Performance Analysis of Java Object Serialization on Windows and Linux

Roslaili Kassim
Faculty of Information Technology and Quantitative Science,
University Technology MARA,
Shah Alam, Selangor, Malaysia
60-012-277 9559
roslaili@tmsk.uitm.edu.my

ABSTRACT

Object serialization is the process of saving an object onto a storage medium such as a file, database or to transmit it across a network connection link in binary form. This process of serializing an object is also called deflating or marshalling an object. The opposite operation, extracting a data structure from a series of bytes, is deserialization (which is also called inflating or unmarshalling). This paper describes the performance analysis comparison between Java arithmetical operations on Windows and Linux operating system. It also evaluates the running time performance over Java object serialization on both operating systems. In this experiment, four Java operations have been developed to represent different sizes of datasets and responses. The result has shown that Java arithmetical operations were faster on Linux than Windows. However, Java object serialization makes the running time of Java operations become faster on Windows. While on Linux, Java object serialization makes the running times of Java operations become slower. Monolithic kernel in Linux and hybrid kernel in Windows were the key factors that could influence this experiment. The running time performances of Java operations are faster on monolithic kernel than hybrid kernel.

Categories and Subject Descriptors

[Performance]

General Terms

Experimentation

Keywords

Object serialization, runtime comparison, monolithic kernel, hybrid kernel, performance analysis

1. INTRODUCTION

The running time performance of an application is one of the key factors that always being considered in the performance benchmark. Apart from reliability, interoperability, security and others, the fast response time was one of the key factors in good applications. Object serialization is one of the mechanisms to improved runtime performance. It is a process of saving an object onto a storage medium such as a file, database or to transmit it across a network connection link in binary form [1]. This process of serializing an object is also called deflating or marshalling an object. The opposite operation, extracting a data structure from a series of bytes, is deserialization (which is also called inflating or unmarshalling) [2]. As discussed in [3], the goals for serializing Java objects are to:

- Have a simple yet extensible mechanism.
- Maintain the Java object type and safety properties in the serialized form.
- Be extensible to support marshaling and unmarshaling as needed for remote objects.
- Be extensible to support simple persistence of Java objects.
- Require per class implementation only for customization.
- Allow the object to define its external format.

Object serialization enable data to be stored or transferred in binary form [4]. Thus, it is expected that object serialization mechanism could effectively improve the runtime as it does need to do less transformation than data that have to be transformed into human readable character. ObjectOutputStream is the primary output stream class that implements the ObjectOutput interface for serializing objects. ObjectInputStream is the primary input stream class that implements the ObjectInput interface for deserializing objects. These high-level streams are each chained to a low-level stream, such as FileInputStream or FileOutputStream. The low-level streams handle the bytes of data. The writeObject method saves the state of the class by writing the individual fields to the ObjectOutputStream. The

readObject method is used to deserialize the object from the object input stream [1].

Runtime performance on Windows and Linux might be influenced by the computer kernel. The computer kernel is the central component of most computer operating systems. Its responsibilities include managing the system's resources i.e. the communication between hardware and software components. Most operating systems rely on the kernel concept [6]. The existence of a kernel is a natural consequence of designing a computer system as a series of abstraction layers, each relying on the functions of layers beneath itself. The kernel, from this viewpoint, is simply the name given to the lowest level of abstraction that is implemented in software. A kernel will usually provide features for low-level scheduling of processes (dispatching), inter-process communication, process synchronization, context switching, manipulation of process control blocks, interrupt handling, process creation and destruction, and process suspension and resumption [7].

Linux kernel was based on monolithic kernel while Windows kernel is based on hybrid kernel [8]. Hybrid kernels are essentially a compromise between the monolithic kernel approach and the microkernel system. This implies running some services (such as the network stack or the filesystem) in kernel space to reduce the performance overhead of a traditional microkernel, but still running kernel code (such as device drivers) as servers in user space. In a monolithic kernel, all OS services run along with the main kernel thread, thus also residing in the same memory area. This approach provides rich and powerful hardware access [10]. The microkernel approach consists of defining a simple abstraction over the hardware, with a set of primitives or system calls to implement minimal OS services such as memory management, multitasking, and inter-process communication [11]. Other services, including those normally provided by the kernel such as networking, are implemented in user-space programs referred to as *servers*. Microkernel are easier to maintain than monolithic kernels, but the large number of system calls and context switches might slow down the system because they typically generate more overhead than plain function calls [8]. A microkernel allows the implementation of the remaining part of the operating system as a normal application program written in a high-level language, and the use of different operating systems on top of the same unchanged kernel. It is also possible to dynamically switch among operating systems and to have more than one active simultaneously [8].

2. RELATED WORKS

The analysis of object serialization has been described in [2] and its comparisons on Java and .Net platform have been discussed. It is based on benchmark technique by using profiler and the benchmarking was done on binary and XML serialization. Another benchmark was done in [15] but its evaluation was based on Java.net package and XML-RPC application. Several suggestions to optimize the performance include to improve the reflection in object serialization, to create new objects and improvement in data representation and marshaling. Besides, the compression technique was also being used in order to reduce the size of client machine's request and server machine's response.

More solutions on optimizing object serialization were described in [16]. The running time can be decreased by generating serialization code for each class that can be serialized instead of using default Java serialization mechanism which using object reflection to convert object to byte to be sent over the wire. In addition, objects should be created without calling user defined properties when building object graph from the stream to avoid side effects.

In [17], the source code of default Java serialization mechanism has been modified to enable reusing the existing functionality from its subclasses. Finally, the call graph application has been used in serialization/marshaling which represent the example of object traversal [18]. A serializer will transform partial graphs of objects into a stream of bytes. It is suggests that the serialization code that will be generated should be using minimal code or alternatively the dynamic traversal should be arranged so that it can be executed with minimal running time impact.

The review of related works has shown that many authors have discussed about the object serialization concept. However, there is no standardized method for the object serialization assessment where most of it has been done on distributed environments. The authors of [2] and [15] have analyzed the execution performances and compare within different types of software components. The authors [2] and [15] have discussed the possible solutions for object serialization problems and authors of [16] – [18] have invented new object serialization mechanisms to improve the execution performance of the Standard Java Object Serialization. This work focused on performance analysis of serialization mechanism built into .NET platform and Java platform. This paper is organized as follows. First, the object serialization evaluation method has been described. Then the results will be presented and the comparison of execution performance on different applications will be analyzed. Finally, the differences in term of runtime performance will be identified for both operating system and the conclusions for possible improvements will be suggested.

3. SERIALIZATION EVALUATION

3.1 Method

The goal of this paper is to evaluate the object serialization performance from the time perspective that is how fast the serialization process will be performed on different operating system. However, because of it involve the sending and receiving stream from and to the memory buffer, the size of memory will have effects on the execution performance. Thus in order to resolve this issue, each operations running time have been captured by using System.nanoTime() method. It could determine the execution performance as well as the effect of memory buffer for these operations. Besides, it also could provide reliable results than profiling tool which usually involves overhead latency.

For the purpose of this experiment, four arithmetical operations have been developed and each of the operations will be discussed further below.

i. IntPrime
ii. IntRandom
iii. DoubleAverage
iv. DoubleLogarithm

IntPrime was developed to represents small dataset with small response. This operation determines whether the given integer is prime or otherwise. It will read the input stream from the external file, processes it and then writes the output to the output stream. The computation will be executed to determine whether the input number is a prime number or not.

IntRandom was developed to represents small dataset with big response. It reads input, n and return n random numbers to the output stream.

DoubleAverage is to represent big dataset with small response. It reads a list of numbers from input file, calculates the average and send the average to the output stream.

Big dataset with big response was represented by DoubleLog. It reads inputs from the input file and produces the output to the output stream. This operation transforms each value in the input to logarithmic value and writes all of the transformed values to the output stream.

The objective of this experiment is to determine the differences in object serialization performance for different datasets and response. In Addition, the experiment also would like to benchmark the running time performance of Java arithmetical operations on windows and Linux.

To measure the time required for object serialization, each test have been repeated for 50 times and its average value have been calculated, which is then reported in this paper.

3.2 Software and Hardware Equipment.

The experiment was carried out on Intel Core Duo machine with 1.66 GHz processor and 1GB RAM. Microsoft Windows XP Professional with Service Pack 2 has been used for Windows evaluation and Linux Ubuntu has been used for Linux evaluation. Java operations were developed using JCreator LE 4.00 and were compiled using Java Development Kit (JDK) 1.6.0_02.

4. OBJECT SERIALIZATION ANALYSIS.

The runtime results collected in this experiment have been compiled in the tables below. All of these tables contain runtime results for Java application on Linux and Windows. Each arithmetical operation has been tested on six input data and 50 iterative executions have been conducted for each of the input data. Then, the mean of those 50 runtimes has been calculated and presented in the table of results.

4.1 Small Dataset/Small Response Analysis

IntPrime represents small request and small response. The data values have been generated randomly and the output have been based on each of these data values. Table 1 shows that the runtime that is needed by Java application execution to write the output to the console on Linux were faster than that on Windows.

	Java Runtime on Linux		Java Runtime on Windows	
Dataset Value	Output to Console	Output to File	Output to Console	Output to File
41	0.00044	0.00036	0.00068	0.00048
6334	0.00058	0.00045	0.00092	0.00047
11478	0.00028	0.00031	0.00090	0.00046
15724	0.00064	0.00032	0.00087	0.00048
18467	0.00029	0.00036	0.00087	0.00043
19169	0.00032	0.00069	0.00087	0.00043

Table 1: Object Serialization Performance (sec) for IntPrime

4.2 Small Dataset/Big Response Analysis

The operation for this execution has been represented by IntRandom. Based on this experiment, the Java runtime execution needed less time on Linux compared to Windows for output written to the console. However, output written to file takes longer runtime on Linux compared to Windows.

Dataset Value	Java Runtime on Linux: Output to Console	Java Runtime on Linux: Output to File	Java Runtime on Windows: Output to Console	Java Runtime on Windows: Output to File
1	0.00063	0.00023	0.00166	0.00035
20000	0.33181	0.66109	2.88009	0.51913
40000	0.63943	1.34012	5.55438	1.02406
60000	0.95494	2.00427	8.32729	1.55400
80000	1.27567	2.66835	11.11038	2.05999
100000	1.59147	3.29027	13.90043	2.59699

Table 2: Object Serialization Performance (s) for IntRandom

4.3 Big Dataset/Small Response Analysis

DoubleAverage read a list of double numbers and return its average. Based on this experiment, the Java runtime execution needed less time on Windows compared to Linux.

Dataset Size	Java Runtime on Linux: Output to Console	Java Runtime on Linux: Output to File	Java Runtime on Windows: Output to Console	Java Runtime on Windows: Output to File
1	0.00063	0.00023	0.00148	0.00049
20000	0.33181	0.66109	0.00963	0.00741
40000	0.63943	1.34012	0.01764	0.01476
60000	0.95494	2.00427	0.02528	0.02207
80000	1.27567	2.66835	0.03360	0.02952
100000	1.59147	3.29027	0.03863	0.03790

Table 3: Object Serialization Performance (s) for DoubleAverage

4.4 Big Dataset/Big Response Analysis

This operation reads list of double numbers before transform it into a list of log numbers and write it back to the output console or to the output file. Based on this experiment, the Java runtime execution needed less time on Linux compared to Windows for output written to the console. However, output written to file takes longer runtime on Linux compared to Windows.

Dataset Size	Java Runtime on Linux: Output to Console	Java Runtime on Linux: Output to File	Java Runtime on Windows: Output to Console	Java Runtime on Windows: Output to File
1	0.00550	0.00040	0.00092	0.00048
20000	0.55359	0.71453	4.57041	0.60476
40000	1.10458	1.46031	9.06196	1.22794
60000	1.66074	2.20536	13.65480	1.82977
80000	2.22988	2.84291	18.23333	2.44221
100000	2.80685	4.09612	22.85148	3.08079

Table 4: Object Serialization Performance (s) for DoubleLog

5. RUNTIME PERFORMANCE COMPARISONS

In this section, the comparison of the running time between Java runtime execution on Linux and Windows have been performed. Figure 1, 2, 3 and 4 represent four arithmetical operations that have been discussed earlier in this paper. It is found that on each of the operation, object serialization to the file could improve most of the running times on Windows than Linux. However the Java runtime executions on Linux show that object serialization make the runtime become longer. The comparison will be explained further below.

5.1 Small Request/Small Response

For this operation, it has been showed in the graph that object serialization on Windows better than object serialization on Linux. The improvement in runtime execution of Java on Windows can be seen obviously while the runtime execution for Java on Linux shows differently.

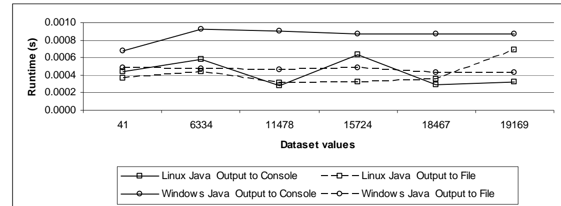

Figure 1: IntPrime Operation Performance for Java, C# and C++.

5.2 Small Request/Big Response

In IntRandom operation, the result of its running times was different than the previous operation. Object serialization on Windows shows that it can improve runtime but object serialization on Linux makes the runtime become longer.

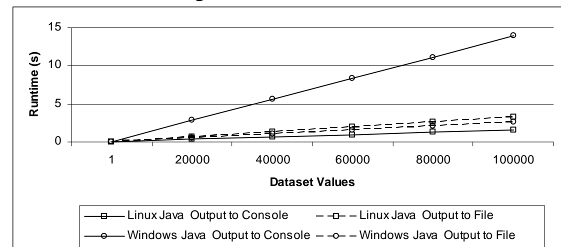

Figure 2: IntRandom Operation Performance for Java, C# and C++.

5.3 Big Request/Small Response

In this experiment, it shows that object serialization has successfully improve the Java execution runtime although the improvement were very small compared to the other operations.

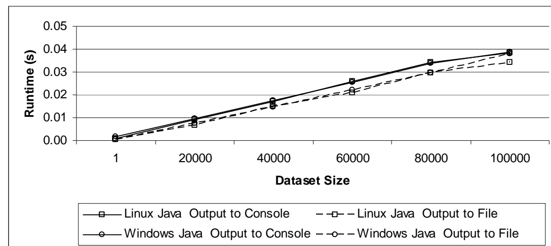

Figure 3: DoubleAverage Operation Performance for Java, C# and C++.

5.4 Big Request/Big Response

The result of the DoubleLog operation has shown that object serialization on Windows could improve runtime but object serialization on Linux makes no difference in runtime.

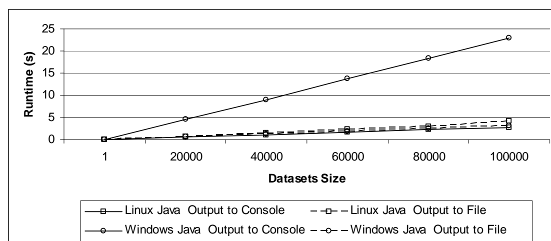

Figure 4: DoubleLog Operation Performance for Java, C# and C++.

6. DISCUSSION

In this experiment, four types of arithmetical operations such as operations to determine prime number, generate random number, count the average and transform the number to its logarithm have been evaluated. Each operation has been written in Java using Java Development Kit (JDK). Each operation has been written into two versions; the operation with its output written to the output console and the operation with its output written to the file. Object Serialization Analysis in Section 4 and 5 has shown that almost all operations running on Windows with its output written to the file are better in terms of its runtime. However, object serializations on Linux could not improve the Java execution runtime. These results suggest that:

i. Object serialization on Windows may improve the running time performance compared to object serialization on Linux
ii. Without serialization, Java operation runtime improved tremendously if it is running on Linux as compared to Java runtime on Windows.

The process of serialization consists of several important steps. In serialization, the compiler will access the class information and gather the state of the attributes and relations before marshal the state information to a stream, where the in memory representation has to be changed to the representation, used for the serialized stream. Then the stream will be written to the memory file, database or other location.

Based on the results that have been collected in the experiment, the main finding was object serialization to file storage is faster than object serialization to the output console in most of the arithmetical operations. The explanation over this analysis was much related to Input/Output stream mechanism in Java. Output stream being displayed on the output console offer only temporary storage of data and the data is lost when the program terminates. Files and database are used for long time retention of large amounts of data. However, this is true for only Java object serialization on Windows. Experiment on Linux has produced different result.

Data [15] maintained in files often is called persistent data. Computers store files on secondary storage devices such as magnetic disks, optical disks and magnetic tapes. All data items that are being processed by computers are reduced to combinations of 0s and 1s. The smallest data item that computers support is called a bit (binary digit). Every character in a computers character set such as decimal digits, letters or special symbols is represented as patterns of bits. In Java, each file is views as a sequential stream of bytes.

In this experiment, when an input file is opened, an object was created and a stream will be associated with the object. In most of the compilers, when each of the arithmetical operations executes, the runtime environment creates three stream objects; Standard Input Stream Object, Standard Output Stream Object and Standard Error Stream Object. Standard Input Stream Object enables a program to input data from the input file, Standard Output Stream enables a program to store the output data to the output file and Standard Error Stream Object enables a program to displays error message to the screen in case of the input file is not found. In this experiment, object serialization is performed with byte-based streams, so the input and output file will be binary files. Binary files are not human readable as compared to response to output console which is being displayed as the characters. The process of converting binary output into characters output will consumes time. Therefore, object serialization to the file is faster than object serialization to the output console because the time taken by the computer for translating bit into characters has been eliminated.

Intermediate representation may be output by programming language implementations to reduce hardware and operating system dependencies by allowing the same code to run on different platforms. Intermediate code may be either directly executed on a virtual machine, or it may be further compiled into machine code for better performance.

The runtime execution of Java operations on Linux outperformed the runtime execution of Java operations

on Windows. This possibly cause by preemption. Preemption in computing is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such a change is known as a context switch. It is normally carried out by a privileged task or part of the system known as a preemptive scheduler, which has the power to pre-empt, or interrupt, and later resume, other tasks in the system.

In any given system design, some operations performed by the system may not be preemptible. This usually applies to Kernel functions and service interrupts which, if not permitted to run to completion, would tend to produce race conditions resulting in deadlock. Barring the scheduler from preempting tasks while they are processing kernel functions simplifies the kernel design at the expense of system responsiveness. The distinction between user mode and kernel mode, which determines privilege level within the system, may also be used to distinguish whether a task is currently preemptible.

Some modern systems have preemptive kernels, designed to permit tasks to be preempted even when in kernel mode. Examples of such systems are the Linux kernel 2.6 and some BSD systems. On the other hands, preemptive multitasking is used to distinguish a multitasking operating system, which permits preemption of tasks, from a cooperative multitasking system wherein processes or tasks must be programmed to yield when they do not need system resources.

In simple terms: Pre-emptive multitasking involves the use of an interrupt mechanism which suspends the currently executing process and invokes a scheduler to determine which process should execute next. Therefore all processes will get some amount of CPU time at any given time.

At any specific time, processes can be grouped into two categories: those that are waiting for input or output (called "I/O bound"), and those that are fully utilizing the CPU ("CPU bound"). In early systems, processes would often "poll", or "busy wait" while waiting for requested input (such as disk, keyboard or network input). During this time, the process was not performing useful work, but still maintained complete control of the CPU. With the advent of interrupts and preemptive multitasking, these I/O bound processes could be "blocked", or put on hold, pending the arrival of the necessary data, allowing other processes to utilize the CPU. As the arrival of the requested data would generate an interrupt, blocked processes could be guaranteed a timely return to execution.

Other factor that might influenced the Java operations performance was the type of the operating system kernel. Windows is based on hybrid kernel of microkernel and monolithic kernel while Linux is based on monolithic kernel. Both kernels have the advantages and disadvantages e.g. Linux monolithic kernel need less time to perform Java operations than Windows hybrid kernel but Linux monolithic kernel doesn’t support Java Object serialization as in Windows hybrid kernel.

7. CONCLUSION

In this paper, it is shown that the performance of object serialization could be increased if the output of the application is being written to the file for certain operations. However this is true for object serialization on Windows. Results on Linux have shown that Java object serializations to the file take longer time than output being sent to the console. Overall, Java arithmetical operations are faster running on Linux than Windows. This experiment tested the arithmetical operations which focus on big dataset/big response, big dataset/small response, small dataset/big response and small dataset/small response application.

8. REFERENCES:

[1] Java Serialization http://www.javabeginner.com/object-serialization.htm

[2] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, A. Zivkovic, August 2003, Object serialization analysis and comparison in Java and .NET, ACM SIGPLAN Notices, Volume 38 Issue 8

[3] System Architecture http://java.sun.com/javase/6/docs/platform/serialization/spec/serial-arch.html

[4] Object Serialization http://java.sun.com/javase/6/docs/technotes/guides/serialization/

[5] Michael Jang, 2004, Mastering Red Hat Enterprise Linux 3, Sybex Corporation

[6] M. Tim Jones, 2007, Anatomy of the Linux kernel: History and architectural decomposition http://www.ibm.com/developerworks/linux/library/l-linux-kernel/

[7] Daniel P. Bovet, Marco Cesati, December 2002, Understanding the Linux Kernel, 2nd Edition

[8] MS Windows NT Kernel-mode User and GDI White Paper http://www.microsoft.com/technet/archive/ntwrkstn/evaluate/featfunc/kernelwp.mspx?mfr=true

[9] Dylan Griffiths, Dwight Makaroff, 2006, Hybrid vs. Monolithic OS Kernels: A Benchmark Comparison, IBM Centre for Advanced Studies Conference, Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, ACM New York, NY, USA

[10] Volkmar Uhlig, 2007, The Mechanics of In-Kernel Synchronization for a Scalable Microkernel, ACM SIGOPS Operating Systems Review, Volume 41, Issue 4 (July 2007), Pages: 49 – 58, ACM New York, NY, USA

[11] M. Allman, March 2003, An evaluation of XML-RPC, ACM SIGMETRICS Performance Evaluation Review, Volume 30 Issue 4

[12] F. Huet, D. Caromel, H. E. Bal, Nov. 2004, A High Performance Java Middleware with a Real Application, Proceedings of the ACM/IEEE SC2004 Conference

[13] B. Haumacher, M. Philippsen, 1999, More Efficient Object Serialization, Lecture Notes In Computer Science; Vol. 1586, Springer-Verlag

[14] K. Lieberherr, B. Patt-Shamir, D. Orleans, 2004 , Traversals of object structures: Specification and Efficient Implementation, ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 26 , Issue 2 (March 2004) Pages: 370 – 412

[15] J. Meyer, T. Downing, 1997, Java Virtual Machine, O'Reilly