecture.*
2. *Operating System.*
3. *C++ language.*
4. *JVM Instruction Set.*
5. *Interpreters.*

# Outline

1. Brief history of **Just In Time** compilation
2. **What** is a **JIT compiler**?
3. **Where** is it used?
4. **How** does it work? A look at HotSpot JVM
5. (*Tiny*) **C++ implementation** of a JIT compiler
6. Conclusions

# Outline

1. Brief history of **Just In Time** compilation
2. **What** is a **JIT** compiler?
3. **Where** is it used?
4. **How** does it work? A look at HotSpot JVM
5. (*Tiny*) **C++ implementation** of a JIT compiler
6. Conclusions

# Origins of JIT Compilers

- The first signs of JIT compilers date back to the 1960, from the LISP's creator John McCarthy.
- Another ancestor of initial just-in-time compilers is the regular expression compiler created by Ken Thompson in 1968, which converted regexes to IBM 7094 CPU native code.
- In the '70 and '80, the calculation power of computers was unlike today's. Whoever had an interpreted program and wanted to speed it up, a “**mixed-code**” approach was the best idea.
- Mixed-code approach was good, however maintaining pieces of program, write in native code and other pieces in interpreted code, was not an easy task.

# Origins of JIT Compilers (cont'd)

- From a mixed-code approach, developers moved to "**throw-away**" compiling.
- Program pieces were compiled dynamically according to the necessities. When the memory was about to run out, then the compiled code would be thrown away.
- At the end of the '90, **Java** was born. Initially, the Java Virtual Machine was really *inefficient*.
- **Sun Microsystem** developed a Just-in-time compiler to boost Java performances.

# Outline

1. Brief history of **Just In Time** compilation
2. **What is a JIT compiler?**
3. **Where** is it used?
4. **How** does it work? A look at HotSpot JVM
5. (*"Tiny"*) **C++ implementation** of a JIT compiler
6. Conclusions



# What is “Just in Time” compilation?

**Just-in-Time compilation** is a dynamic translation technique that compiles running code during the execution of a program at runtime rather than before execution.

JIT compilation is a **merge** between **ahead-of-time compilation** and **interpretation**.

# Benefits and Drawbacks of a JIT compiler

As a combination of these two traditional approaches, the JIT compilation brings **advantages** and **drawbacks** of **both**.

## Compilation to Native Code

### Advantages

1. **Blazing fast and efficient**
2. Developers can interact directly with the underlay hardware

### Disadvantages

1. **Poor portability** due to CPU architecture
2. Large program size

## Just-in-Time



- Some JIT compilers need time to startup.
- + JIT compilers know a lot of information about the program during run-time

## Interpretation

### Advantages

1. Greater Portability
2. Small program size
3. **Knows a lot of information during run-time**

### Disadvantages

1. **Really slow** because of interpreter

# Outline

1. Brief history of **Just In Time** compilation
2. **What** is a **JIT compiler**?
3. **Where** is it used?
4. **How** does it work? A look at HotSpot JVM
5. (*Tiny*) **C++ implementation** of a JIT compiler
6. Conclusions

# Where it is used?

- Most **browsers** today, use JIT compilers to enhance performances of web pages and applications. Browser engines compile JavaScript code in native one.
- Other than browsers, programs written in Python, running on **PyPy**, “may” also gain performance boost.

Apple's SquirrelFish



Google's V8



Mozilla's SpiderMonkey



## Where it is used? (cont'd)

- JIT compilation is used inside the **Linux** Kernel for network packet filtering (see **eBPF**).
- Even **Android** use JIT compilation to run its applications!
- Furthermore, in the video games **emulation** scene, JIT compilers are used to enhance the performances (basically, a console ISA is translated to the CPU host ISA).



Android



Dolphin Emulator



**They are used everywhere,  
even in your pockets!**

# Outline

1. Brief history of **Just In Time** compilation
2. **What** is a **JIT compiler**?
3. **Where** is it used?
4. **How** does it work? A look at HotSpot JVM
5. (*Tiny*) **C++ implementation** of a JIT compiler
6. Conclusions

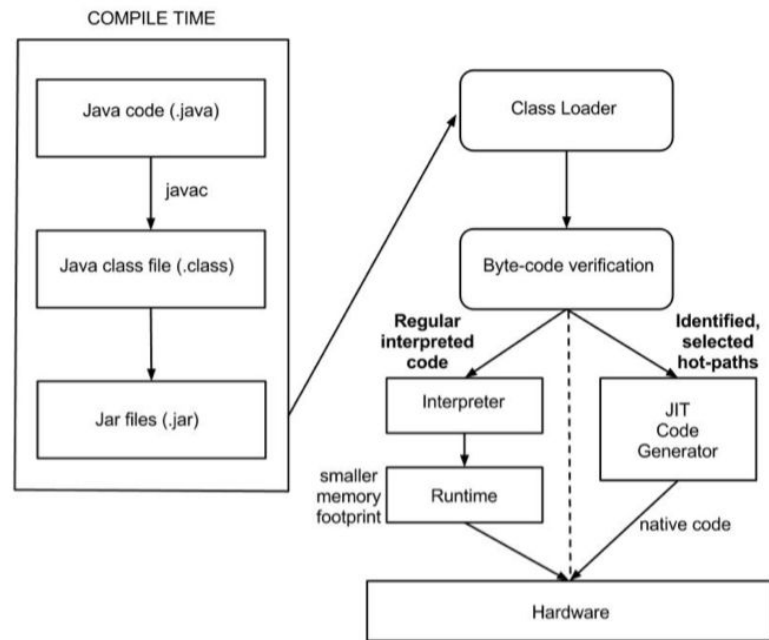
# How a JIT compiler works

- In JIT compilation process, starting with the interpreter, some features of a static compiler are built into the system.
- A JIT compiler *will isolate some sections of the code at run-time which are accessed more often.*
- Then it will *compiles them to native code*, **aggressively** optimizing those sections in the process.



# HotSpot JVM

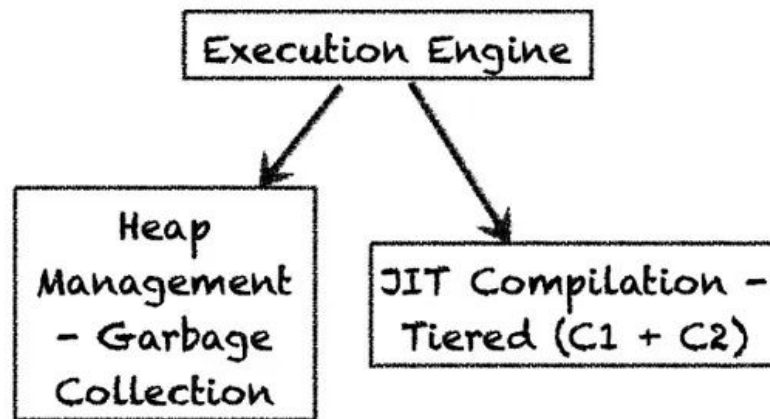
- **HotSpot JVM** is the default interpreter of Java.
- The virtual machine is equipped with a JIT compiler.
- HotSpot practice “**trace-JIT**” compilation.
- Frequently used methods inside Java programs will be compiled in native code.
- The methods compiled in machine code are called **hot methods** 🔥.



# HotSpot JVM (cont'd)

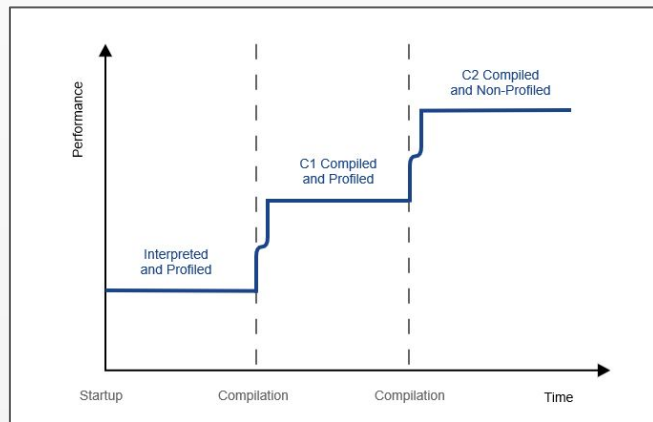
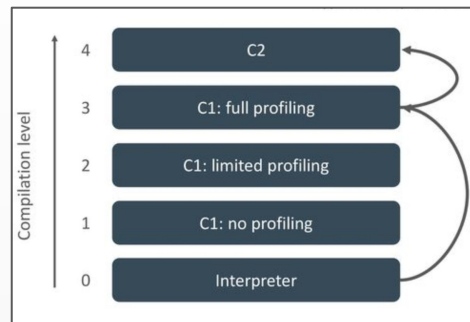
HotSpot has **two main JIT compilers** that are executed according to established thresholds:

1. the **Client compiler**, or **C1**, has a low threshold ( $\approx 1.500$  method calls), this is used to reduce startup time.
2. the **Server compiler**, or **C2**, has a bigger threshold ( $\approx 10.000$  method calls) and it generates efficient optimized code for critical methods.



# HotSpot's Tiered Compilation

- HotSpot JVM comes with a “**tiered-compilation mode**”.
- At the startup, the JVM interprets the bytecode and monitors it to get profiling information about the execution path.
- Firstly **C1**, will be executed to compile the bytecode into machine code to reach native performance.
- After collecting other informations, **C2** will re-compe all the code optimizing it.
- Finally, during the execution the **deoptimization** phase may happen.



# JITWatch

It is possible to monitor the HotSpot JIT compilers. **JITWatch** is a tool for understanding the behaviour of the Java HotSpot Just-In-Time (JIT) compilers during execution of a Java program.

Source	Bytecode (double click for JVM spec)	Assembly <input checked="" type="checkbox"/> Labels #3 (C2 / OSR / Level 4)
<pre> 1 // The Sandbox is designed to help you learn about the HotSpot JIT compilers. 2 // Please note that the JIT compilers may behave differently when isolating a 3 // in the Sandbox compared to running your whole application. 4 5 public class SimpleInliningTest 6 { 7     public SimpleInliningTest() 8     { 9         int sum = 0; 10 11         // 1_000_000 is F4240 in hex 12         for (int i = 0 ; i &lt; 1_000_000; i++) 13         { 14             sum = this.add(sum, 99); // 63 hex 15         } 16 17         System.out.println("Sum:" + sum); 18     } 19 20     public int add(int a, int b) 21     { 22         return a + b; 23     } 24 25     public static void main(String[] args) 26     { 27         new SimpleInliningTest(); 28     } 29 } </pre>	<pre> 0: aload_0 1: invokespecial   #1 // Method java/lang/Object.&lt;init&gt;:()V 4: iconst_0 5: istore_1 6: iconst_0 7: istore_2 8: iload_2 9: ldc             #7 // int 1000000 11: if_icmpge      28 14: aload_0 15: load_1 16: bipush        99 18: invokevirtual   #8 // Method add:(II)I 21: istore_1 22: linc           2, 1 25: goto          8 28: getstatic      #14 // Field java/lang/System.out:Ljava/io/PrintStream; 31: load_1 32: invokedynamic  #20, 0// InvokeDynamic 0:makeConcatWithConstants 37: invokevirtual   #24 // Method java/io/PrintStream.println:(Ljava/lang/String;)V 40: return </pre>	<pre> # [method: {0x0000000136000388} '&lt;init&gt;' '()'V in 'SimpleInliningTest' [Entry Point] 0x00000001266bf4a0: e82b 611c ; - SimpleInliningTest::&lt;init&gt;@8 (line 12) 0x00000001266bf4c0: 8b5e 0844 ; - SimpleInliningTest::&lt;init&gt;@11 (line 12) 0x00000001266bf4d4: ffd2 45b8 ; - SimpleInliningTest::&lt;init&gt;@8 (line 12) 0x00000001266bf4e0: 4181 fb00 ; - SimpleInliningTest::&lt;init&gt;@8 (line 12) 0x00000001266bf4f8: 0000 456b ; - SimpleInliningTest::&lt;init&gt;@8 (line 12) 0x00000001266bf500: db45 2bda ; *ilnc (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf520: 418b e841 0x00000001266bf524: ; *ilnc (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf524: 03ed 6bdd 0x00000001266bf528: ; - SimpleInliningTest::&lt;init&gt;@22 (line 12) 0x00000001266bf528: 03ed 6bdd 0x00000001266bf528: ; *iadd (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf528: 03ed 6bdd 0x00000001266bf528: ; - SimpleInliningTest::add@2 (line 22) 0x00000001266bf528: 03ed 6bdd 0x00000001266bf528: ; - SimpleInliningTest::add@2 (line 22) 0x00000001266bf528: 6341 03db ; - SimpleInliningTest::&lt;init&gt;@11 (line 12) 0x00000001266bf530: 0f00 7c34 ; *iadd (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf540: 0866 90e8 0x00000001266bf544: ; *ilnc (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf544: b88a 58f8 0x00000001266bf548: ; *iadd (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf548: b88a 58f8 0x00000001266bf548: ; - SimpleInliningTest::add@2 (line 22) 0x00000001266bf548: b88a 58f8 0x00000001266bf548: ; - SimpleInliningTest::&lt;init&gt;@18 (line 14) 0x00000001266bf548: 83c3 6341 ; *ilnc (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf560: 6666 90e8 0x00000001266bf564: ; *ilnc (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf564: 988a 58f8 ; - SimpleInliningTest::add@2 (line 22) 0x00000001266bf564: 988a 58f8 ; - SimpleInliningTest::&lt;init&gt;@8 (line 12) 0x00000001266bf574: 63bd 4042 ; - SimpleInliningTest::&lt;init&gt;@18 (line 14) 0x00000001266bf58c: 2466 90e8 0x00000001266bf590: ; *ilnc (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf590: 6c8a 58f8 ; *ilnc (reexecute=0 rethrow=0 return_oop=0) 0x00000001266bf590: 6c8a 58f8 ; - SimpleInliningTest::&lt;init&gt;@8 (line 12) [ExceptionHandler] 0x00000001266bf5a0: e95b a962 0x00000001266bf5ac: 2c24 05e9 [/MachCode] </pre>

# So, why don't use an AOT compiler instead?

Well... That is a good question!

Java developers introduced an experimental AOT compiler in **JDK 9**

(see [JEP 295: Ahead-of-Time Compilation](#))

But...

## JEP 295: Ahead-of-Time Compilation

<i>Owner</i>	Vladimir Kozlov
<i>Type</i>	Feature
<i>Scope</i>	Implementation
<i>Status</i>	Closed / Delivered
<i>Release</i>	9
<i>Component</i>	hotspot / compiler
<i>Discussion</i>	hotspot dash compiler dash dev at openjdk dot java dot net
<i>Effort</i>	M
<i>Duration</i>	M
<i>Reviewed by</i>	John Rose, Mikael Vidstedt
<i>Endorsed by</i>	John Rose
<i>Created</i>	2016/09/15 01:20
<i>Updated</i>	2018/10/05 22:52
<i>Issue</i>	8166089

### Summary

Compile Java classes to native code prior to launching the virtual machine.

### Goals

- Improve the start-up time of both small and large Java applications, with at most a limited impact on peak performance.
- Change the end user's work flow as little as possible.

### Non-Goals

It is not necessary to provide an explicit, exposed library-like mechanism for saving and loading compiled code.

### Motivation

JIT compilers are fast, but Java programs can become so large that it takes a long time for the JIT to warm up completely. Infrequently-used Java methods might never be compiled at all, potentially incurring a performance penalty due to repeated interpreted invocations.

# So, why don't use an AOT compiler instead? (cont'd)

... since developers saw a little use of this compiler, and, seeing as the amount of work to maintain it was **pretty huge**\*, they decided to **remove it!** (see <https://openjdk.java.net/jeps/410>)

(\*) Just think all the CPU architectures out of here: x86\_64, ARM, MIPS, *RISC-V*, *PowerPC* (💀) and so many others...

## JEP 410: Remove the Experimental AOT and JIT Compiler

Owner	Vladimir Kozlov
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	17
Component	hotspot / compiler
Discussion	hotspot dash compiler dash dev at openjdk dot java dot net
Effort	S
Duration	S
Reviewed by	Mikael Vidstedt
Created	2021/03/10 02:59
Updated	2021/08/05 02:44
Issue	8263327

### Summary

Remove the experimental Java-based ahead-of-time (AOT) and just-in-time (JIT) compiler. This compiler has seen little use since its introduction and the effort required to maintain it is significant. Retain the experimental java-level JVM compiler interface (JVMCI) so that developers can continue to use externally-built versions of the compiler for JIT compilation.

### Motivation

Ahead-of-time compilation (the jaotc tool) was incorporated into JDK 9 as an experimental feature via JEP 295. The jaotc tool uses the Graal compiler, which is itself written in Java, for AOT compilation.

The Graal compiler was made available as an experimental JIT compiler in JDK 10 via JEP 317.

We have seen little use of these experimental features since they were introduced, and the effort required to maintain and enhance them is significant. These features were not included in the JDK 16 builds published by Oracle, and no one complained.

and no one complained.



# Outline

1. Brief history of **Just In Time** compilation
2. **What** is a **JIT compiler**?
3. **Where** is it used?
4. **How** does it work? A look at HotSpot JVM
5. (*Tiny*) **C++ implementation** of a JIT compiler
6. Conclusions



# Disclaimer!

The next slides will show C++ and Assembly code.

The code implements an interpreter which evaluates a small (a really small one) subset of JVM instructions, specifically, the ones about integer operations. Of course the code is only used for didactic purposes; real JIT compilers are much **more complex** (and **surely more efficient, more memory-safe**) than this one, so be aware for it!

All the JVM instructions can be found here:

<https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html>

# Under the hood: Code (1)

```

1 int main(int argc, char** argv) {
2
3     auto env = std::map<int, int>{{VAR_X, 3}};
4
5     auto squareFun = std::vector<std::string> {
6         "iload_0",
7         "iload_0",
8         "imul",
9         "ireturn"
10    };
11
12    auto jit_compiler = JITCompiler{env};
13    auto jvm_interpreter = JVMInterpreter{env};
14
15    ASTNode* tree = JVMParser::parse_from_bytecode(squareFun);
16    JITFunction fun = jit_compiler.compile_jvm_function(tree);
17
18    jit_compiler.dump_assembly();
19
20    std::cout << "Interpreted:\t" << jvm_interpreter.interpret(tree) << "\n";
21    std::cout << "Compiled:\t\t" << fun();
22
23    return 0;
24 }

```

```

1 public class Main {
2
3     public static int square(int x) {
4         return x * x;
5     }
6
7 }

```

The code shown above is a Java function used to compute the square of a number. The compiled JVM bytecode version can be obtained using the `javap` utility.

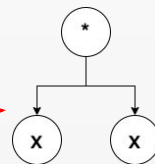
# Under the hood: Code (2)

```

1 int main(int argc, char** argv) {
2
3     auto env = std::map<int, int>{{VAR_X, 3}};
4
5     auto squareFun = std::vector<std::string> {
6         "iload_0",
7         "iload_0",
8         "imul",
9         "ireturn"
10    };
11
12    auto jit_compiler = JITCompiler{env};
13    auto jvm_interpreter = JVMInterpreter{env};
14
15    ASTNode* tree = JVMParser::parse_from_bytecode(squareFun); }
16    JITFunction fun = jit_compiler.compile_jvm_function(tree);
17
18    jit_compiler.dump_assembly();
19
20    std::cout << "Interpreted:\t" << jvm_interpreter.interpret(tree) << "\n";
21    std::cout << "Compiled:\t\t" << fun();
22
23    return 0;
24 }

```

The **JVMParser** class transform the bytecode into an **Abstract Syntax Tree** (like the one shown below). This code representation is useful for both interpretation and compilation of this example.



This is the Abstract Syntax Tree representing the JVM bytecode. The leafs refers to function parameters, while the root node is a multiplicative operation between x and x.

# Under the hood: Compiler's Core

## Compiler's core.

Here is where the *magic* happens.

Next slides will *explain* the **three** labelled blocks.

```

1 typedef int(*JITFunction)();
2
3 JITFunction compile_jvm_function(ASTNode* tree) {
4
5     assembly.clear();
6
7     auto memory = static_cast<uint8_t*>(mmap(nullptr, 1024,
8                                     PROT_READ | PROT_WRITE | PROT_EXEC,
9                                     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0));
10
11     if (memory == MAP_FAILED) {
12         throw std::runtime_error{"Cannot allocate memory for the compiled function!"};
13     }
14
15     // push rbp
16     assembly.push_back(0x55);
17     // mov rbp, rsp
18     assembly.push_back(0x48); assembly.push_back(0x89); assembly.push_back(0xe5);
19
20     // Compile the AST to assembly code
21     aux_compile(tree);
22
23     // pop rbp
24     assembly.push_back(0x5d);
25     // ret
26     assembly.push_back(0xc3);
27
28     // Copy instructions inside the function
29     for (std::size_t i = 0; i < assembly.size(); i++) memory[i] = assembly[i];
30
31     return reinterpret_cast<JITFunction>(memory);
32 }

```

1

2

3

# Under the hood: Compiler's Core (1)

The first piece of the body asks to the operating system to reserve 1KB of memory inside the **heap** using **mmap** *syscall*. In this area of memory we are going to write our compiled function.

We cannot use the standard **malloc** function because we have to set some flags about the allocated memory.

These flags allow us to tell to the OS the desired memory protections. Specifically, we want that our memory can be **readable**, **writable** (risky flag) and, the most important one, **executable**.

Most of **browser exploits** are due to how JIT compilers use this memory! The attacker could write inside the memory arbitrary code! For more information see: [JIT Spraying](#).

```
auto memory = static_cast<uint8_t*>(mmap(nullptr, 1024,  
                                           PROT_READ | PROT_WRITE | PROT_EXEC,  
                                           MAP_PRIVATE | MAP_ANONYMOUS, -1, 0));  
  
if (memory == MAP_FAILED) {  
    throw std::runtime_error{"Cannot allocate memory for the compiled function!"};  
}
```

# Under the hood: Compiler's Core (2)

The second body piece writes the **assembly code** into a vector of bytes (`uint8_t`).

The first and the latter parts are standard **x86\_64** instructions used to create a new stack frame for the function's execution.

(Since my computer uses an *Intel i7*, I wrote **x86\_64** instructions, on **ARM/RISC-V** processor the code will not work).

The middle part, where the **aux\_compile** function is invoked, uses the AST showed before to produce assembly instructions according to the tree structure.

Finally, copy the compiled assembly instructions contained inside the **assembly** vector into the new allocated memory pointed by the **memory** pointer.

```
std::vector<std::uint8_t> assembly;
```

```
// push rbp
assembly.push_back(0x55);
// mov rbp, rsp
assembly.push_back(0x48); assembly.push_back(0x89); assembly.push_back(0xe5);

// Compile the AST to assembly code
aux_compile(tree);

// pop rbp
assembly.push_back(0x5d);
// ret
assembly.push_back(0xc3);

// Copy instructions inside the function
for (std::size_t i = 0; i < assembly.size(); i++) memory[i] = assembly[i];
```

# Under the hood: Compiler's Core (3)

The last body piece casts the pointer to `uint8_t` to a function pointer!

The function pointer has a definition like this one: `int(*JITFunction)()`

The cast is the real deal we were looking for. Basically, this operation will allow the program to call the compiled function during the run-time, resulting in the function execution.

Since our example compiles and computes only integer numbers the function will return an integer.

```
return reinterpret_cast<JITFunction>(memory);
```

# Under the hood: Inspecting Call (1)

```
1 call    compile_jvm_function()
2 mov     qword ptr [rbp - 8], rax
3 call    qword ptr [rbp - 8]
```

1. After the compilation, the fun variable contains a pointer to the allocated function.

```
1 int main(int argc, char** argv) {
2
3     auto env = std::map<int, int>{{VAR_X, 3}};
4
5     auto squareFun = std::vector<std::string> {
6         "iload_0",
7         "iload_0",
8         "imul",
9         "ireturn"
10    };
11
12    auto jit_compiler = JITCompiler{env};
13    auto jvm_interpreter = JVMInterpreter{env};
14
15    ASTNode* tree = JVMParser::parse_from_bytecode(squareFun);
16    JITFunction fun = jit_compiler.compile_jvm_function(tree);
17
18    jit_compiler.dump_assembly();
19
20    std::cout << "Interpreted:\t" << jvm_interpreter.interpret(tree) << "\n";
21    std::cout << "Compiled:\t\t" << fun();
22
23    return 0;
24 }
```



# Under the hood: Inspecting Call (2)

```

1 call    compile_jvm_function()
2 mov     qword ptr [rbp - 8], rax
3 call    qword ptr [rbp - 8]

```

```

19      std::cout << "Compiled:\t\t" << fun();

```

2. When the CPU will execute the `call` instruction, the Program Counter will be updated with the value saved inside the stack. This memory address points to the allocated memory of previously compiled function.

Instruction		Description
<b>call</b>	<i>Label</i>	Push return address and jump to label
<b>call</b>	<i>*Operand</i>	Push return address and jump to specified location

# Under the hood: Output

Once the function is compiled inside the program's memory we can invoke it! This is the result:

```
Compiled code: 55 48 89 e5 c7 45 fc 03 00 00 00 c7 45 f8 03 00 00 00 8b 45 fc 0f af 45 f8 89 45 f8 5d c3
Interpreted:    9
Compiled:       9
Process finished with exit code 0
```

Console Output

Compiled code in x86\_64

```
0: 55          push    rbp
1: 48 89 e5     mov     rbp, rsp
4: c7 45 fc 03 00 00 00 mov     dword ptr [rbp - 4], 3
b: c7 45 f8 03 00 00 00 mov     dword ptr [rbp - 8], 3
12: 8b 45 fc     mov     eax, dword ptr [rbp - 4]
15: 0f af 45 f8  imul    eax, dword ptr [rbp - 8]
19: 89 45 f8     mov     dword ptr [rbp - 8], eax
1c: 5d          pop     rbp
1d: c3          ret
```

# A Real JIT Compiler

- If you would like to see how a **real JIT** compiler is implemented, see **LuaJIT**.
- The compiler works for the **Lua** programming language and it is used in a lot of applications.
- For more details, see the “**LuaJIT Project**” at <https://luajit.org>.

## LuaJIT

Home

LuaJIT

Download ↴

Installation

Running

Extensions

FFI Library

ffi.\* API

FFI Semantics

jit.\* Library

Lua/C API

Status

FAQ

Performance

on x86/x64

on ARM

on PPC

on PPC/e500

on MIPS

Wiki »

Mailing List

Sponsors

## LuaJIT

LuaJIT is a **Just-In-Time Compiler (JIT)** for the **Lua** programming language. Lua is a powerful, dynamic and light-weight programming language. It may be embedded or used as a general-purpose, stand-alone language.

LuaJIT is Copyright © 2005-2021 Mike Pall, released under the **MIT open source license**.

### Compatibility

Windows	Linux	BSD	macOS	POSIX
FreeBSD	Android	iOS		
PS3	PS4	PS Vita	Xbox 360	
GCC	CLANG LLVM	MSVC		
x86	x64	ARM	PPC	e500
MIPS				
Lua 5.1 API+ABI	+ JIT	+ BitOp	+ FFI	Drop-in DLL/.so

### Overview

3x - 100x	115x VM	90x JIT	63x C	24x ASM	11x Lua
--------------	------------	------------	----------	------------	------------

LuaJIT has been successfully used as a **scripting middleware** in games, appliances, network and graphics apps, numerical simulations, trading platforms and many other specialty applications. It scales from embedded devices, smartphones, desktops up to server farms. It combines high flexibility with high performance and an unmatched low memory footprint.

LuaJIT has been in continuous development since 2005. It's widely considered to be one of the **fastest dynamic language implementations**. It has outperformed other dynamic languages on many cross-language benchmarks since its first release — often by a substantial margin.

For **LuaJIT 2.0**, the whole VM has been rewritten from the ground up and relentlessly optimized for performance. It combines a **high-speed interpreter**, written in assembler, with a **state-of-the-art JIT compiler**.

An innovative **trace compiler** is integrated with advanced, SSA-based optimizations and highly tuned code generation backends. A substantial reduction of the overhead associated with dynamic languages allows it to break into the performance range traditionally reserved for offline, static language compilers.

# Outline

1. Brief history of **Just In Time** compilation
2. **What** is a **JIT compiler**?
3. **Where** is it used?
4. **How** does it work? A look at HotSpot JVM
5. (*Tiny*) **C++ implementation** of a JIT compiler
6. **Conclusions**

# Conclusions

We saw what JIT compilers are, *how they are built* (conceptually speaking) and *where they are used*. There is a lot of content that wasn't shown in slides, I will leave some resources and references I used for this presentation.

# Resources

- A brief history of Just In Time by *John Aycock*
- JIT through the ages by *Neeraja Ramanan*
- The Java HotSpot VM by Tobias Hartmann  
([https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/ist-dam/documents/Education/Classes/Spring2018/210\\_Compiler\\_Design/Slides/2018-Compiler-Design-Guest-Talk.pdf](https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/ist-dam/documents/Education/Classes/Spring2018/210_Compiler_Design/Slides/2018-Compiler-Design-Guest-Talk.pdf))
- Understanding Java JIT Compilation with JIT Watch  
(<https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html>)
- How the JIT compiler boosts Java performance in OpenJDK  
(<https://developers.redhat.com/articles/2021/06/23/how-jit-compiler-boosts-java-performance-openjdk>)
- JVM JIT-compiler overview  
([http://cr.openjdk.java.net/~vlivanov/talks/2015\\_JIT\\_Overview.pdf](http://cr.openjdk.java.net/~vlivanov/talks/2015_JIT_Overview.pdf))
- Just in Time Compilation Explained  
(<https://www.freecodecamp.org/news/just-in-time-compilation-explained/>)
- What the JIT!? Anatomy of the OpenJDK HotSpot JVM  
(<https://www.infoq.com/articles/OpenJDK-HotSpot-What-the-JIT/>)
- Deep Dive Into the New Java JIT Compiler - Graal  
(<https://www.baeldung.com/graal-java-jit-compiler>)
- How to write a JIT Compiler  
(<https://github.com/spencertipping/jit-tutorial>)
- Adventures in JIT compilation:  
<https://eli.thegreenplace.net/2017/adventures-in-jit-compilation-part-1-an-interpreter/>
- Writing a minimal x86-64 JIT compiler in C++  
(<https://solarianprogrammer.com/2018/01/10/writing-minimal-x86-64-jit-compiler-cpp/>)
- Just-in-time compilation  
([https://en.wikipedia.org/wiki/Just-in-time\\_compilation](https://en.wikipedia.org/wiki/Just-in-time_compilation))
- How JIT Compilers are implemented and Fast: Pypy, LuaJIT, Graal and more (<https://carolchen.me/blog/technical/jits-impls/>)
- A deep introduction to JIT compilers: JITs are not very Just-in-time  
(<https://carolchen.me/blog/technical/jits-intro/>)
- OpenJDK Wiki  
(<https://wiki.openjdk.java.net/display/HotSpot/Compiler>)

# Thanks

I hope you enjoyed these topics and found them interesting.

I would like to thank professor **Andrea Corradini** for the opportunity he gave to me for this presentation. Furthermore, I would like to thank my colleagues and my friends for their support and feedback.