Java Compiler: Review on Code Generation and Optimization Techniques

Prajakta P. Gotarane  
M.Tech Scholar,  
SNDT Women’s university,  
Santacruz, Mumbai,  

Prof. Sumedh N. Pundkar  
Asst. Prof. In Comp. Sci. & Tech,  
SNDT Women’s university,  
Santacruz, Mumbai.

ABSTRACT

Today, Java has gaining popularity in software development. Java has widely used in many system and lots of running system has been developed. In compiler design code optimization is one of the important phase. This paper describes new java code optimization techniques for execution of java program by reducing time and space consumption. We review and discuss some effective compiler optimization techniques for java compiler such as Dead code elimination, Inlining small function, code hoisting, eliminating common sub-expression. Here, In this paper we describe some new techniques to reduce runtime overhead on java compiler by proper use of data structure, exception handling and garbage collection. We provide some pattern in java which provides expert optimized code to user which will be the optimized solution in terms of time and space complexity.

Keywords

Java Compiler, Code Optimization, Data structure.

1. INTRODUCTION

Basically the function of any compiler is to transform a piece of source code (such as c and c++) into machine code and also compiler optimized that program in such a way to make it run faster. Java compiler is different from c & c++
language suitable for many writing programs, that can be distributed and reuse on many /multiple platform. Java is excellent language because of flexibility and reusability. Java is platform independent language so java compiler transform java source code into platform independent byte code and which is then executed by java virtual machine i.e JVM.

In a compiler design optimization is one of the important phase. Code optimization is process of transforming a piece of code to make it more efficient in terms of time and space without changing the output. The difference should be visible to code use is that the program will run faster and it consumes less memory. In fig.1. The task of front end (Lexical, Syntax, Semantic Analysis) is to generate intermediate code of source code for target machine. But the code generated by front end is unoptimized code. So the optimization is something that removes unnecessary code lines and then arrange the sequence of statement in such order to speed up program execution without wasting resources such as CPU and memory.

In this paper to validate our approach of code optimization in java the important point is it takes place on working code. So always perform optimization on a code after you get the code working. The structure of this paper is as follows section II we discuss various compiler code optimization methodologies. Section III comparative study of various java code optimization techniques section IV Paper ends with conclusion and future work.

2. LITERATURE REVIEW:
Optimization Strategies:

It is a time to look into exactly what optimization we can perform to speed up the execution of program. We can not always able to optimized every piece of problem code, but our goal is to find out the area in java programming language where we can optimized that code to make it more efficient in terms of time and space.

2.1 Rethink Algorithms

Many C/C++ programmers have traditionally resorted to assembly language when the issue of performance is raised. As a Java programmer, you don't have this option. This is actually a good thing because it forces you to take a closer look at your design approach instead of relying on heavier processor dependence to solve your problems. What the assembly heads don't realize is that much more significant gains can be made by entirely rethinking an algorithm than by porting it to
assembly. And the amount of time spent hand-coding tedious assembly can easily result in a leaner, more efficient algorithm [2].

This same ideology applies to Java programming. Before you run off writing native methods and expanding loops to get every little ounce of performance (which you learn about in the next sections), take a step back and see whether the algorithm itself has any weaknesses. To put this all into perspective, imagine if programmers had always resorted to optimizing the traditional bubble sort algorithm and had never thought twice about the algorithm itself. The quick sort algorithm, which is orders of magnitude faster than the bubble sort without any optimization, would never have come about.

2.2 Use Native Methods

Native methods (methods written in C or C++ that can be called from Java code) are typically much faster than Java methods. The reason I'm reluctant to promote their use is that they blow the platform-independence benefit of using Java, therefore tying your program to a particular platform. If platform independence isn't high on your list, however, by all means look into rewriting problem methods in C [2].

2.3 Use Inline Methods

Inline methods, whose bodies appear in place of each method call, are a fairly effective means of improving performance. Because the Java compiler already inlines final, static, and private methods when you have the optimization switch turned on, your best bet is to try to make as many methods as possible final, static, or private. If this isn't possible and you still want the benefits of inlined code, you can always inline methods by hand: Just paste the body of the method at each place where it is called. So we can archive the speed of execution

2.4 Replace Slow Java API Classes and Methods

There may be times when you are using a standard Java API class for a few of its features, but the extra baggage imposed by the generic design of the class is slowing you down. In situations like this, you may be better off writing your own class that performs the exact functionality you need and no more. This streamlined approach can pay off big, even though it comes at the cost of rewriting code that already works [2].

Another similar situation occurs when you are using a Java API class and you isolate a particular method in it that is dragging down performance. In this situation, instead of rewriting the entire class, just derive from it and override the troublesome method. This is a good middle-of-the-road solution because you
leverage code reuse against performance in a reasonable manner [2].

2.5 Use Look-Up Tables

An established trick up the sleeve of every programmer who has wrestled with floating-point performance problems is the look-up table. Look-up tables are tables of constant integer values that are used in place of time-consuming calculations. For example, a very popular type of look-up table is one containing values for trigonometric functions, such as sine. The use of trigonometric functions is sometimes unavoidable in certain types of programs. If you haven’t noticed, Java’s trigonometric functions are all floating-point in nature, which is a bad thing in terms of performance. The solution is to write an integer version of the desired function using a look-up table of values. This relatively simple change is sometimes a necessity considering the performance hit you take by using floating-point math [2].

2.6 Constant Folding:

This optimization involves simply taking an expression where all operands are constants, and replacing the expression with the constant result computed at compile time. In this examples, the compilers should replace the expression 3+5 with the value of 8.

Performed constant propagation and useless expression elimination, essentially generating code for System.out.println(8).

Class CF{
    public void f() {
        int a = 3 + 5; // This value is obviously always 8 so
        System.out.println(a);
    }
}

2.7 Constant Propagation:

This optimization involves essentially computing as many values as possible as compile time, rather than at run time. In this example, the value of a is set to the constant 5, so the value of the argument of the println() can be set to 5. Javac optimized this by propagating the constant, as well as eliminating the useless expression a=5. The other compilers do not perform this optimization.

Class CP {
    public void f() {
        int a = 5;
        System.out.println(a); // a will always be 5 so
        // can just generate code for System.out.println(5)
        // which makes the statement a = 5 a useless
    }
}
2.8 Useless Expression Elimination:
This very simple optimization entails eliminating assignments to variables when the variable is not used again after the assignment. Javac optimizes this function correctly to the "empty" function (simply return) while none of the other compilers perform this optimization.

```java
Class UE {
    public void f() {
        int a = 5; // since the value of a is never used, it
        // does not need to be assigned
    }
}
```

2.9 Copy Propagation:
This very simple optimization entails replacing use of one variable with another when it is known that the two are equal. This is most often performed along with useless expression elimination. None of the compilers perform this optimization, though Javac does with constants.

```java
Class CP {
    public void f(int a) {
        int b = a; // this value of a is never used so this
        // statement is not necessary
        c = b;
        System.out.println(c);
    }
}
```

2.10 Redundant Expression Elimination:
This optimization involves removing a variable assignment to a value that is not used between the assignment and the next time the variable is assigned to a different value. In this example, variable c is first assigned to be the first parameter, a, but is assigned again to the second parameter, b, before the variable c is used. Therefore, the first assignment to c can be eliminated in the compiled program. Unfortunately, none of the compilers perform this optimization.

```java
class Red {
    public void f(int a, int b) {
        int c;
        c = a; // this value of a is never used so this
        // statement is not necessary
        c = b;
        System.out.println(c);
    }
}
```

2.11 Common SubExpression Elimination:
This optimization involves replacing repeated assignments to an identical mathematical expression with assigning the first variable to the expression, then assigning the other variables to the first variable. This optimization is often used with "temporary" variables that the compiler uses to store intermediate results. In this example, a\*b is
computed twice, but it is only necessary to compute it once, then merely copy the value into the second variable. None of the compilers perform this optimization.

class CSE {
    public void f(int a, int b) {
        int c = a*b;
        int d = a*b;  // we already know a*b so this should be
        // changed to d = c
        System.out.println(c);
        System.out.println(d);
    }
}

2.12 Mathematical Strength Reduction:

This optimization involves changing mathematical expression that take "longer" time with ones that will take "shorter" time. This can be, for example, replacing multiplication (b = a*2) with addition (b = a+a) or replacing exponentiation (b = a^2) with multiplication (b = a*a). In this example, the expression b = a*2 can be replaced with the expression b = a+a.

which is faster on most processors. Again, none of the compilers perform this optimization.

class MSR{
    public void f(int a) {
        int b = a*2;  // this can be changed to a+a which is
        // faster on most processors
        System.out.println(b);
    }
}

2.13 Global Optimizations

Several code segments were tested for optimizations beyond basic blocks. None of the compilers produced good results.

2.13.1 Global Constant Propagation:

This optimization is an extension of constant propagation within a basic block where constant expressions are replaced at compile time by the constant value throughout the program. In this example, the javac compiler propagated the constant 5 into the if statement (changing the c = b to c = 5) and eliminated the then useless variable b, but did not propagate the constant further. The other compilers did not perform any optimizations on this code.

Class GCP{
    public void f(boolean a) {
        int b = 5;
        int c;
        if(a) {
            c = b; }
        else {
            int d = c;  // c will always be 5 so do not compute
        }
    }
}
2.13.2 Global Common Subexpression Elimination:

This optimization is an extension of common subexpression elimination within a basic block where common expression are to be stored in a temporary value so that they need only be computed once at run time. In this example, d=a*b; should be changed to t=a*b; d = t; and e=a*b; should be changed to t=a*b; e=t; so that f = a*b can be replaced by f = t. Again, none of the compilers perform this optimization. The results of this code are a bit misleading in that the code with or without the optimization have the same number of instructions. This is because the common subexpression is small, on larger subexpressions there would be a noticeable reduction in the number of instructions required.

class GCSE {
    public void f(int a, int b, boolean c) {
        int d = 0;
        int e = 0;
        int f;
        if(c) {
            d = a*b; }
        else {
            e = a*b; }
        f = a*b; // No matter what c is, a*b has been computed
    }
}

2.13.3 Dead Code Elimination:

This optimization involves removing byte codes from the output that will never be executed. In this example, the if clause will never be true, so the byte codes that are generated by the line System.out.println(5) are not necessary. This optimization is very useful with constant propagation to eliminate unnecessary code in constructs like debug = false; ... if(debug) { ... } ..... 

class DC {
    public void f() {
        if(false)
            System.out.println(5); // the clause of the if
statement will never be true, so this statement can be eliminated.

2.13.4 Code Hoisting:

This optimization involves moving common code that will be executed regardless of the path of execution (i.e. very busy expressions) to be computed as early as possible so as to reduce the size of the code. This optimization does not affect running time of the code, but for a language like Java where the code is likely to be transferred over a network immediately before execution, code size will make a large impact on the perceived running time of the code. Also, after hoisting, the if statement in this example because useless, so it can be eliminated. Copy propagation can also be done after hoisting the code, to reduce this function to essentially a System.out.println(a). None of the compilers perform this optimization.

```java
class CodeHoist {
    public void f(int a, boolean b) {
        int c;
        if(b) {
            c = a; 
        } else {
            c = a; 
        } // Since c=a is computed no matter what the value of
        // b is, the statement can be hoisted to before the if
        // statement. This will not affect the running time
        // of the program, but will make the code smaller.
        System.out.println(c);
    }
}
```

2.14 Errors detected by the compiler and runtime errors

All syntax errors and some of the semantic errors (the static semantic errors) are detected by the compiler, which generates a message indicating the type of error and the position in the Java source file where the error occurred (notice that the actual error could have occurred before the position signaled by the compiler). Other semantic errors (the dynamic semantic errors) and the logical errors cannot be detected by the compiler, and hence they are detected only when the program is executed. Let us see some examples of errors detected at runtime:

Example 1: Division by zero: 2) File does not exist: 3) Dereferencing of a null reference:
3. COMPARATIVE STUDY

Table 1. Comparison of various code optimization strategies

<table>
<thead>
<tr>
<th>S. N</th>
<th>Code Optimization Techniques</th>
<th>Extract of the paper</th>
<th>Required Execution time</th>
<th>Required Memory</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant Folding Optimization is done by replacing all expressions with constant result computed at compile time</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Constant Propagation: Done by computing as possible as values at compile time rather than run time</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Useless Expression Elimination: Done by eliminating unnecessary expressions</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Copy Propagation: Done by replacing one variable by another when they are equal</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Common SubExpression Elimination: It involves replacing repeated assignments to an identical mathematical expression with assigning the first variable to the expression, then assigning the other variables to the first variable.</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Mathematical Strength Reduction It involves changing mathematical expression that take &quot;longer&quot; time with ones that will take &quot;shorter&quot; time</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Global Constant Propagation This optimization is an extension of constant propagation within a basic block where constant expressions are replaced at compile time by the constant value throughout the program.</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Global Common Subexpression Elimination: This optimization is an extension of common subexpression elimination within a basic block where common expression are to be stored in a temporary value so that they need only be computed once at run time</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Dead Code Elimination: This optimization involves removing byte codes from the output that will never be executed.</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Code Hoisting It involves moving common code that will be executed regardless of the path of execution to be computed as early as possible so as to reduce the size of the code</td>
<td>Less</td>
<td>Less</td>
<td>More</td>
<td></td>
</tr>
</tbody>
</table>
4. CONCLUSION AND FUTURE SCOPE

This paper provides a small survey on java code optimization techniques. Optimization is done to reduce execution time and code size (file size). Our focus is on compiling various patterns and find out the area in java coding to optimized it and produced efficient java code for user. And there by increase the performance of program.

In future our intent is to focus on implementation of java compiler using above strategies as well as array optimization, loop optimization using proper data structure and will do the analysis of result by using proposed techniques which is the expert code generator for java programming language.

REFERENCES:

[1] book- IBM Just-In-Time Compiler (JIT) for Java :Best practices and coding guidelines for improving performance


LILJA,” Techniques for Obtaining High Performance in Java Programs” Minnesota Supercomputing Institute, University of Minnesota.

[8] Peter Sestoft, “Java performance Reducing time and space consumption”, Royal Veterinary and Agricultural University, Copenhagen, Denmark and IT University of Copenhagen, Denmark Version 1.0 of 2003-03-31.


[12] Compilers--- Alfred V.aho && Ravi Sethi


ABOUT AUTHORS

Ms. Prajakta Gotarane

Completed her B.E in Information technology Pune University. Currently pursuing master degree in computer science and technology from Usha Mittal Institute of Technology, SNDT women’s University, and Mumbai.

Prof. Sumedh Pundkar

Completed his B.E in computer science and engineering from Amravati University. He has competed his master degree from Mumbai university. Currently he is working as Assistant Professor and Head of department in Usha Mittal Institute of Technology, SNDT womens University, and Mumbai. His field of research is Software engg, distributed computing and security.