Just-In-Time (JIT) Compiler

1. Overview

The *Just-In-Time (JIT)* compiler, which commonly referred as *JIT Compiler*, is a part of *Java Runtime Environment (JRE)* that improves the performance of `Java` applications at runtime. In computing *JIT* compilation, which is also known as "Dynamic Translations" is compilation done during execution of a program.

Traditional `Java` compiler compiles high-level `Java` source code (*.java* files) to bytecode readable by *JVM* (*.class* files) and then *JVM* interprets bytecode to machine instructions at runtime. *JIT* compiler helps to improve the performance of Java programs by compiling bytecodes into native machine code at runtime. *JIT* compiler is used surpass the performance of static compilation and maintain advantages of bytecode interpretation.

2. JIT Compilation

JIT compilation is also referred as "Hotspot Compilation". Name "Hotspot" comes from the approach used by *JVM* to compile a code. One famous statement about program execution is "80% of execution time is spent in executing 20% of code". A main objective of optimization to optimize this 20% code which is used frequently in a program as performance heavily depends on such section of code. These critical sections are known as Hotspot.

There are 2 types of *JIT* compilers used to optimize, one is used for a client and another one is for a server. Programs, which are written to be on the server are more resource consuming and collect more accurate statistics. It is taking more time to find a best-optimized option. Client compiler took less time to start as they precompiled some source code and they are not producing as accurate result as server compiler does.

JIT compiler can use more than one compilation thread to optimize code faster and reduce start up delay.

2.1 Impact on Application Start-up

JIT compilation requires memory and processor time to optimize methods. When *JVM* initialize millions of methods which are being used in an application are called and it took significant start-up time to compile and optimize it.

In practice, all methods are not compiled for the very first time they are being called. *JVM* maintains a call count to identify highly used methods. It increments counts each time method is being called. *JVM* interprets a method until its count reaches a threshold for *JIT* compilation, once it reached that threshold, *JIT* compiler compile that method and make an optimization to improve the performance of an application.

Threshold value should be set very carefully to maintain the balance between high performance and less start up time.

2.2 Phases of Compilation

Compilation of *JIT* compiler consists five phase life cycle to generate optimized machine code:

Inlining: Merge small methods to inline calls to speed up frequently executed method calls.

Local optimization: Analyze small portion of code and optimize it to improve method's performance.

Control Flow Optimization: Analyze and rearrange control flow of method to execute it in better order.

Global Optimization: Analyze large portion of code to optimize methods. This phase is expensive in terms of memory and time usage but it provides a great level of optimization.

Native Code Generation: Only Step in this cycle which is platform dependent. During this phase code is translated to machine code with some optimizations are performed based on platform characteristics.

There are several ways to perform each of phases and there are several sub-phases for them too.

2.3 Impact of Disabling JIT Compiler

JIT compiler is enabled by default and it automatically gets activated when a `Java` method is called. It is not recommended to disable *JIT* compiler unless we need a workaround with some *JIT* compilation problems.

Disabling *JIT* compiler means entire `Java` program will be interpreted and then it might be a lot worse in performance than before.

2.4 Example of JIT Compilation in JRockit JVM

Let's look how *JVM* optimizes code to improve performance, for better understanding the code is written like `Java` program but *JVM* performs optimization on bytecode.

Code Before Optimization:

```
[java]
Class A {
    B b;
    public void test(){
        int x = b.retrieve();
        // more code
        int y = b.retrieve();
        int sum = x + y;
    }
}
Class B {
    int v;
    final int retrieve() {
        return v;
    }
}
[/java]
```

Let's look at the step by step process made by *JIT* compiler to optimize `test()` method:
1) Starting Point:

```
[java]
public void test(){
    int x = b.retrieve();
    // more code
    int y = b.retrieve();
    int sum = x + y;
}
[/java]
```

2) Inline Final Method:
To reduce latencies replace `b.retrieve()` with its contents.

```
[java]
public void test(){
    int x = b.val;
    // more code
    int y = b.val;
    int sum = x + y;
}
[/java]
```

3) Remove Redundant Calls:
We already called `retrieve()` once and assigned it to local variable `x`. We can reduce latencies by using that local variable with `y` instead of calling a function again.

```
[java]
public void test(){
    int x = b.val;
    // more code
    int y = x;
    int sum = x + y;
}
[/java]
```

4) Copy Propagation:
we know that variables `x` and `y` contains same value so we can use one variable and can eliminate an extra variable.

```
[java]
public void test(){
    int x = b.val;
    // more code
    x = x;
    int sum = x + x;
}
[/java]
```

5) Dead Code Elimination:
After performing other steps to optimize code, we will end up with some dead code which is no longer mean anything to our program. Those codes need to be removed to further optimize a program.

```
[java]
public void test(){
    int x = b.val;
    // more code
    int sum = x + y;
}
[/java]
```

Now let's see a code after performing optimization on it.

Code After Optimization:

```
[java]
Class A {
    B b;
    public void methodOne(){
        int x = b.retrieve();
        // more code
        int sum = x + x;
    }
}
```

```
Class B {
    int val;
    final int retrieve() {
        return val;
    }
}
[/java]
```

3. Drawback of JIT Compilation

- Limited ahead of time compilation consumes more memory and time at startup which is introducing Strat up delay for an application.
- JIT introduced one more complex layer in program execution which developers don't really understand and makes it more difficult to improve performance affected by other factors.

4. Final Words

The simplest tool used to increase the performance of `Java` application is *"Just-In-Time"* compiler. It improves the performance of an application significantly. *JIT* compiler will improve performance but introduce some delay in startup and can consume more memory too. But all in all, *JIT* is a brilliant component in `Java` Virtual Machine which makes our code to run faster.