



॥ सा विद्या या विमुक्तये ॥

JAVA*: Runtime Optimization in Java Virtual Machines

Course-seminar report for CS 304: Operating Systems, Spring Semester 2019-20

K. Sai Anuroop, Unnati Athwani, Vivek Mishra, Shalaj Kumar Yadav[†]

Course Instructor: **Prof. Rajshekar K.**

Department of Computer Science and Engineering, IIT Dharwad

April 15, 2020

*Oracle and Java are registered trademarks of Oracle and/or its affiliates

[†]Email IDs of team members in order: [170030035, 170010006, 170020007, 170010015] @iitdh.ac.in

Abstract

In this report we discuss about runtime optimizations in Java Virtual Machines. We first present some basic framework about Java, virtual machines, JVM and code optimization. Next, we explain the philosophy behind Java and understand how Java works, by going through the concepts of bytecode, role of JVM and JIT compiler. Finally, we delve into the runtime optimizations part, where we explain about hotspots, inlining, local, global and control flow optimizations.

Keywords

Java, Virtual Machine, Java Virtual Machine (JVM), Hotspot, WORA, Bytecode, Runtime, Optimization, JIT Compiler, Program

Contents

1	Introduction	4
1.1	Java	4
1.2	Virtual Machine	4
1.2.1	System Virtual Machine	4
1.2.2	Process Virtual Machine	5
1.3	Java Virtual Machine (JVM)	5
1.4	Code Optimization	6
2	How Java works?	6
2.1	Java Philosophy	6
2.2	Bytecode	6
2.3	Role of JVM	7
2.4	Just-In-Time (JIT) Compiler	7
3	Runtime Optimizations	7
3.1	Hotspots	7
3.2	Inlining	8
3.3	Local Optimizations	9
3.3.1	Register Allocation	9
3.3.2	Algebraic Simplification and Reassociation	9
3.3.3	Constant Folding	10
3.4	Control Flow Optimizations	10
3.4.1	Code Motion	10
3.4.2	Loop Unrolling	11
3.4.3	Exception-Directed Optimization	12
3.5	Global Optimizations	12
3.5.1	Escape Analysis	12
3.5.2	Synchronization Optimizations	12
3.5.3	Dead Code Elimination	13
3.6	Native Code Generation	13
	Summary	14
	Conclusion	14
	References	15

1 Introduction

1.1 Java

Java is a general-purpose programming language that is class-based, object-oriented, and designed to have as few implementation dependencies as possible. It is intended to let application developers write once and run anywhere, meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. [1]

1.2 Virtual Machine

A Virtual Machine (VM) is an emulation of a computer system, where these machines use computer architectures to provide the functionality of a physical computer. The physical device on which a virtual machine works is known as the *Host*, whereas the virtual machine is known as the *Guest*.

A single host can have multiple numbers of guests. It is to be noted that the software within the guest cannot make changes to the software of the host system. The computer software that creates and runs the virtual machine is known as the *Hypervisor*. Based on their functions, there are two different types of virtual machines, about which we discuss below.

1.2.1 System Virtual Machine

This type of VM provides full virtualization. Acting as the substitute for a real machine, this type of VM provides functionalities to execute an entire operating system. Hardware resources are shared and managed, forming multiple environments on the host system. These environments are isolated from each other but exist on the same physical host. [2]

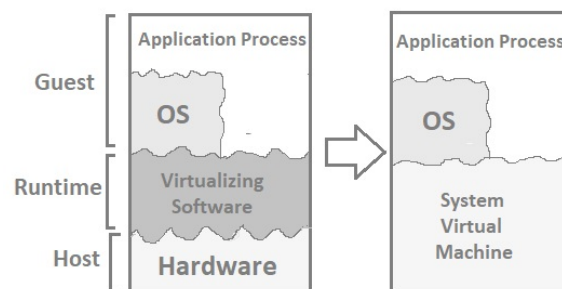


Figure 1: System Virtual Machine

1.2.2 Process Virtual Machine

This type of VM is also known as an *Application Virtual Machine* or a *Managed Runtime Environment*. It runs as a normal application inside the host operating system, supporting a single process. It is created with the starting of the process and is destroyed when the process ends. It provides a platform-independent programming environment to the process, allowing it to execute in the same manner as it currently executes on this platform, on any of the other platforms. [3]

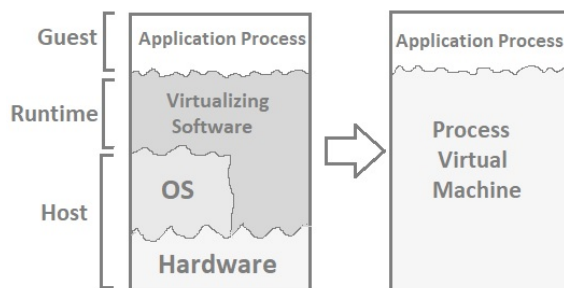


Figure 2: Process Virtual Machine

Process Virtual Machine has become popular with the Java programming language, which uses a Java Virtual Machine for the execution of its programs, about which we shall study in the next section 1.3.

1.3 Java Virtual Machine (JVM)

Java Virtual Machine is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

To understand the Java Virtual Machine, we must first be aware that one may be talking about any of the following three different things when one says *Java Virtual Machine*:

- **The abstract specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm.
- **A concrete implementation** which is known as JRE (Java Runtime Environment).
- **A runtime instance** of JVM which is created whenever a java command is written on the command prompt to run a java class.

Thus, each Java application runs inside a runtime instance of some concrete implementation of the abstract specification of the Java virtual machine.

The Java *HotSpot*¹ Virtual Machine is a core component of the Java SE platform. It im-

¹We shall shortly see in section 3.1 about the eponymous behaviour of certain sections of code, from which the name is derived.

plements the Java Virtual Machine Specification, and is delivered as a shared library in the Java Runtime Environment. [4][5]

1.4 Code Optimization

Code optimization is any method of code modification to improve code quality and efficiency. A program may be optimized so that it becomes smaller in size, consumes less memory, executes more rapidly, or performs fewer input/output operations. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources
- Optimization should itself be fast and should not delay the overall compiling process

2 How Java works?

2.1 Java Philosophy

Java has a philosophy called WORA [6], which stands for *Write Once, Run Anywhere*. Java tries to achieve platform independence by compiling Java code into an intermediate format called bytecode, which is expected to be run on any machine equipped with a JVM, regardless of the underlying computer architecture. In most other programming languages, their compilers generate code that can execute on a specific target machine.

2.2 Bytecode

The Java source compiler (*javac*) reads Java source files and compiles them into class files (see figure 3). As mentioned in section 2.1, unlike many other compilers, like gcc, it does not produce native code for a given target architecture, but it creates bytecode.

The bytecode is composed of a portable instruction set, where all operation codes are represented in a single byte. As a result of this, the bytecode is architecture agnostic, but it is also a lot less performant in itself than the optimized native code.

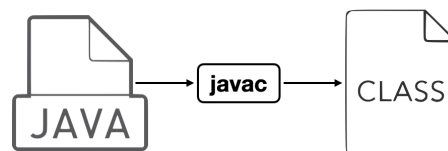


Figure 3: Compile Time

2.3 Role of JVM

Bytecode compilation process does not involve much optimization, except for some inlining (about which we shall discuss in section 3.2 shortly). This means that the bytecode is a lot similar to the source code and interpreting this bytecode directly is rather slow, which now opens doors to some good optimization requirements. Early JVM versions did not have any further optimizations, they worked with the bytecode by interpreting the opcodes directly (see figure 4). To make things more performant, then came the use of Just-In-Time compiler, which enables the runtime to execute optimized code.

The JVM takes its time to carefully watch the code as it executes on the virtual machine. For example it analyses what parts of the code are executed often to decide what part of it is worth the optimization. It also analyses how the code is executed to decide what optimizations can be applied.

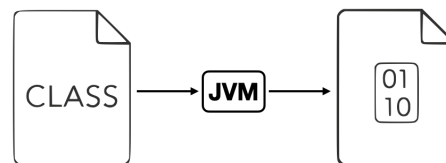


Figure 4: Run Time

2.4 Just-In-Time (JIT) Compiler

JIT compiler interacts with the JVM at run time and compiles appropriate bytecode sequences into native machine code. JIT compilation makes those optimizations possible that take runtime information and statistics into account. The JIT compiler produces optimized code tailored to the target architecture that replaces the interpreted version on the fly. Since compilation happens on a background thread, it does not really interrupt the program execution.

JIT can lead to performance gains in the execution speeds, if methods in the source code are executed frequently. Otherwise it is to be noted that the time a JIT compiler takes to compile the bytecode is added to the overall execution time, and could lead to a higher execution time than an interpreter for executing the bytecode.

However, JIT can also significantly affect startup time, even if the program eventually achieves very good peak performance. [7][8]

3 Runtime Optimizations

3.1 Hotspots

In a typical program, there is only a small portion of the code that is executed frequently, and often, it is this portion of the code that significantly affects the performance of the whole

program. Such sections of code are called HotSpots. The more the JVM runs a particular method or a loop, the more information it gathers to make sundry optimizations so that a faster native code is generated.

```
1 int sum = 0;
2 for (int i = 0; i <= 1000; i++) {
3     sum += i;
4 }
```

Code Snippet 1: Since this code runs multiple times, it is a HotSpot

In the above example, a local copy of `sum` would be stored in a register, specific to a particular thread. All the operations would be done to the value in the register and when the loop completes, the value would be written back to the memory. Let us consider another insightful example below.

```
1 for (int i = 0; i <= 50; i++) {
2     System.out.println(myobj1.equals(myobj2)); //two objects
3 }
```

Code Snippet 2: This kind of behavior is only possible when the JVM knows how the code behaves

In this case, JVM would notice that for each iteration, `myobj1` is of class `String` and hence, it would generate code corresponding to the `.equals()` method of the `String` class directly. Thus, as no lookups will be required, the compiled code would execute faster. [9]

We now discuss some of the other optimizations made by the JIT compiler during runtime. [10]

3.2 Inlining

Inlining eliminates the cost of the method calls by substituting a method call with the body of called method. Bringing related code together enables further optimizations, which would otherwise have been impossible for separate methods, like Common Subexpression Elimination² or Heap Allocation Elimination.

```
1 public void bar() {
2     int product = foo(2,3);
3     // Do something with product
4 }
5 public static int foo(int a, int b) {
6     return a*b;
7 }
```

Code Snippet 3: A simple case where inlining will be done

In the above example, method `foo` will be inlined eventually.

²For more details about Common Subexpression Elimination and other local optimizations, please refer to the lecture slides of CS 406 Compilers course [11]

3.3 Local Optimizations

Local optimizations analyze and improve a small section of the code at a time. Many local optimizations implement tried and tested techniques used in classic static compilers. The optimizations include:

3.3.1 Register Allocation

Register allocation is the process of assigning variables to registers and managing data transfer in and out of registers. Using intelligent algorithms like a customized version of Linear Scan Register Allocation is very critical, as a good register allocator can be orders of magnitude *better* than a bad one. [12]

3.3.2 Algebraic Simplification and Reassociation

Algebraic simplification uses algebraic properties of operators or particular operand combinations to simplify expressions.

- Some statements can be deleted like in below.

```
1 x = x + 0;  
2 x = x * 1;
```

Code Snippet 4: These statements would eventually be deleted

- Some statements can be simplified.

```
1 x = x * 0;  
2 y = y ** 2;  
3 x = x * 8;  
4 x = x * 15;
```

These statements will be simplified as in below.

```
1 x = 0;  
2 y = y * y;  
3 x = x << 3;  
4 t = x << 4; x = t - x;
```

Code Snippet 5: Simplified Statements

Reassociation refers to using associativity, commutativity, and distributivity to divide expressions into parts that are constant, loop invariant and variable.

- Associativity and distributivity can be applied to improve parallelism (reducing the height of expression trees).

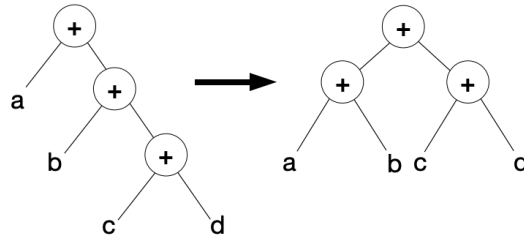


Figure 5: Tree height reduction

3.3.3 Constant Folding

Constant Folding [13] is a technique where the compiler evaluates constant expressions at compile time and replaces those expressions with their respective values. Expressions with constant operands can be evaluated at compile time, thus improving run-time performance and reducing code size by avoiding evaluation at compile-time. [14]

```
1 int foo () {
2     return 3 + 5;
3 }
```

Code Snippet 6: Code fragment before constant folding

In the code fragment below, the expression $(3 + 5)$ can be evaluated at compile time and replaced with the constant 8. Below is the code fragment after constant folding.

```
1 int foo () {
2     return 8;
3 }
```

Code Snippet 7: Code fragment after constant folding

Constant folding is a relatively easy optimization and can be applied after applying other optimizations that create constant expressions.

3.4 Control Flow Optimizations

Control flow optimizations analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency. [15]

3.4.1 Code Motion

Code motion, also called *Code Hoisting*, unifies sequences of code common to one or more basic blocks to reduce the code size and potentially avoid expensive re-evaluation. [16]

The most common form of code motion is loop-invariant code motion that moves statements that evaluate to the same value during every iteration of the loop, to somewhere outside the loop. [17]

```

1 a = 200;
2 while(a>0) {
3     b = x + y;
4     if (a % b == 0)
5         System.out.println(a);
6 }

```

This code can be further optimized as given below.

```

1 a = 200;
2 b = x + y;
3 while(a>0) {
4     if (a % b == 0)
5         System.out.println(a);
6 }

```

Code Snippet 8: Code optimization using Code Motion

3.4.2 Loop Unrolling

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions. [18]

```

1 for (i = 0; i < 100; i++)
2     g ();

```

Code Snippet 9: Program not using Loop Unrolling

In the code fragment below, the body of the loop can be replicated once and the number of iterations can be reduced from 100 to 50. Below is the code fragment after loop unrolling. [19]

```

1 for (i = 0; i < 100; i += 2) {
2     g ();
3     g ();
4 }

```

Code Snippet 10: Program after using Loop Unrolling

Advantages:

- Increases program efficiency.
- Reduces loop overhead.
- If statements in a loop are not dependent on each other, they can be executed in parallel.

Disadvantages:

- Increased program code size, which can be undesirable.
- Increased usage of registers in a single iteration to store temporary variables, which may reduce performance.

3.4.3 Exception-Directed Optimization

Exception-Directed Optimization (EDO) [20] optimizes exception-intensive programs without slowing down exception-minimal programs. It is a feedback-directed dynamic optimization consisting of three steps:

- Exception Path Profiling: Attempts to detect hot exception paths.
- Exception Path Inlining: Embeds every hot exception path into the corresponding catching method.
- Throw Elimination: Replaces a throw with a branch to the corresponding handler.

3.5 Global Optimizations

Global Optimizations work on the entire code at once. They can provide a great increase in performance, but come with the cost of greater compilation time. Hence, they are efficient as well as expensive. Some of these optimizations are:

3.5.1 Escape Analysis

This is an important technique which Just-In-Time Java compiler can use to analyze the scope of a new object and decide whether it might not be allocated on Java heap space. This is a very important performance optimization, because stack allocation and de-allocation are much faster than heap space allocation. This can be easily applied on objects that are not visible outside the current method and scope. [21]

```
1 public String getObjDescription() {  
2     Obj car = new Obj();  
3     String description = car.generateDescription();  
4     return description;  
5 }
```

Code Snippet 11: A code snippet where the object is neither visible nor accessible outside `getObjDescription()` method, hence suitable for optimization

3.5.2 Synchronization Optimizations

Whenever variables are shared between threads, we must use synchronization to ensure that updates made by one thread are visible to other threads on a timely basis. The Java programming language provides synchronization constructs (synchronized methods and blocks) to permit safe use of concurrently-accessed data structures. These constructs are used pervasively in both the standard libraries and the run-time system. In many cases, a large number of these operations may be safely removed without compromising program semantics, thus improving performance. Removing these operations manually may be inconvenient, error-prone or even impossible. [22]

```

1 public void lockElision() {
2     Obj resource = new Obj();
3     synchronized (resource) {
4         // do something
5     }
6 }

```

Code Snippet 12: The synchronized section has no real effect, because the lock can only be accessed by the current thread

3.5.3 Dead Code Elimination

As the name suggests, dead code is the code that is ineffective, i.e., does not contribute to the outcome of the program. It can be removed without any noticeable behaviour change in the program, except for performance. The JVM removes all these unnecessary code as part of this optimization making the implementation much faster.

```

1 boolean x = true;
2 if (x) {
3     // do something
4 } else {
5     // this is dead code!
6 }

```

Code Snippet 13: A trivial example of a dead code

It is not an error as it is still syntactically correct, but the compiler does give a warning. If not removed by the user, it is anyways removed by the compiler for the sake of better performance.

3.6 Native Code Generation

Native code generation processes vary, depending on the platform architecture. Generally, during this phase of the compilation, the trees of a method are translated into machine code instructions. Some small optimizations are performed according to architecture characteristics. The compiled code is placed into a part of the JVM process space called the *code cache*; the location of the method in the code cache is recorded, so that future calls to it will call the compiled code. This subsequently leads to performance gains in the execution speed, unless the compiled methods are executed less frequently.

Summary

Java is a general-purpose programming language that is class-based, object-oriented, and designed to have as few implementation dependencies as possible. It follows the WORA philosophy. Each Java application runs inside a runtime instance of some concrete implementation of the abstract specification of the Java virtual machine. The Java source compiler reads Java source files and compiles them into class files. JIT compiler interacts with the JVM at run time and compiles appropriate bytecode sequences into native machine code. JIT compilation makes those optimizations possible that take runtime information and statistics into account. These runtime optimizations include inlining; local optimizations like register allocation, algebraic simplification and reassociation, constant folding; control flow optimizations like code motion, loop unrolling, exception-directed optimization; global optimizations like escape analysis, synchronization optimizations and dead code elimination.

Conclusion

This report provided some basic concepts about Java, its philosophy, working principles; discussed the need for optimization and presented some runtime optimization techniques along with their brief analysis. As discussed in the report, JIT compiler makes these optimizations and significantly increases the performance of running applications.

References

- [1] [https://en.wikipedia.org/wiki/Java_\(programming_language\)](https://en.wikipedia.org/wiki/Java_(programming_language))
- [2] <https://www.elprocus.com/virtual-machine/>
- [3] https://en.wikipedia.org/wiki/Virtual_machine#Process_virtual_machines
- [4] <https://www.artima.com/insidejvm/ed2/jvm.html>
- [5] <https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html>
- [6] https://en.wikipedia.org/wiki/Write_once,_run_anywhere
- [7] <https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/>
- [8] <https://advancedweb.hu/jvm-jit-optimization-techniques/>
- [9] https://www.tutorialspoint.com/java_virtual_machine/java_virtual_machine_tutorial.pdf
- [10] <https://www.ijraset.com/files/serve.php?FID=3347>
- [11] <https://hegden.github.io/cs406/slides/week10.pdf>
- [12] <https://www.oracle.com/technetwork/java/whitepaper-135217.html>
- [13] <https://web.stanford.edu/class/archive/cs/cs143/cs143.1112/materials/lectures/lecture14.pdf>
- [14] http://compileroptimizations.com/category/constant_folding.htm
- [15] <https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/>
- [16] <https://www.quora.com/p/9888/explain-code-motion-1/>
- [17] <https://www.geeksforgeeks.org/code-optimization-in-compiler-design/>
- [18] <https://www.geeksforgeeks.org/loop-unrolling/>
- [19] https://compileroptimizations.com/category/loop_unrolling.htm
- [20] <http://swmath.org/software/29926>
- [21] <http://performantcode.com/compiler/escape-analysis/>
- [22] http://web.eecs.umich.edu/~bchandra/courses/papers/Ruf_Sync.pdf

