Estimation Benefit of Java Optimization Techniques

Asmaa Yaseen Hamo, Rahma Saleem Alsawaf
Software Engineering Department, Computer and Mathematics Science College, University of Mosul, Mosul, Iraq

Abstract: optimization is the process of modifying working code to a more optimal state based on a particular goal. Code optimization may be on execution time or memory utilization. This paper include an experiment that measures the execution time of Java code before and after applying of 28th techniques for time optimization. The result shows that some techniques may reduce the execution time till 1/10, it can give the user an impression on the benefit of every techniques.

Introduction

When writing Java code it can be easy to make simple mistakes that seem harmless on the surface but, as the application grows larger, it can show themselves to be slow, resource intensive processes that could use a tune-up. So it is important to optimize the code [1]. Optimization is the process of transforming a piece of code to make it more efficient (either in terms of time or space) without changing its output or side-effects. The only difference visible to the code’s user should be that it runs faster and/or consumes less memory[3]. This paper aims to measure the benefits of some time optimization techniques for Java code.

Code optimization can be divided into three distinct types, which are based on the needs of the developer: Maintainability, Size and Speed. Maintainability optimization is performed to help make code more manageable in the future. This type of optimization is usually geared toward the structure and organization of code rather than modifications to the algorithms used in the code. Size optimization involves making changes to code that result in a smaller executable class file. The cornerstone of size optimization is code reuse, which comes in the form of inheritance for Java classes. Speed optimization is without a doubt the most important type of optimization when it comes to Java programming. Speed optimization includes all the techniques and tricks used to speed up the execution of code. Considering the performance problems inherent in Java, speed optimization takes on an even more important role in Java than it does in other languages such as C and C++[16].

Literature Review

In (2008) Kevin Williams, etc, writes a paper that presented analysis of existing interpreter optimization techniques on the Cell BE Processor and introduced novel optimizations made possible by the architectural features of the Cell BE SPE [4].

Also in (2008) Huib van den Brink discussed the optimization techniques used in the Java HotSpot Compiler, in order to execute the program code as fast and efficient as possible [5].

In (2010) Peter Sestoft made experiments show that there is no obvious relation between the execution speeds of different software platforms, even for the very simple programs studied here: the C, C# and Java platforms are variously fastest and slowest [8].

In (2012) Pawan Nagar and Nitasha Soni presents a basic framework that allows the application programmers to recognize the constraints of application programs in instruction scheduling [6].

Also in (2012) Hiroshi Inoue and Toshio Nakatani presented a techniques to identify the instructions and objects that frequently cause cache misses without using the HPM of the processor and then showed its effectiveness in compiler optimization using two examples. The key insight is that the cache misses are often caused by pointer dereferences in hot loops in the Java programs [7].

Time optimization techniques for java code

Below, are some time optimization techniques. The description includes the name, the piece of code before optimization and the code after making the optimization.

Technique 1: Use String length to compare empty_string_variables

The String.equals() method is overkill to test for an empty string. It is quicker to test if the length of the string is 0.[9]
Usage Example:
Without optimization
package com.rule;
class Use_String_length_to_compare_empty_string_violation {
    public boolean isEmpty(String str) {
        // Violation
        return str.equals(""};
    }
}

Technique 2: Avoid invoking time-consuming methods in loop
Moving method calls which may take a long time outside of loops can improve performance [10].
Usage Example:
Without optimization
import java.util.Arrays;
public class Test {
    public int[] sortArray(int[] a) {
        for(int i=0; i<100; i++) {
            Arrays.sort(a);  // Violation
            //Some other code
        }
        return a;
    }
}

Technique 3: Avoid empty if
Avoid empty "if" block structure [10].
Usage Example:
Without optimization
package com.rule;
class Avoid_empty_if_violation {
    public void method() {
        final int ZERO = 0;
        int i = 10;
        if (i < ZERO) {  // Violation
            }
        i = ZERO;
    }
}

Technique 4: Avoid unnecessary if
Avoid unnecessary if statements [10].
Usage Example:
Without optimization
package com.rule;
public class Avoid_unnecessary_if_Violation
{
    public void method()
    {
        if (true)      // Violation.
        {
            //Some Code ...
        }
        if (!true)     // Violation.
        {
            //Some Code ...
        }
    }
}

Technique 5: Avoid_unnecessary_parentheses
Avoid unnecessary parentheses in an expression [10].
Usage Example:
Without optimization
package com.rule;
public class Avoid_unnecessary_parentheses_Violation
{
    public void method()
    {
        if((method()))  // Violation.
        {
            //Do Something..
        }
    }
}

Technique 6: Avoid_using_MessageFormat
Avoid using MessageFormat as it is slow [10].
Usage Example:
Without optimization
package com.rule;
import java.text.MessageFormat;
public class Avoid_using_MessageFormat_Violation
{
    public void method()
    {
        final int N = 25000;
        Object argvec[] = new Object[2];
        MessageFormat f = new MessageFormat
        ("The square of {0,number,#} is {1,number,#}");
        for (int i = 1; i <= N; i++)
        {
            argvec[0] = new Integer(i);
            argvec[1] = new Integer(i * i);
            String s = f.format(argvec);
            System.out.println(s);
        }
    }
}
Technique 7: Avoid new with string
Avoid using new with String objects [10][11].
Usage Example:
Without optimization
package com.rule;
class Avoid_new_with_string_violation
{
    public int action(String str)
    {
        String s = new String(str);
        // Violation
        return s.length();
    }
}

With optimization
package com.rule;
class Avoid_new_with_string_correction
{
    public int action(String str)
    {
        String s = str;
        // Correction
        return s.length();
    }
}

Technique 8: Avoid null check before instanceof
Avoid null check before checking instanceof [10].
Usage Example:
Without optimization
package com.rule;
public class Avoid_null_check_before_instanceof_violation
{
    public void method(Object o)
    {
        if(o != null && o instanceof Object)
        // Violation.
        {
            // Do Something.
        }
    }
}

With optimization
package com.rule;
public class Avoid_null_check_before_instanceof_correction
{
    public void method(Object o)
    {
        if(o instanceof Object)
        // Correction
        {
            // Do Something.
        }
    }
}

Technique 9: Do not create instances just to call getClass on it [10][12][13].
Usage Example:
Without optimization
package com.rule;
public class Avoid_instantiation_for_getClass_violation
{
    public void method()
    {
        Class c = (new Avoid_instantiation_for_getClass_violation()).getClass();
        // Violation
    }
}

With optimization
package com.rule;
public class Avoid_instantiation_for_getClass_correction
{
    public void method()
    {
        Class c = Avoid_instantiation_for_getClass_correction.class;
        // Correction
    }
}
Technique 10: Reduce_switch_density

Reduce switch density for performance reasons [10].

Usage Example:
Without optimization
package com.rule;
public class Reduce_switch_density_violation{
    public void method(){
        switch (x) { // Violation.
            case 1:
                if(status)
                    // More Statements
                break;
            case 2:
                // More Statements
                break;
            default:
                
        }
    }
}

With optimization
package com.rule;
public class Reduce_switch_density_correction{
    public void method(){
        switch (x) { // Correction.
            case 1:
                if(status)
                    method1();
                break;
            case 2:
                method2();
                break;
            default:
                i--;
        }
        public method1()
        {  // Do Something.
        }
        public method2()
        {  // Do Something.
        }
    }
}

Technique 11: Avoid_new_Integer_toString

Avoid creating objects of primitive types to call the toString() method instead use valueOf(...) in String class to convert primitive types into their String equivalent [10].

Usage Example:
Without optimization
package com.rule;
class Avoid_new_Integer_toString_violation{
    public void print(){
        String str = new Integer(1).toString();  // Violation
    }
}

With optimization
package com.rule;
class Avoid_new_Integer_toString_correction{
    public void print(){
        String str = String.valueOf(1);  // Correction
    }
}
Technique 12: Avoid_passing_primitive_int_to_Integer_constructor

Avoid creating objects of primitive types using the constructor [5].

Usage Example:
Without optimization
public class Test
{
    public void fubar()
    {
        Integer i = new Integer(3);
        // Violation
        //...
    }
}

With optimization
public class Test
{
    public void fubar()
    {
        Integer i = Integer.valueOf(3); // Fixed
        //...
    }
}

Technique 13: Avoid_passing_primitive_long_to_Long_constructor

Avoid creating objects of primitive types using the constructor [5].

Usage Example:
Without optimization
public class Test
{
    public void fubar()
    {
        Long i = new Long(3);
        // Violation
        //...
    }
}

With optimization
public class Test
{
    public void fubar()
    {
        Long i = Long.valueOf(3); // Fixed
        //...
    }
}

Technique 14: Avoid_passing_primitive_char_to_Character_constructor

Avoid creating objects of primitive types using the constructor [5].

Usage Example:
Without optimization
public class Test
{
    public void fubar()
    {
        Character i = new Character('a');
        // Violation
        //...
    }
}

With optimization
public class Test
{
    public void fubar()
    {
        Character i = Character.valueOf('a'); // Fixed
        //...
    }
}

Technique 15: Avoid_unnecessary_substring

Avoid using String.substring(0) [10].

Usage Example:
Without optimization
package com.rule;
public class Avoid_unnecessary_substring_violation
{
    public void method()
    {
        String str="AppPerfect";
    }
}

With optimization
package com.rule;
public class Avoid_unnecessary_substring_correction
{
    public void method()
    {
        String str="AppPerfect";
        String str1 = str;
        //Correction.
    }
}
String str1 = str.substring(0);  
   // Violation.
   }
   
   }  

Technique 16: Avoid equality with boolean
Avoid comparing a boolean with "true" [10].
Usage Example: 

Without optimization
package com.rule;
   class Avoid_equality_with_boolean_violation
   {
      boolean method(String value)
      {
         boolean b = false;
         String str = "S";
         if (value.endsWith(str) == true)  
            // Violation
            {
               b = true;
            }
            return b;
         }
   }

With optimization
package com.rule;
class Avoid_equality_with_boolean_correction
{
   boolean method(String value)
   {
         boolean b = false;
         String str = "S";
         //........
         if (value.endsWith(str))  
            // Correction
            {
                b = true;
            }
         return b;
   }
}

Technique 17: Avoid instantiation of boolean
Avoid instantiation of Boolean instead use the static constants defined in Boolean class [10].
Usage Example: 

Without optimization
package com.rule;
   class Avoid_instantiation_of_boolean_violation
   {
      public Boolean method()
      {
         Boolean b = new Boolean(true);
         // Violation
         return b;
      }
   }

With optimization
package com.rule;
class Avoid_instantiation_of_boolean_correction
{
   public Boolean method()
   {
      Boolean b = Boolean.TRUE;
      // Correction
      return b;
   }
}

Technique 18: Use single quotes when concatenating character to String
Use single quotes instead of double quotes when concatenating single character to a String [10].
Usage Example: 

Without optimization
package com.rule;
   public class Use_single_quotes_when_concatenating_character_to_String_violation
   {
      Use_single_quotes_when_concatenating_character_to_String_violation()
      {
         String s = "a";
         s = s + "a";  // Violation
      }
   }

With optimization
package com.rule;
public class Use_single_quotes_when_concatenating_character_to_String_correction
{
   Use_single_quotes_when_concatenating_character_to_String_correction()
   {
      String s = "a";
      s = s + 'a';  // Correction
   }
}
Technique 19: Avoid multi-dimensional_arrays
Try to use single dimensional arrays in place of multidimensional arrays [10].
Usage Example:
Without optimization
package com.rule;
    public class Avoid_multi_dimensional_arrays_violation
    {
        public void method1(int[][] values)
        // Violation
            {
                for (int i = 0; i < values.length; i++)
                    {
                        System.out.println(values[i][0] + ":" + values[i][1]);
                    }
        }
    }
    public void method2()
        {
            int[][] arr = new int[][]{[1,2], [2,4], [3,6], [4,8]};
            // Violation
            method1(arr);
        }
    }

Technique 20: Use_String вместо StringBuffer для константных строк
Use String instead of StringBuffer for constant Strings [5].
Usage Example:
Without optimization
package com.rule;
    class Use_String_instead_StringBuffer_for_constant_strings_violation
    {
        public String getSign(int i)
        {
            final String even = new StringBuffer("EVEN"); // violation
            final StringBuffer odd = new StringBuffer("ODD"); // violation
            final StringBuffer msg = new StringBuffer("The number is ");
            if ((i / 2)*i == i)
                {
                    msg.append(even);
                }
            else
                {
                    msg.append(odd);
                }
        return msg.toString();
        }
    }

With optimization
package com.rule;
    class Use_String_instead_StringBuffer_for_constant_strings_correction
    {
        public String getSign(int i)
        {
            final String even = "EVEN"; // correction
            final String odd = "ODD"; // correction
            final StringBuffer msg = new StringBuffer("The number is ");
            if ((i / 2)*i == i)
                {
                    msg = msg.append(even);
                }
            else
                {
                    msg.append(odd);
                }
        return msg.toString();
        }
    }
Technique 21: Avoid creating double from string
Avoid creating double from string for improved performance [10][14].
Usage Example:
Without optimization
public class Avoid_creating_double_from_string_violation
{
    public void method()
    {
        Double db = new Double("3.44"); // Violation
        Double.valueOf("3.44"); // Violation
        if(db == null)
            { // Do Something
                db = null;
            }
    }
}

With optimization
public class Avoid_creating_double_from_string_violation
{
    public void method()
    {
        Double db = new Double(3.44); // Correction
        Double.valueOf(3.44); // Correction
        if(db == null)
            { // Do Something
                db = null;
            }
    }
}

Technique 22: Always use right_shift_operator_for_division_by_powers_of_2
It is more efficient and improves performance if shift operators are used [10].
Usage Example:
Without optimization
public class Test
{
    public int calculate (int num)
    {
        return num / 4; // Violation
    }
}

With optimization
public class Test
{
    public int calculate (int num)
    {
        return num >> 2; // Fixed
    }
}

Technique 23: Use_shift_operators
Shift operators are faster than multiplication and division [10][14].
Usage Example:
Without optimization
package com.rule;
class Use_shift_operators_violation
{
    public void method()
    {
        int x = 0;
        int X = x / 4; // Violation
        int Y = x * 2; // Violation
        X++;
        Y++;
    }
}

With optimization
package com.rule;
class Use_shift_operators_correction
{
    public void method()
    {
        int x = 0;
        int X = x >> 2; // Correction
        int Y = x << 1; // Correction
        X++;
        Y++;
    }
}

Technique 24: Avoid using exponentiation
Do not use exponentiation [10].
Usage Example:
With optimization
package com.rule;
public class Avoid_using_exponentiation_correction
{
Without optimization
package com.rule;
    public class Avoid_using_exponentiation
        {
            public int getPower(int iBase, int iPow)
            {
                int iRet = (int)Math.pow(iBase, iPow);
                // Violation
                return iRet;
            }
        }

Technique 25: Moving Secondary Boolean Operation Outside The For Loop Any operation that contain in for loop and not depend in for loop should be moving outside the loop [16].
Usage Example :
Without optimization
Int y=1;
Int z=1;
for(i=0;i<10;i++)
x=x+y+z   // Violation
With optimization
Int y=1;
Int z=1;
Int t = y + z ;
for(i=0;i<10;i++)
x = x + t   // Correction

Technique 26: Optimize Declarations.
The lower the number of local variables in a function, the better the compiler will be able to fit them into registers. Thus, the compiler will avoid performing pointer frame operations on local variables on the stack. If all local variables are in registers, performance will be better than accessing them from memory. If no local variables are present, the compiler will avoid the overhead of frame pointer set up and restoration. Declare variables with the inner-most scope possible. You will get better performance if local variables are declared only in cases where needed, rather than in every case. Put declarations inside if statements if possible [16].
Usage example :
Without optimization
int foo( int&x )
{
    if ( x > 0 )
    {
        int y = 7;
        return x + y;
    }
    return x;
}
With optimization
int foo( int&x )
{
    int y = 7;
    if ( x > 0 )
    {
        return x + y;
    }
    return x;
}

Technique 27: In for loops, count down rather than up.
When it comes to for loops, count down to zero rather than up if possible. The test against zero is done every iteration and it's faster than any other test [16].
// This...
for( int x = 99; x > 0; --x )
// ...is faster than this
for( int x = 1; x < 100; ++x )
Technique 28: Use operator=, rather than just the operator.
//This...
x += 3;
// ...is faster than this
x = x + 3;
The Experiment: The experiment is done on Lenovo laptop that has CPU 2.30 GH Intel® core ™ i3-2350M, 4GB RAM and 2.92 GB usable. The above computer works on windows 7 operating system. The java netbeans is used for executing java code.

The experiment is as follows:

Every techniques is included in a for loop for hundred million times and the time is obtained before entering the loop and after leaving the loop. The difference between both is considered the execution time, the result is shown in table 1.

Table 1: Comparison Of Execution Time Of Optimization Techniques

<table>
<thead>
<tr>
<th>Techniques name</th>
<th>Time before optimize the code (in millisecond)</th>
<th>Time after optimize the code (in millisecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique 1</td>
<td>874</td>
<td>89</td>
</tr>
<tr>
<td>Technique 2</td>
<td>1818</td>
<td>93</td>
</tr>
<tr>
<td>Technique 3</td>
<td>67</td>
<td>59</td>
</tr>
<tr>
<td>Technique 4</td>
<td>60</td>
<td>59</td>
</tr>
<tr>
<td>Technique 5</td>
<td>67</td>
<td>59</td>
</tr>
<tr>
<td>Technique 6</td>
<td>735</td>
<td>93</td>
</tr>
<tr>
<td>Technique 7</td>
<td>48</td>
<td>2</td>
</tr>
<tr>
<td>Technique 8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Technique 9</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Technique 10</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Technique 11</td>
<td>3752</td>
<td>3146</td>
</tr>
<tr>
<td>Technique 12</td>
<td>992</td>
<td>90</td>
</tr>
<tr>
<td>Technique 13</td>
<td>865</td>
<td>727</td>
</tr>
<tr>
<td>Technique 14</td>
<td>818</td>
<td>135</td>
</tr>
<tr>
<td>Technique 15</td>
<td>50</td>
<td>9</td>
</tr>
<tr>
<td>Technique 16</td>
<td>126</td>
<td>100</td>
</tr>
<tr>
<td>Technique 17</td>
<td>86</td>
<td>11</td>
</tr>
<tr>
<td>Technique 18</td>
<td>1124</td>
<td>960</td>
</tr>
<tr>
<td>Technique 19</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>Technique 20</td>
<td>422</td>
<td>199</td>
</tr>
<tr>
<td>Technique 21</td>
<td>263</td>
<td>135</td>
</tr>
<tr>
<td>Technique 22</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Technique 23</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Technique 24</td>
<td>283</td>
<td>5</td>
</tr>
<tr>
<td>Technique 25</td>
<td>893</td>
<td>892</td>
</tr>
<tr>
<td>Technique 26</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Technique 27</td>
<td>98</td>
<td>97</td>
</tr>
<tr>
<td>Technique 28</td>
<td>85</td>
<td>80</td>
</tr>
</tbody>
</table>

Conclusion and future work

As shown in table 1 the optimization techniques shorten the execution time but in various values. For example techniques 1 shorten the time in approximately divided on nine but techniques 26 only half time. In the future work we recommend to design an automated tool that make optimization without the need for users to go into details of the code. Also, the tool may give the used a recommendation on the strong of the technique.

Reference

[4]. Kevin Williams 1, Albert Noll 2, Andreas Gal 3 and David Gregg1, (2008), “Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine” , Trinity College Dublin, Dublin, Ireland,ETH Zurich, Zurich, Switzerland. 3 University of California, Irvine, CA, USA.
[6]. Pawan Nagar1, Nitasha Soni2, (2012), "Optimizing Program-States using Exception-Handling Constructorin Java ".1M.Tech.Scholar, CSE Department, Lingaya’s University ,Haryana, India ,2Lecturer, CSE Department, Lingaya’s University, Haryana, India, International Journal Of Engineering Science &Advanced Technology,

[7]. Hiroshi Inoue and Toshio Nakatani , (2012) , "Identifying the Sources of Cache Misses in Java Programs Without Relying on Hardware Counters ",© ACM, 2012. This is the author's version of the work.


