Research & Whitepaper Volume 1.0

High Speed Java

Java at the speed of C – an analysis of High Speed Java programming environment

Author: Bikramjit Debnath

November 06

Contents

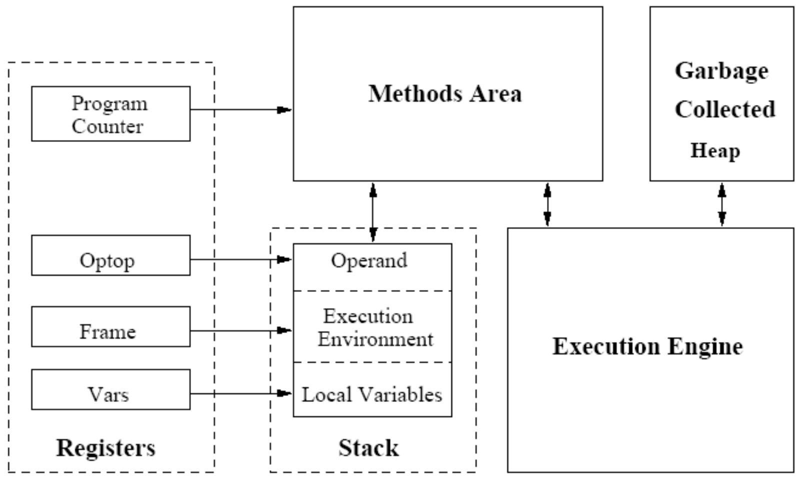
1

Deep down the JVM

The Java Virtual machine (JVM) is one of the most neglected subjects during estimation and development of high-performance java applications. The JVM provides the most vital interface between java code and the target platform. The following section elaborates the crucial JVM internals in an easy to understand approach.

The JVM executes the java byte-codes. The implementation of JVM across various platforms made java portable. It is said to be virtual for it is implemented in software rather than in the hardware platform. JVM interprets a stream of byte-code as a sequence of instructions for execution. JVM is stack-based and consists of one-byte **opcode** followed by zero or more **operands**. The JVM instruction set defines more than 202 standard opcodes, around 25 quick variations of some opcodes, and three reserved opcodes. Operands provide additional information to execute the action.

The JVM is divided in five basic components as shown:



As shown in the above figure, **registers, stack, garbage-collected heap, methods area,** and **execution engine** are implemented in every JVM. The byte codes are stored in the methods area. The program counters point to the next byte for execution in the methods area. All operations in the JVM occur through the stack. Data is pushed into the stack from constant pool in the methods area and from the local variable section of the stack. The first section of the **stack frame** is **local variables section** which contains all the local variables for current method invocation. The **vars** register points to this section of stack frame. The second section is **execution environment** which maintains the stack operations. The **frame** register points to this section. The final section is **operand stack**, the **optop** register points to this section, which is used for storing parameters and temporary data for expression evaluations.

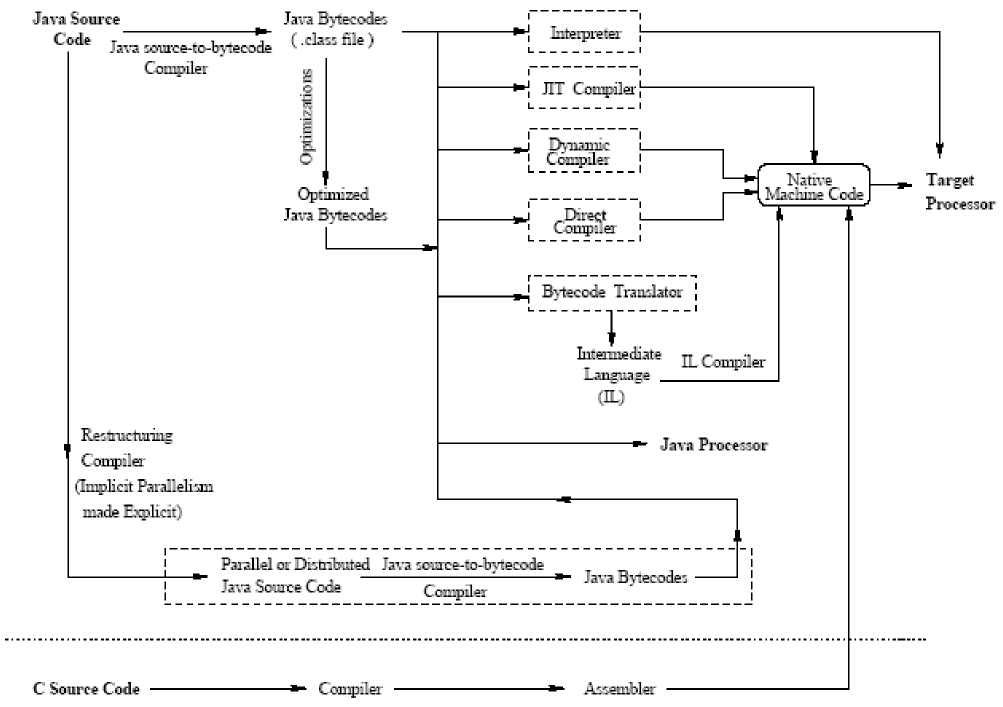
Memory is dynamically allocated from the garbage-collected heap using the **new** operator in executing programs. The JVM specification does not allow the user to explicitly free allocated memory. Instead, the garbage-collection process monitors the memory for returning unused objects to the memory pool. A large variety of garbage collection algorithms have been developed, including **reference counting, mark-sweep, mark-compact, copying** and **non-copying implicit collection**.

The core of the JVM is the **execution engine**. This is a virtual processor which interacts with me method area for byte-code execution. Various implementation of execution engine are depicted in the subsequent section.

Alternative execution techniques

A disadvantage of byte-code compilation is the computational model of the byte code which implements a stack machine. Mapping this (virtual) stack machine on existing CPUs (that are register based) is harder than directly generating register-oriented code Until recently, many high-performance codes have been implemented in the **C** language, mainly for reasons of efficiency. From a software-development point-of-view, it is desirable to use Java instead of C, allowing using features like **threads, objects, exceptions, runtime type checks,** and **array bounds checks** in order to ease programming and debugging. Unfortunately, several of these features introduce performance overheads, making it difficult to obtain the same sequential speed for Java as for C. Hence, along with Java's widespread usability; the need for a more efficient execution mode has become apparent.

Therefore, apart from traditional interpretation techniques for JVM implementation, a variety of execution techniques have been proposed to reduce the time of java programs. The alternatives compared to the execution of programs written in typical languages like C are shown in the following figure:



The alternatives for executing java code as shown above are:

- Java Interpreters
- Java Compilers
- Java Compiler Optimizations
- Hot-spot JVM (Dynamic compiler)
- Just-In-Time (JIT) Compilers
- Direct Compilers
- Byte-Code to Source Translators
- Java Processors

Each of this implementation varies up to a great extent when tested to solve CPU and memory hungry problems like – **Iterative Deeping A* algorithms (15-puzzle), Traveling Sales Person Problem (TSP), Successive Over relaxations (SOR)** etc. The results show that choosing the best suitable JVM for a specific application (Distributed, Embedded etc.) can make the speed similar to C versions. For example, the **Manta system** (http://www.cs.vu.nl/manta) is such a fast native java compiler.

Note

The detailed discussions about the above techniques are out of the scope of this document.

2

High Speed Java – Tips and Techniques

The last section identified the key areas of JVM in executing java code. High performance in java code execution can be achieved mainly in two ways – (1) After-compilation (AC) techniques and (2) Before-compilation (BC) Techniques. AC techniques mainly involve speculating on the correct implementation of JVM rather than depending upon any available interpreter for high-performance application development. The following section explores some BC techniques to improve the performance.

After-Compilation (AC) & Before-Compilation (BC) Techniques

Optimization during the development phase (**BC**) and the execution phase (**AC**) are crucial for high-speed execution of Java codes. The common optimization techniques are:

- Object In lining (AC)
- Method In lining (AC)
- Escape Analysis (AC)
- Bound-check elimination/deactivation (AC)
- Closed-world Assumption (BC)
- Common-sub expression elimination (BC)
- Loop-invariant code motion (BC)
- Minimal serialization, RMI and synchronization. (BC)

The basic understanding of AC techniques will be helpful to select the appropriate version of JVM which supports those optimization features.

Object In lining

Java's object model leads to many small objects with references to other objects. Performance is improved by reducing overheads from object creation, garbage collection, and pointer dereferencing. The following code fragment shows an example of object in lining (When compiler can derive that the array "**a"** is never reassigned in objects of class **A** (left), then the array can be statically in lined into objects of class **A** (right).

class A{
int[] a = new int[10];
}
Original class A with separate array object
	class A{
	int a[10];
	}
	A with inclined array

Note that the shown optimization can not be implemented manually because Java lacks a corresponding syntactical construct for arrays.

Method In lining

In C-like languages, function in lining is a well-known optimization for avoiding the costs of function invocation. In object-oriented programs it is common to have many, small methods. So method in lining is desirable for Java. However, in the presence of polymorphism and dynamic class loading, only methods declared as **static** can be safely in lined. For all other methods, sub-classes may override the implementation of a super class. These program semantics at the core of object-oriented programming prevent efficient method in lining. In the following example, the method **inc** would be an ideal candidate for in lining due to its small size. However, it can only be in lined if the compiler can safely derive that there exists no subclass of **A** that replaces the implementation of **inc**.

```
class A{
int a;
void inc(){ a++; }
void other(){ inc(); }
}
```

Escape Analysis

Escape analysis considers the objects created by a given method. When the compiler can derive that such an object can never escape the scope of its creating thread (for example, by assignment to a static variable), then the object becomes **unreachable** after the method has terminated. In this case, object allocation and garbage collection can be avoided all together by creating the object on the **stack** of the running thread rather than via the general-purpose (**heap**) memory. In the case of creation on the stack, method-local objects can be as efficient as function-local variables in C.

Bound-check elimination/deactivation

The violation of array boundaries is a frequently occurring programming mistake with C-like languages. To avoid these mistakes, Java requires array bounds to be checked at runtime, possibly causing runtime exceptions. This additional safety comes at the price of a performance penalty. A simple-minded, but unsafe optimization is to suppress the code generation for array-bounds checks altogether. The idea is that boundary violations will not occur after some successful, initial program **testing** with bounds checks activated. Completely deactivating array-bounds checks thus gives the unsafety of C at the speed of C. For example, if a method accesses **a[i]** in more than one statement, then only the first access needs a bound check. For all other accesses, the checks can safely be omitted as long as Manta can derive from the code that neither the array base "**a**" nor the index "**i**" has been changed in the meantime. For this purpose, the compiler can perform a **data-flow analysis**, keeping track of the array bases and related sets of index identifiers for which bounds checks have already been issued

Closed-world Assumption

Many compiler optimizations require knowledge about the complete set of Java classes that are part of an application. Java's polymorphism, in combination with dynamic class loading prevents such optimizations. In this case, the programmer has to explicitly annotate methods as **final** in order to enable a large set of optimizations. However, the final declaration has only limited applicability as it selectively disables polymorphism. Its use for improving application performance furthermore contradicts its original intention as a means for class-hierarchy design. Many high-performance applications (scientific) consist of a fixed set of classes and do not use dynamic class loading at all. Such applications can be compiled under a closed world assumption: **all classes are available at compile time** to gain excellent runtime performance.

Common-sub expression elimination

Calculating the common sub-expression initially and then substituting all references with the calculated value in a loop increases the runtime performance for reducing the load on JVM stack.

for(…){
a[i] = a[j]*3*CONST_VAL;
b[i] = a[i]+ a[j]*3*CONST_VAL;
a[i] = b[i] - CONST_VAL;
}
Original loop with multiple use of a common expression
	int sub = a[j]*3*CONST_VAL;
	for(…){
	a[i] = sub;
	b[i] = a[i]+ sub;
	a[i] = b[i] - CONST_VAL;
	}
	loop with common sub-expression elimination

Loop-invariant code motion
Moving the sub-expressions or the piece of code that does not depend upon the iterative loop values out of the loop enhances the performance of execution.

for(…){
iVal = a[j]*3*CONST_VAL;
a[i] = iVal;
b[i] = a[i]+ iVal;
a[i] = b[i] - iVal;
}
Original loop with improper use of a common expression
	iVal = a[j]*3*CONST_VAL;
	for(…){
	a[i] = iVal;
	b[i] = a[i]+ iVal;
	a[i] = b[i] - iVal;
	}
	loop with loop-invariant code motion

Minimal synchronization, RMI and serialization

In Java, synchronization is provided through **monitors**, which language level constructs are used to guarantee mutually-exclusive access to shared data. Since java allows an object to be synchronizable (with or without synchronized methods), using a lock structure for each object can be very costly in terms of memory. This requires the runtime-system to first query in the **monitor-cache** before it is used, which is quite inefficient. Further the monitor-cache itself should be locked to avoid race condition. Thus monitor approach is not scalable.

The **thin lock** and **think lock** approach in some JVM to optimize locking mechanism to avoid locks up to an excessive nesting depth improves performance. However, optimal use of synchronization is necessary to improve execution performance in any application.

The current Java RMI is designed to support client-server applications that communicate over TCP based networks. Some RMI design goals results in severe performance limitations for high-performance applications on closely connected environments such as clusters and distributed memory processors.

Similarly serialization is also a costly operation, and making class serializable selectively, optimal use of RMI methods can improve the overall performance of any application.

Conclusion

From a software-development point-of-view, it is desirable to use Java instead of C, allowing using object-oriented features in order to ease programming and debugging. Unfortunately, several of these features introduce performance overheads, making it difficult to obtain the same sequential speed for Java as for C. This paper discusses the various way of improving the runtime performance of Java application with the inside view of JVM and elaborates a range of existing optimization techniques for Java both during development and compilation and their performance impact on application.