International Journal of Enhanced Research in Science Technology & Engineering, ISSN: 2319-7463
Vol. 3 Issue 5, May-2014, pp: (151-162), Impact Factor: 1.252, Available online at: www.erpublications.com

Estimation Benefit of Java Optimization Techniques

AsmaaYaseen 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, which 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("");
	}
}
```

```
package com.rule;
class Use_String_length_to_compare_empty_string
_correction
{
		public boolean isEmpty(String str)
		{
		return str.length()==0;		// Correction
		}
}
```

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;
 }
}
```

With optimization

```
public class Test
{
 public int[] sortArray(int[] a)
{
  Arrays.sort(a);  // Fixed
  for(int i=0; i<100; i++)
 {
//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;
	}
}
```

With optimization

```
package com.rule;
class Avoid_empty_if_correction
{
		public void method()
		{
			final int ZERO = 0;
			int i = 10;
			/*
			if (i < ZERO)		// Correction
			{
			}
			*/
			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 ...
		}
	}

}
```

With optimization

```
package com.rule;
public class Avoid_unnecessary_if_Correction
{
public void method()
		{
boolean flag = true;
		if (flag)		//Correction
			{
		  //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..
		}
	}

}
```

With optimization

```
package com.rule;
public class Avoid_unnecessary_parentheses_correction
{
		public void method()
		{
			if(method())			//Correction.
			{
				// 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);
		}
```

With optimization

```
package com.rule;
public class Avoid_using_MessageFormat_correction
{
		public void method()
		{
			final int N = 25000;
			String s;
			for (int i = 1; i <= N; i++)
			{

		s = "The square of " + i + " is " + (i * i);
		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.clas
s;
 // 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 qoutes 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 inplace of multidimensional arrays [10].

Usage Example:

Without optimization

```
package com.rule;
   public class Aviod_multidimensional_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[][]{
// Violation
                                {1,2},
                                {2,4},
                                {3,6},
                                {4,8}
                };
                method1(arr);
        }
    }
```

```
package com.rule;

public class Aviod_multidimensional_arrays_correction
{
        public void method1(int[] values1, int[] values2) //
Correction
        {
                for (int i = 0; i < values1.length; i++)
                {

        System.out.println(values1[i] + ":" + values2[i]);
                }
        }

        public void method2()
        {

        int[] arr1 = new int[]{1,2,3,4}; // CORRECTION

        int[] arr2 = new int[]{2,4,6,8}; // CORRECTION
                method1(arr1, arr2);
        }
`
```

Technique 20:Use_String_instead_StringBuffer_for_constant_strings

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 StringBuffer 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_strin
gs_correction
{
         public String getSign(int i)
         {
                  final String even = "EVEN";
           // correction
                  final String odd = "ODD";   // correction

         final StringBuffer msg = new StringBuffer("The n
umber 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
_violation
   {
   public int getPower(int iBase, int iPow)
   {
   int iRet = (int) Math.pow(iBase, iPow);
 // Violation
                    return iRet;
          }
   }
```

```
public int getPower(int iBase, int iPow)
          {
                    int iRet = 1;
                    for (int i = 0; i < iPow; i++) // Correction
                    {
                              iRet *= iBase;
                    }
                    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 a 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

Techniques name	Time before optimize the code (in millisecond)	Time after optimize the code(in millisecond)
Technique 1	874	89
Technique 2	1818	93
Technique 3	67	59
Technique 4	60	59
Technique 5	67	59
Technique 6	735	93
Technique 7	48	2
Technique 8	4	3
Technique 9	9	2
Technique 10	5	4
Technique 11	3752	3146
Technique 12	992	90
Technique 13	865	727
Technique 14	818	135
Technique 15	50	9
Technique 16	126	100
Technique 17	86	11
Technique 18	1124	960
Technique 19	22	21
Technique20	422	199
Technique 21	263	135
Technique 22	7	2
Technique 23	2	1
Technique 24	283	5
Technique 25	893	892
Technique 26	2	1
Technique 27	98	97
Technique 28	85	80

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 approximelity 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

[1]. Michael Dorf ,(2012) , "5 Easy Java Optimization Tips " ,http://www.learncomputer.com/java-optimization-tips .
[2]. IBM ,(2009) , "Optimizing C code atoptimization level 2 " ,Copyright International Business Machines Corporation 2009.
[3]. Maggie Johnson,(2008) ,"Code Optimization",Handout 20.
[4]. Kevin Williams[1],Albert Noll[2],Andreas Gal[3] and David Gregg[1] ,(2008) , "Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine"[1]Trinity College Dublin, Dublin, Ireland,[2]ETH Zurich, Zurich, Switzerland. [3]University of California, Irvine, CA, USA.
[5]. Huib van den Brink ,(2008), "The current and future optimizations performed by the Java HotSpotCompiler" ,Institute of Information and Computing Sciences, Utrecht UniversityP.O. Box 80.089, 3508 TB Utrecht, The Netherlands.

[6]. Pawan Nagar[1],Nitasha Soni[2], (2012)," Optimizing Program-States using Exception-Handling Constructerin Java ",[1]M.Tech.Scholar, CSE Department, Lingaya's University ,Haryana, India ,[2]Lecturer, 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.
[8]. Peter Sestoft ,(2010) ,"Numeric performance in C, C# and Java",IT University of CopenhagenDenmark,Version 0.9.1 of 2010-02-19.
[9]. http://www.onjava.com/pub/a/onjava/2002/03/20/optimization.html?page=4 http://www.javaperformancetuning.com/tips/rawtips.shtml.
[10]. http://www.appperfect.com/support/java-coding-rules/optimization.html.
[11]. Tony Sintes ,(2002) ,"The String class's strange behavior explained",http://www.javaworld.com/article/2077355/core-java/don-t-be-strung-along.html.
[12]. IBM ,(2013), "Migrating to Java 2 Standard Edition (J2SE) 5",http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Class.html ,
[13]. Ont Community ,(2012),"About .class operator ",Oracle.
[14]. Technology Specialist, (2012), "One 4 All", http://www.javaperformancetuning.com/tips/rawtips.shtml, ,http://www.glenmccl.com/jperf/.
[15]. Felix Hernandez-Campos ,(2002) ,"COMP 144 Programming Language Concepts" ,The University of North Carolina at Chapel Hill.Ben Van Vliet ,(2008),"C++FA 3.1 OPTIMIZING C++", http://www.benvanvliet.net/Downloads/CFA3.1_Optimizing%20CPP.
[16]. Guihot, H. (2012). Optimizing Java Code. Pro Android Apps Performance Optimization, Springer**:** 1-31.