Academic Journal of Nawroz University (AJNU), Vol.10, No.4, 2021

https://doi.org/10.25007/ajnu.v10n1a709

Performance Evaluation of Java Programming Strategies

Qusay Idrees Sarhan

Department of Computer Science, College of Science, University of Duhok, Duhok, Kurdistan Region, Iraq

ABSTRACT

Java is one of the most demanding programming languages nowadays and it is used for developing a wide range of software applications including desktop, mobile, embedded, and web applications. Writing efficient Java codes for those various types of applications (which some are critical and time-sensitive) is crucial and recommended best practices that every Java developer should consider. To date, there is a lack of in-depth experimental studies in the literature that evaluate the impact of writing efficient Java programming strategies on the performance of desktop applications in terms of runtime. Thus, this paper aims to perform a variety of experimental tests that have been carefully chosen and implemented to evaluate the most important aspects of desktop efficient Java programming in terms of runtime. The results of this study show that significant performance improvements can be achieved by applying different programming strategies.

Keywords: Java programming language, Java programming strategies, Java performance, Java optimization strategies, experimental evaluation, performance measurement.

1. Introduction

Java programming language is massively used in the development of various software applications including desktop, mobile, embedded, and web applications [1]. These applications cover robotics, safety and security, e-health care, smart homes, Internet of Things (IoT)-based products, and real-time industrial control systems [2-5]. Therefore, it is crucial to efficiently program these applications especially when the aim is to develop critical and time-sensitive applications where each microsecond matters. Besides, writing efficient Java code is recommended best practices that every Java developer should consider. To increase the performance of Java-based applications in terms of execution time, different programming strategies can be used. Writing efficient code is the process of modifying the existing code of an application to improve its performance. In other words, to make it use fewer resources, to reduce its size, to consume lesser power, to reduce compilation time, or to execute fast [6]. This is performed without changing the semantics and outputs of the application wanted to be optimized for better performance [7]. This paper experimentally analyzes the performance impact of using various programming strategies for the efficient writing of Java-based applications. It therefore aims at and contributes to the following: (a) To provide a set of experimental tests that were designed and implemented to enable the performance measurement and analysis of various Java programming strategies. As a result, Java developers can write efficient code directly, based on the obtained results. (b) To prove that even the Java compiler is designed to automatically provide code optimization, is not always enough to achieve the best performance. Java developers therefore can have a notable role in this regard by manually applying various programming strategies that achieve better performance. (c) To show that there are no standard rules to follow in order to optimize the codes of a Java-based application. Thus, Java developers should try to optimize their codes by practicing and experimenting with different programming strategies to determine which strategy delivers the best performance in comparison with others.

The remainder of this paper is organized as follows: Section 2 briefly presents the related work of this study. Section 3 describes in detail the testing methodology used to conduct this experimental study.

Section 4 presents the experimental results and outcomes of this study. Finally, the conclusions of this study and some possible future works are provided in Section 5.

2. RELATED WORK

The most related works published on the topic of this study are briefly presented here. The authors in [8] proposed a few numbers of programming strategies for reducing the Java compiler runtime overhead and for improving the Java code quality. However, all the proposed strategies were not implemented and evaluated experimentally. The authors in [9] presented and suggested to use several efficient Java programming strategies. However, all of the suggested strategies were not evaluated experimentally to really know which strategy outperforms the others and why. The authors in [10] and [11] discussed the benefits of optimizing Java bytecode (the compiled format of Java source code) by presenting the results of a few tests. However, these two studies are dedicated to optimizing the Java compiled code rather than Java source code. The author in [12] proposed an optimization tool for Java programmers. The tool has the ability to automatically check Java codes that need to be optimized. Further, they showed that different optimization issues can be detected by the proposed tool. However, the study is not dedicated to measuring the performance of programming strategies in terms of execution time. The authors in [13] and [14] provided a brief explanation of writing efficient code for Java embedded and real-time systems. Also, they presented a few programming strategies that lead to high performance in such systems. However, the authors did not conduct any experimental tests to show the performance impact of their strategies. The authors in [15] presented an overview of code optimization and its techniques and provided a general optimization guideline on how to optimize code for microcontrollers and embedded systems regardless of the used programming languages. However, the authors did not conduct any experimental evaluation to show the benefits of this guideline. Most of the aforementioned studies did not conduct in-depth experimental evaluations for their programming strategies. Many others did not present how they conducted their test methodologies. In addition, there were no details about how they measured the efficiency of the used programming strategies and they did not provide their software and hardware testing specifications. In contrast, this comprehensive study provides twenty five different programming strategies as experimental test scenarios that were developed and implemented to reduce the execution time of Java-based applications. Also, it provides a well-defined test methodology that can be employed by other developers and researchers to obtain similar results if they perform/duplicate this experimental study.

3. TESTING METHODOLOGY

The testing methodology followed in this study includes test conditions, test scenarios (each with several programming strategies), test metrics, and testbed setup to experimentally evaluate the effect of using different programming strategies on the performance of Java-based applications in terms of execution time.

3.1 Test Conditions

In this study, a number of test conditions were considered as follows:

- All test scenarios were implemented using the same programming language which is Java and the same IDE software which is Apache NetBeans with their default configurations and settings. Also, they were executed on the same hardware system.
- Every test scenario has been programmed and executed with the same parameters and data values.
- No extra processing has been performed in each test scenario in order to measure the performance precisely.

- For each test scenario, at least two tests were implemented: (a) A test with an unoptimized code. (b) A test with an optimized code (code after applying the corresponding efficient changes).
- The performance of each programming strategy was measured for the execution time/runtime (the time required to execute a programming strategy).
- Before conducting the experiments and measurements, all user programs, excluding the used IDE, were closed.
- The Internet was switched off from the computer while evaluating the scenarios.
- Each test scenario was executed 1000 times; afterwards, the average execution time has been taken for more precision.
- The execution time of each programming strategy was obtained by implementing the pseudo code shown below.

Pseudo code of execution time measurement:

i. **Begin**
ii. get start time
iii. execute a code strategy
iv. get end time
v. elapsed time = (end time - start time)
vi. **End**

3.2 Test Scenarios

Efficient programming strategies are transformations applied to software codes to decrease their execution times. In this experimental study, twenty five programming strategies were employed to perform the experimental evaluations and analysis. It is worth mentioning that these strategies were selected because they can be used to perform a wide variety of programming tasks, with a notable effect on Java-based applications performance. Moreover, both inexperienced and experienced Java developers can use them in their software development process.

3.3 Test Metrics

The execution time was used as a metric to experimentally assess and compare the performance of each programming strategy. Thus, any programming strategy of a specific test scenario requires less time to be executed is considered as the best strategy and outperforms the others.

3.4 Testbed Setup

This study's test scenarios were implemented and executed using the software and hardware presented in Table 1 with their specifications.

Table 1: Software and hardware specifications

	Specifications	Detail
Software	Apache NetBeans IDE	12.0
	Java Development Kit (JDK)	12.0.1
	Windows OS	Windows 10 (64-bit)
Hardware	Model	HP Elitebook Revolve 810
	CPU Type	Intel Core i5-4300U
	CPU Speed	2.5 GHz
	CPU Cores	4
	RAM	8 GB

4. EXPERIMENTAL RESULTS AND DISCUSSION

This section presents and discusses the experimental results of this study.

4.1 Newline printing

Three programming strategies have been used to print a newline to the input/output screen as presented in Table 2. In strategy 1 and 2, each prints two characters '\r' and '\n' which is a carriage return character followed by a line feed character to the screen. In strategy 3, the lineSeparator() method of the class System is invoked to print a newline to the screen. The experimental results show that strategy 3 requires more time to be executed compared to strategy 1 and 2. The reason for this is that strategy 3 is a system-dependent newline printer which means before printing a newline to the screen, it checks the used operating system. If the operating system is UNIX-based system, it prints "\n" only; on Microsoft Windows systems it prints both '\r' and '\n'. Thus, this checking process consumes time.

Table 2 : Strategies to print newline

	Strategy Implementation	Execution Time (µs)
Strategy 1	System.out.println("Java");	0.005
Strategy 2	System.out.print("Java" + "\r\n");	0.005
Strategy 3	System.out.print("Java" + System.lineSeparator());	0.008

4.2 Strings concatenation

Two programming strategies have been used to concatenate string variables as presented in Table 3. In strategy 1, the concatenation is performed by using the concat() function of the class String. In strategy 2, the concatenation is performed by using the addition operator +. The experimental results show that strategy 2 requires more time to be executed compared to strategy 1. The reason for this is that strategy 2 accepts string and non-string arguments thus it implicitly converts all its arguments to strings then performs the concatenation, whereas strategy 1 does not apply any conversion as it only accepts string argument which saves time. It is worth mentioning that the concat() function takes only one string argument and concatenates it with another string, thus each time only two strings can be concatenated. Whereas the addition operator + takes any number of arguments and concatenates them all.

Table 3: Strategies to perform string concatenation

	Strategy Implementation	Execution Time (µs)
Strategy 1	String str1="12"; String str2="34"; String str3=str1.concat(str2); System.out.println(str3);	0.0049
Strategy 2	String str1="12"; String str2="34"; String str3 = str1 + str2; System.out.println(str3);	0.0055

4.3 String to char array conversion

Two programming strategies have been used to convert a string to a char array as presented in Table 4. In strategy 1, the built-in function toCharArray() is used to perform the required conversion. In strategy 2, a for-loop is used to copy the contents of the same string to a char array element by element. The experimental results show that strategy 2 requires more time to be executed compared to strategy 1. The reason for this is that strategy 2 employs more statements including the for-loop to perform the conversion.

Table 4: Strategies to perform string to char array conversion

	Strategy Implementation	Execution Time (µs)

Strategy 1	String str="Java";
	int len=str.length();
	char temp []=str.toCharArray();
	for(byte i=0;i<len;i++)
	System.out.println(temp[i]);
		0.0183

Strategy 2	String str="Java";
	int len=str.length();
	char temp[]=new char[len];
	for (byte i = 0; i < len; i++)
	temp[i] = str.charAt(i);
	for(byte i=0;i<len;i++)
	System.out.println(temp[i]);
		0.0260

4.4 Char array to string conversion

Three programming strategies have been used to convert a char array to a string as presented in Table 5. In strategy 1, the built-in function valueOf() of the String class with the name of the char array as an argument is used to perform the conversion. In strategy 2, each element in the char array is converted to a string and merged using the addition operator + inside a for-loop. In strategy 3, iteration over the elements of the char array and append each element to the StringBuilder is applied. The experimental results show that strategy 2 and 3 require more time to be executed compared to strategy 1. The reason for this is that they both use a for-loop to iterate over the elements of the char array. Besides, they use additional statements to perform the conversion.

Table 5: Strategies to perform char array to string conversion

	Strategy Implementation	Execution Time (µs)
Strategy 1	char x[]={'J','a','v','a'}; String s=String.valueOf(x); System.out.println(s);	0.0050

Strategy 2	char x[]={'J','a','v','a'};
	String s="";
	for (byte i=0;i<x.length;i++)
	s=s+x[i];
		0.0096

	System.out.println(s);	

Strategy 3	char x[]={'J','a','v','a'};
	StringBuilder strbld=new StringBuilder();
	for (byte i=0;i<x.length;i++)
	strbld.append(x[i]);
	String s=strbld.toString();
	System.out.println(s);
		0.0104

4.5 Matching word in a string

Three programming strategies have been used to match a given text/word in a string as presented in Table 6. In strategy 1, the substring() function is used to perform the matching operation via a string-to-string comparison. In strategy 2, the charAt() function is used to perform the matching operation via a char-to-char comparison. In strategy 3, regular expressions are used to perform the matching operation. The experimental results show that strategy 2 and 3 require more time to be executed compared to strategy 1. The reason for this is that strategy 2 uses a for-loop to iterate over the elements of the string char by char. Besides, it uses additional statements to finish the matching process. Strategy 3 also implicitly performs a lot of comparisons to find the required word.

Table 6: Strategies to match word in a string

	Strategy Implementation	**Execution Time (µs)**

Strategy 1	String sentence="Java is a programming language";
	String word="programming";
	boolean result=false;
	int wordLen=word.length();
	int diffLen=sentence.length()-wordLen;
	for(int i=0;i<=diffLen;i++)
	
	if(sentence.substring(i,wordLen+i)==word)
	{
	result=true;
	break;
	}
	System.out.println(result);
		0.0060

Strategy 2	String sentence="Java is a programming language";
	String word="programming";
	boolean result=false;
	int count=0,i=0;
	while(i!=sentence.length())
	{
	
	if(sentence.charAt(i)==word.charAt(count))
	{
	count++;
	if(count==word.length())
	{
	result=true;
	break;
	}
	}
	else
	if(count!=0)
	{
	i--;
	count=0;
	}
	i=i+1;
	}
	System.out.println(result);
		0.0103

Strategy 3	String sentence="Java is a programming language";
	String word="programming";
	boolean result=false;
	result=sentence.matches("(.*)"+word+"(.*)");
	System.out.println(result);
		0.0141

4.6 Evaluating fixed values

Many applications evaluate a set of fixed values such as ports sizes (SMALL, MEDIUM, LARGE), days of the week (SATURDAY, SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY), and directions (NORTH, EAST, SOUTH, WEST). There are two programming strategies to evaluate such sets of values as presented in Table 7. In strategy 1, byte values are used to represent such sets of fixed values. Whereas string values are used to represent such sets of fixed values in strategy 2. The experimental results show that strategy 2 requires more time to be executed compared to strategy 1. The reason for this is that performing string comparison using equals() function decreases the performance because the compiler needs to compare a sequence of characters in each string.

Table 7: Strategies to evaluate a set of fixed values

	Strategy Implementation	**Execution Time (µs)**

Strategy 1	byte SMALL = 1, MEDIUM = 2, LARGE = 3;
	byte selectedSize = 2;
	boolean result=false;
	if (selectedSize == MEDIUM)
	result=true;
	System.out.println(result);
		0.0053

Strategy 2	String SMALL = "SMALL", MEDIUM = "MEDIUM", LARGE = "LARGE"; boolean result=false; String selectedSize = "MEDIUM"; if (selectedSize.equals(MEDIUM)) result=true; System.out.println(result);	0.0086

4.7 Using break statement in sequential search

Searching for a value in an array can be performed either using break statement or without it as presented in Table 8. In strategy 1, the process starts with comparing the value to be found with the values in the defined array from the beginning and when the required value is found, break statement is used to stop the search process. Using break statement is very useful as there is no need to iterate over all the values of the array. In strategy 2, the search process iterates over all the values in the defined array searching for the specified value even if that value has been found at the beginning of the array. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time.

Table 8: Strategies to perform sequential search

	Strategy Implementation	**Execution Time (µs)**
Strategy 1	boolean result=false; String list[]= { "item1", "item2", "item3", "item4", "item5" }; for(byte i=0;i<list.length;i++) if(list[i]=="item3") { result=true; break; } System.out.println(result);	0.0056
Strategy 2	boolean result=false; String list[]= { "item1", "item2", "item3", "item4", "item5" }; for(byte i=0;i<5;i++) if(list[i]=="item3") result=true; System.out.println(result);	0.0078

4.8 Global vs. local variables

Variables can be defined as global or private based on how they are planned to be accesses and changed as presented in Table 9. In strategy 1, both variables sum and a[] are defined as global; thus, they can be accessed from anywhere in the program. Whereas in strategy 2, both sum and a[] are local variables as they are defined inside the function summation(). Therefore, both variables are accessed only within that function. To ensure this, the compiler applies some mechanisms to prevent local variables from being accessed from outside the function that defines them. Therefore, accessing local variables takes more time. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time.

Table 9: Strategies to use variables: global vs. local

	Strategy Implementation	**Execution Time (µs)**
Strategy 1	static int sum = 0; static int a[ ] = {5, 10, 15, 20, 25, 30, 35,40}; void summation() { for(byte i=0;i<a.length;i++) sum=sum+a[i]; System.out.println(sum); }	0.0066
Strategy 2	void summation() { int sum = 0; int a[ ] = {5, 10, 15, 20, 25, 30, 35,40}; for(byte i=0;i<a.length;i++) sum=sum+a[i]; System.out.println(sum); }	0.0098

4.9 1D array initialization

Two programming strategies have been used to initialize a 1D array with a given value as presented in Table 10. In strategy 1, the fill() function of Array class is used to initialize the array a[100] with the value 2. Whereas in strategy 2, the array a[100] is initialized with the value 2 by using two for-loop statements. Using for-loop statements takes more time to be executed as they perform many increment ++ and comparison < operations. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time.

Table 10: Strategies to initialize a 1D array

	Strategy Implementation	Execution Time (µs)

Strategy 1	int a[]=new int [100];
	int sum=0;
	Arrays.fill(a, 2);
	for(int i=0;i<a.length;i++)
	sum=sum+a[i];
	System.out.println(sum);
		0.0052

Strategy 2	int a[]=new int [100];
	int sum=0;
	for(int i=0;i<a.length;i++)
	a[i]=2;
	for(int i=0;i<a.length;i++)
	sum=sum+a[i];
	System.out.println(sum);
		0.008

4.10 Row-major order vs. column-major order

Two programming strategies have been used to write data into a 2D array as presented in Table 11. In strategy 1, the array a[][] is initialized with the value 2 via row-major order (data is written row by row). In strategy 2, the array a[][] is initialized with the value 2 via column-major order (data is written column by column). The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that arrays in Java are stored in row-major order. That means consecutive elements of a row are contiguous in memory and so it is faster to read memory at contiguous locations. Consequently, if the array is stored in row-major order, then iterating sequentially through its elements in row-major order is faster than iterating through its elements in column-major order.

Table 11: Strategies to perform row-major order and column-major order

	Strategy Implementation	Execution Time (µs)

Strategy 1	int size=100, sum=0;
	int a[][]=new int [size][size];
	for(int i=0; i<size; i++)
	for(int j=0; j<size; j++)
	a[i][j]=2 ;
	for(int i=0; i<size; i++)
	for(int j=0; j<size; j++)
	sum=sum+a[i][j];
	System.out.println(sum);
		0.0215

Strategy 2	int size=100, sum=0;
	int a[][]=new int [size][size];
	for(int i=0; i<size; i++)
	for(int j=0; j<size; j++)
	a[j][i]=2 ;
	for(int i=0; i<size; i++)
	for(int j=0; j<size; j++)
	sum=sum+a[j][i];
	System.out.println(sum);
		0.0356

4.11 Function calling overhead

Calling a function to execute a particular task always has a time overhead. Two programming strategies have been used to call a function several times as presented in Table 12. If a function is to be called multiple times, inexperienced developers frequently repeat the calling statement as in strategy 2. On the other hand, a for-loop with an iteration number equal to how many times that function wanted to be executed as in Strategy 1 can be put inside the function. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that strategy 2 increases the overhead in each function calling (each call requires extra time).

Table 12: Strategies to call a function several times

	Strategy Implementation	Execution Time (µs)

Strategy 1	void printString(String str)
	{
	for(int i=0;i<3;i++)
	System.out.println(str);
	}
	// calling the function one time
	printString("Testing");
		0.0135

Strategy 2	void printString(String str)
	{
	System.out.println(str);
	}
	// calling the function three times
	for(int i=0;i<3;i++)
	printString("Testing");
		0.0197

4.12 Loop rolling vs. loop unrolling

This experimental test involves initializing the integer array a[99] to the value 5 in a loop then calculating the summation of the elements in another loop as presented in Table 13. Strategy 1 unrolls both loops and performs three assignment operations in each iteration of the first loop and performs three summation operations in each iteration of the second loop. Strategy 2 traverses both loops step by step and performs one assignment operation in each iteration of the first loop and performs one summation operation in each iteration of the second loop. The experimental results show that loop unrolling in strategy 1 requires less execution time compared to strategy 2 which uses loop rolling. The reason for this is that loop unrolling reduces the number of iterations, comparisons, and repeating the loop body several times.

Table 13: Strategies to perform looping: loop rolling vs. loop unrolling

	Strategy Implementation	**Execution Time (µs)**

Strategy 1	int sum=0;
	int a[]=new int[99];
	for(int i=0;i<99;i+=3)
	{
	a[i]=5;
	a[i+1]=5;
	a[i+2]=5;
	}
	for(int i=0;i<99;i+=3)
	sum=sum+a[i]+a[i+1]+a[i+2];
	System.out.println(sum);
		0.005

Strategy 2	int sum=0;
	int a[]=new int[99];
	for(int i=0;i<99;i++)
	a[i]=5;
	for(int i=0;i<99;i++)
	sum=sum+a[i];
	System.out.println(sum);
		0.0076

4.13 Data types selection

Using the correct data types for programming tasks has a notable impact on program performance. The byte data type is used in strategy 1 to declare, initialize, and then calculate the summation of the array a[100]. The same task is carried out in strategy 2 but with the use of the int data type as presented in Table 14. The experimental results show that the extra size of the int data type takes more time to be processed compared to using the byte data type. It is recommended to check the data variable range to be used for the programming task at hand, and then a suitable data type that fits the required range has to be chosen and used accordingly.

Table 14: Strategies to use different data types

	Strategy Implementation	**Execution Time (µs)**

Strategy 1	byte sum=0;
	byte a[]=new byte[100];
	for(byte i=0;i<a.length;i++)
	a[i]=1;
	for(byte i=0;i<a.length;i++)
	sum= (byte)(sum+a[i]);
	System.out.println(sum);
		0.0059

Strategy 2	int sum=0;
	int a[]=new int[100];
	for(int i=0;i<a.length;i++)
	a[i]=1;
	for(int i=0;i<a.length;i++)
	sum= sum+a[i];
	System.out.println(sum);
		0.0087

4.14 For-loop vs. iterator

Iterating over the elements of a list can be performed using standard for-loop or iterator as presented in Table 15. Strategy 1 iterates over the elements of a list of size 100 using a standard for-loop whereas strategy 2 iterates over the same elements using an iterator. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that the used iterator in strategy 2 creates a new

String instance for each element in the list (in this test, a String instance will be created 100 times in order to iterate over all elements of the used list) which consumes time.

Table 15: Strategies to perform looping: for-loop vs. iterator

	Strategy Implementation	Execution Time (µs)

Strategy 1	int listSize=100;
	List<String> list = new ArrayList<>(listSize);
	for(int i=0;i<listSize;i++)
	list.add("Java");
	for(int i=0;i<listSize;i++)
	System.out.println(list.get(i));
		0.3535

Strategy 2	int listSize=100;
	List<String> list = new ArrayList<>(listSize);
	for(int i=0;i<listSize;i++)
	list.add("Java");
	for(String value: list)
	System.out.println(value);
		0.4287

4.15 Checking empty strings

Two programming strategies have been used to check if a string object is empty or not as presented in Table 16. In strategy 1, the length() function is used to check if the object str is empty or not by checking its length. If the length is equal to 0, then it means the object str is empty. Here, an int-to-int comparison is employed. In strategy 2, the checking operation is performed using the equals() function which performs a string-to-string comparison. The experimental results show that strategy 2 requires more time to be executed compared to strategy 1. The reason for this is that strings (objects) comparison consumes more time than integers (variables) comparison.

Table 16: Strategies to check empty strings

	Strategy Implementation	Execution Time (µs)

Strategy 1	String str = new String();
	boolean result=false;
	if(str.length()==0)
	result=true;
	System.out.println(result);
		0.005

Strategy 2	String str = new String();
	boolean result=false;
	if(str.equals(""))
	result=true;
	System.out.println(result);
		0.007

4.16 Pausing program execution

Two programming strategies have been used to pause a program execution for a given period of time as presented in Table 17. In strategy 1, the sleep() function of the class Thread is used whereas the sleep() function of the class TimeUnit is used in strategy 2. It is worth mentioning that during the pausing period of strategy1, other tasks can be performed (e.g., I/O pins manipulations and mathematical calculations). Thus, it is very useful to perform multi-task operations. During the pausing period of strategy2, other tasks cannot be performed. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that strategy 2 uses specific mechanisms to not make the processor time available to the other tasks of a program to be executed during the pausing period. Thus, doing so consumes more time compared to strategy 1.

Table 17: Strategies to pause program execution

	Strategy Implementation	Execution Time (µs)

Strategy 1	int timePeriod = 1, x=50;
	x=x+250;
	System.out.println(x);
	Thread.sleep(timePeriod*1000);
	x=x+50;
	System.out.println(x);
		995.9022

Strategy 2	int timePeriod = 1, x=50;
	x=x+250;
	System.out.println(x);
	TimeUnit.SECONDS.sleep(timePeriod);
	x=x+50;
	System.out.println(x);
		1000.1927

4.17 Eliminating identical expressions

Elimination of identical expressions aims to replace identical expressions that evaluate the same value several times with a single variable holding the same computed value as presented in Table 18. In strategy 2, a number of mathematical operations are computed

including the identical expression (a*b) repeated three times. In strategy 2, the identical expression (a*b) is replaced with the variable temp instead of repeating the same operation three times. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that the time required to calculate (a*b) only one time with the time to store and retrieve temp is less than the time required for calculating (a*b) several times.

Table 18: Strategies to eliminate identical expressions

	Strategy Implementation	Execution Time (µs)

Strategy 1	int x, y, a=5, b=5, i=100, j=100, k=1000;
	double z;
	int temp=a*b;
	x = i + temp;
	y = j * temp;
	z = k / temp;
	System.out.println(x + " "+y+" "+z);
		0.0499

Strategy 2	int x, y, a=5, b=5, i=100, j=100, k=1000;
	double z;
	x = i + (a * b);
	y = j * (a * b);
	z = k / (a * b);
	System.out.println(x + " "+y+" "+z);
		0.0706

4.18 Using shift operator

Two programming strategies have been used to perform multiplication as presented in Table 19. In strategy 1, each element in the array a[] is multiplied by 2 using the shift operator a[i]<<1; then the summation of all elements is calculated. In strategy 2, the same multiplication process is performed but by using the standard multiplication * sign. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that the shift operator is faster because it works on the bits level and thus directly supported by the processor. To perform the division operation by 2, the shift operator a[i]>>1can by used.

Table 19: Strategies to perform calculations with/without shift operator

	Strategy Implementation	Execution Time (µs)

Strategy 1	int sum=0;
	int a[]=new int[100];
	for(int i=0;i<a.length;i++)
	a[i]=5;
	for(int i=0;i<a.length;i++)
	a[i]=a[i]<<1;
	for(byte i=0;i<a.length;i++)
	sum= sum+a[i];
	System.out.println(sum);
		0.0306

Strategy 2	int sum=0;
	int a[]=new int[100];
	for(int i=0;i<a.length;i++)
	a[i]=5;
	for(int i=0;i<a.length;i++)
	a[i]=a[i]*2;
	for(byte i=0;i<a.length;i++)
	sum= sum+a[i];
	System.out.println(sum);
		0.0559

4.19 Calculations outside loops

Two programming strategies have been used to perform calculations either inside a loop or outside a loop as presented in Table 20. In strategy 1, the expression (Math.pow(2, x)+Math.pow(4, x);) is calculated outside the second for-loop. Then, its calculated value which is stored in the variable y has been used inside the second loop. In strategy 2, the same expression is calculated inside the second loop only. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that the expression is only calculated and executed one time if it is outside the loop rather than being calculated and executed on each loop iteration if it is inside the loop.

4.20 For looping

Two programming strategies have been used to perform for looping as presented in Table 21. In strategy 1, the loop variable i goes from a.length-1 to 0, then i is compared against 0 at each iteration. In strategy 2, the loop variable i goes from 0 to a.length-1, then i is compared against a.length at each iteration. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that comparing against 0 is more efficient in any programming language because the underlying tests are based on < 0, <= 0, == 0, != 0, >= 0 and > 0.

Table 20: Strategies to perform calculations inside/outside loops

	Strategy Implementation	Execution Time (µs)

Strategy 1	double a[]=new double[100];
	int x=2;
	for(int i=0;i<a.length;i++)
	a[i]=x;
	double y=Math.pow(2, x)+ Math.pow(4, x);
	for(int i=0;i<a.length;i++)
	a[i]=a[i]* y;
	for(int i=0;i<a.length;i++)
	System.out.println(a[i]);
		0.8551

Strategy 2	double a[]=new double[100];
	int x=2;
	for(int i=0;i<a.length;i++)
	a[i]=x;
	for(int i=0;i<a.length;i++)
	a[i]=a[i]* (Math.pow(2, x)+Math.pow(4, x));
	for(int i=0;i<a.length;i++)
	System.out.println(a[i]);
		1.6562

Table 21: Strategies to perform for looping

	Strategy Implementation	Execution Time (µs)

Strategy 1	int sum=0;
	int a[]=new int[100];
	for(int i=a.length-1;i-->=0;)
	a[i]=2;
	for(int i=a.length-1;i-->=0;)
	sum= sum+a[i];
	System.out.println(sum);
		0.0258

Strategy 2	int sum=0;
	int a[]=new int[100];
	for(int i=0;i<a.length;i++)
	a[i]=2;
	for(int i=0;i<a.length;i++)
	sum= sum+a[i];
	System.out.println(sum);
		0.0487

4.21 Accessing class functions

Two programming strategies have been used to access the functions of a class as presented in Table 22. In strategy 1, the function printString() is defined as a static function by using the static keyword preceding its name. To call this static function, the name of the class (JavaApplication1) containing the function is directly used in the function calling. In strategy 2, the function printString() is defined as a normal function without the static keyword. To call this non-static function, the object obj of the class (JavaApplication1) containing the required function is created and then is used to call the function. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that strategy 2 includes an extra step which is object creation that consumes more time compared to strategy 1.

Table 22: Strategies to access class functions

	Strategy Implementation	Execution Time (µs)

Strategy 1	class JavaApplication1
	{
	static String printString(String str)
	{
	return str.toUpperCase();
	}}
	String str=JavaApplication1.printString("Java");
	System.out.println(str);
		0.0241

Strategy 2	class JavaApplication1
	{
	String printString(String str)
	{
	return str.toUpperCase();
	}}
	JavaApplication1 obj=new JavaApplication1();
	String str=obj.printString("Java");
	System.out.println(str);
		0.044

4.22 Initializing collections

Two programming strategies have been used to initialize collections of type Set as presented in Table 23. In strategy 1, the collection set is initialized with Item1, Item2, and Item3 via the constructor of the class Set. In strategy 2, the collection set is initialized with Item1, Item2, and Item3 via the function addAll(). The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that strategy 2 includes an extra step which is calling the function addAll() of the object set that consumes more time compared to strategy 1.

Table 23: Strategies to initialize collections

	Strategy Implementation	Execution Time (µs)
Strategy 1	Set<String> set = new HashSet<>(Arrays.asList("Item1", "Item2", "Item3")); for(String value: set) System.out.println(value);	0.0723

Strategy 2	Set<String> set = new HashSet<>();
	set.addAll(Arrays.asList("Item1", "Item2", "Item3"));
	for(String value: set)
	System.out.println(value);
		0.1392

4.23 Adding elements to a list

Two programming strategies have been used to add a number of elements to a list as presented in Table 24. In strategy 1, the addAll() function of the class List is used to add 100 elements to the created list. It can be noted that the input to the addAll() function is an array of size 100. The addAll() function stores all elements of the array into the created list at once. In strategy 2, the add() function of the class List is used to add 100 elements to the created list. It can be noted that the add() function is put inside a loop of 100 iterations. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that strategy 2 calls the add() function 100 times which consumes more time compared to use an array of size 100 for storing the elements and then calling the addAll() function only one time to add all the elements at once.

Table 24: Strategies to add elements to a list

	Strategy Implementation	Execution Time (µs)

Strategy 1	int listSize=100;
	String str[]=new String[100];
	List<String> list = new ArrayList<>(listSize);
	for(int i = 0; i < listSize; i++)
	str[i]="Java";
	list.addAll(Arrays.asList(str));
	for(int i = 0; i < listSize; i++)
	System.out.println(list.get(i));
		1.2432

Strategy 2	int listSize=100;
	List<String> list = new ArrayList<>(listSize);
	for(int i = 0; i < listSize; i++)
	list.add("Java");
	for(int i = 0; i < listSize; i++)
	System.out.println(list.get(i));
		2.1347

4.24 Reading a file line by line

Several programming strategies have been used to read a text file line by line as presented in Table 25. In strategy 1, the readLine() function of the class BufferedReader is used inside a loop to read the specified file Test.txt line by line. In strategy 2, the readLine() function of the class RandomAccessFile is used inside a loop to read the same file line by line. In strategy 3, the lines() function of the class Files is used with the loop ForEachOrdered to read the same file line by line. In strategy 4, the readAllLines() function of the class Files is used to read all lines of the same file at once. Then, all the obtained lines are stored as items in the list f. Afterward, each line (item) is read from the list via for iterator loop. In strategy 5, the nextLine() function of the class Scanner is used inside a loop to read the same file line by line. The experimental results show that strategy 1 outperforms the other strategies in terms of execution time. The reason for this is that strategy 1 reads al the contents of the file and stores them in the main memory (RAM) and then it fetches each line directly from the memory instead of reading line by line from the hard disk as the other strategies do. Reading data from the memory is faster than reading the same data from the hard disk. It is worth mentioning that the file Test.txt was used with ten lines, each of ten bytes.

4.25 Array assignment

Two programming strategies have been used to assign a value to an array element as presented in Table 26. In strategy 1, a value is assigned to the array element a[0] via a temporary variable outside the loop. Thus, the assignment is performed only one time. In strategy 2, a value is assigned to the array element a[0] inside a loop where the assignment be repeated at each iteration. The experimental results show that strategy 1 outperforms strategy 2 in terms of execution time. The reason for this is that the assignment in strategy 1 is performed only one time whereas it is performed 100 times in strategy 2.

Table 25: Strategies to read a text file line by line

	Strategy Implementation	Execution Time (µs)
Strategy 1	BufferedReader f = new BufferedReader (new InputStreamReader(new FileInputStream ("E:/Test.txt"))); String str; while ((str = f.readLine()) != null) System.out.println(str);	0.4919
Strategy 2	Files.lines(Paths.get("E:/Test.txt")). forEachOrdered(System.out::println);	0.6672
Strategy 3	RandomAccessFile f = new RandomAccessFile ("E:/Test.txt" , "rw"); String str; while((str = f.readLine( )) != null) System.out.println(str);	0.7454
Strategy 4	List<String> f = Files.readAllLines (Paths.get ("E:/Test.txt")); for (String str : f) System.out.println(str);	0.8416
Strategy 5	Scanner f = new Scanner(new File ("E:/Test.txt")); String str; while(f.hasNext()) { str = f.nextLine(); System.out.println(str); }	1.5869

Table 26: Strategies to perform array assignment

	Strategy Implementation	Execution Time (µs)
Strategy 1	int a[]=new int[100]; int temp=0; for(int i=0;i<a.length;i++) temp=temp+5; a[0]=temp; System.out.println(a[0]);	0.0241
Strategy 2	int a[]=new int[100]; for(int i=0;i<a.length;i++) a[0]=a[0]+5; System.out.println(a[0]);	0.0349

5. CONCLUSION AND FUTURE WORK

In this experimental study, various programming strategies were employed to write efficient Java-based applications. The selected strategies aim to decrease the execution time of Java-based applications. This may also lead to higher energy efficiency especially when Java applications are created to run on devices with batteries or with limited processing capabilities. To fulfill the above aims, a number of experimental test scenarios were developed and executed to measure and analyze the performance impact of each selected strategy. The obtained results show that some programming strategies have a significant effect on performance efficiency and others have very limited impact. Besides, the results of this study show that Java programmers should be aware of the significant impact that even small and simple changes in coding can have on the performance of their applications. Some possible future works could be: (a) Applying and evaluating the same programming strategies used in this paper on other programming languages such as Python and C#. (b) Employing other data types and sizes which were not used in this paper and measure their impact on the overall performance of each programming strategy. (c) Measuring the performance impact of combining multiple programming strategies in the development of one application. (d) Including and examining other programming strategies which were not used in this paper.

6. REFERENCES

1. Jiang, G. and Zhao, C. (2010). Practice and exploration on bilingual teaching for Java Programming Language. *International Conference on Educational and Information Technology*. 465- 468.
2. Anupam, A. (2016). Tenets of Internet of Things (IoT) application and Java technology. *3rd International Conference on Recent Advances in Information Technology (RAIT)*. 697-699.
3. Li, H. (2011). RESTful Web service frameworks in Java. *IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*. 1-4.
4. Mohammed, T. Y. and Hamada, M. (2016). A cloud-based Java compiler for smart devices. *15th International Conference on Information Technology Based Higher Education and Training (ITHET)*. 1-6.
5. Liu, G. and Fan, G. (2010). Java Real-Time Software and Hardware Development Platform for Embedded Java. *3rd International Conference on Information Management, Innovation Management and Industrial Engineering*. 525-528.
6. Daud, S., Ahmad, R. B. and Murhty, N. S. (2008). The effects of compiler optimizations on embedded system power consumption. *International Conference on Electronic Design*. 1-6.
7. Foleiss J. H., Silva, A. F. D. and Ruiz, L. B. (2011). The Effect of Combining Compiler Optimizations on Code Size. *30th International Conference of the Chilean Computer Science Society*. 187-194.

8. Budimlic, Z. and Kennedy, K. (1997). *Optimizing Java: theory and practice.* Concurrency: Practice and Experience. 9(6). 445–463.
9. Myalapalli, V. K. and Geloth, S. (2015). Minimizing impact on JAVA virtual machine via JAVA code optimization. *International Conference on Energy Systems and Applications*. 19-24.
10. Tyystjärvi, J., Säntti, T. and Plosila, J. (2010). Efficient bytecode optimizations for a multicore Java co-processor system. *12th Biennial Baltic Electronics Conference*. 173-176.
11. Babic, D. and Rakamaric, Z. (2002). Bytecode optimization. *24th International Conference on Information Technology Interfaces*. 377-382.
12. Myalapalli, V. K. and Geloth, S. (2015). High performance JAVA programming. *International Conference on Pervasive Computing (ICPC)*. 1-6.
13. Lei, C. Z., Qiang, T. Z., Ming, W. L. and Liang, T. S. (2005). An effective instruction optimization method for embedded real-time Java processor. *International Conference on Parallel Processing Workshops (ICPPW'05)*. 225-231.
14. Corsaro, A. and Cytron, R. K. (2003). Implementing and optimizing real-time Java. *International Parallel and Distributed Processing Symposium*. 1-1.
15. Gorchakov, Y. A. and Kalganov, S. A. (2008). Programing and code optimization tips for AduC70xx series microcontrollers. *International Conference - Modern Technique and Technologies*, 93-96.