Ruby under Scanner: Comparison with Java

Sudipto Das, Vinod Kone
sudipto@cs, vinod@cs
University of California, Santa Barbara

ABSTRACT

In this paper we aim to compare the Ruby programming language with one the most popular Object Oriented Programming languages Java. For this study, we concentrate on three main aspects for comparing the performance of Ruby with Java,

- *Features that the two languages provide,*
- *Comparison from the programmer's perspective, and*
- *Performance in terms of time.*

From the experiments we performed, we came to the conclusion that Ruby is easier to program and is much more compact language when compared to Java. Ruby increases programmer productivity. But from the performance perspective, Java is the preferred language for large complex computational tasks.

1. INTRODUCTION

The last decade has seen the development of various modern programming languages, namely Java, C#, Smalltalk, Eiffel, Perl, Python, Rexx, Ruby. With the advent of these programming languages, various other paradigms such as Static and Dynamic typing, pure Object Orientation etc have also developed. There has been considerable research in comparing various languages as in [6–8, 16]. In this paper, our goal is to compare Ruby with Java, two very modern programming languages that have gained widespread popularity. Java [13] is a very elegant programming language that has developed a lot from the days it was first released and is an attempt to increase the programmer productivity by relieving the programmer from various tasks such as *pointer manipulation, memory management* etc. and provides a vast library that makes programming complex tasks easier and also retaining the performance benefits. Ruby [4, 10] goes one step further by making the syntax much more compact and simpler and also provides various built-in features that make the common tasks of text processing, creating and processing threads, web programming etc. much easier. Ruby along with the Rails framework [11] has gained widespread popularity in the last couple of years for web based programming.

In this work we concentrate on evaluating the Ruby language, without the Rails framework, and compare it with Java. Our comparison is based on the various aspects of the programming language such as the features of the two languages, ease of development and performance in terms of time and in some cases, the memory requirement. We compare in terms of memory only in selective cases where programs are memory intensive and have high memory requirements. We leave out this aspect in comparisons where memory is not a constraint. For the purpose of this comparison we choose simple programs that are implemented in both Java and Ruby using almost similar approaches. In each program we concentrate on evaluating one or two aspects of the two languages. For the sake of consistency, the same algorithm is used in both the languages, and the same programmer implements the programs in both the languages.

The remainder of this paper is organized as follows. Section 2 compares the work done till now in comparing Ruby with other common languages like Java, Section 3 compares the two languages Ruby and Java based on the features of the two languages as well as analyzes the two languages from the programmer's perspective, Section 4 provides experimental evaluation comparing the two languages in terms of time and memory and finally Section 5 concludes the paper.

2. RELATED WORK

Ruby has gained huge popularity over the last few years because of the flexibility and ease of programming that Ruby lends to the programmers. But very little research has gone into comparing Ruby with other standard programming languages, such as Java or C++, in terms of performance. Most of the work we could find in general compares the features of the languages [1, 3]. There has also been some work in comparing Ruby on Rails [2, 9, 14], concentrating mostly on certain specialized applications. But none of the works provide a really comprehensive study. As we can see [3] provides a very high level overview in comparing the two languages.

In this work, we concentrate on providing a thorough comparison of Ruby and Java. To make this analysis complete, we compare the languages based on

- High-level language features,
- From a programmer's perspective, and
- Performance wise

The following sections delve deeper into providing a comprehensive comparison of the two languages.

3. COMPARISON OF LANGUAGE FEATURES

In this section, we give a comparative study of the two languages Ruby and Java in terms of the various language features as well as from a programmer's perspective. We aim to provide a high level information about the two languages under consideration.

3.1 Features of the Language

In this sub-section, we compare the various features of the two languages. A few features have been listed in Table 1 [15]. The features are explained in greater detail as under

Object Orientation : As [15] defines, Object Orientation of a language is determined based on the following qualities

1. Encapsulation/Information Hiding
2. Inheritance
3. Polymorphism/Dynamic Binding
4. All pre-defined types are Objects
5. All operations performed by sending messages to Objects
6. All user-defined types are Objects

All the above properties are satisfied by Ruby. But Java has eight "basic" types that are not objects. On the other hand, everything in Ruby is an Object Java also fails to meet quality 5 by implementing basic arithmetic as built-in operators, rather than messages to objects. So Java cannot be classified as a Pure Object Oriented Language. But except the way eight basic types are handled, everything else in Java in an object. The designers of Java advocate that the basic types are kept in Java to enhance performance. The basic types in Java have corresponding "Wrapper" classes but using these classes increases the overhead of the system as compared to basic types. They are used mainly with the Collections framework that can only deal with object references.

Static vs. Dynamic Typing : Statically-typed languages like C, C++, Java mandate that all variables are declared with a specific type before they can be used for the first time. The compiler will then ensure that at any given time the variable contains only an object compatible with that type. This type information is even more important in Java which guarantees type-safety, i.e. the variables are used only in the way that its type specification permits. A dynamically typed system such as Ruby, Perl doesn't require variables to be declared as a specific type. Any variable can contain any value or object. Since the type declarations are not required, the variable can be used directly when required. The runtime system determines the type of the variable depending on the value they hold at the time of execution.

The debate over typing of systems (whether static or dynamic) has raged in Object-Oriented circles for many years with no clear conclusion. The advocates of dynamic typing argue that it increases productivity and lends greater flexibility to the programmer. On the other hand, proponents of static typing argue that it increases efficiency of the resulting product and also result in safer and more reliable systems.

Inheritance : Inheritance is the ability for a class or object to be defined as an extension or specialization of another class or object. This is an important feature of Object Orientated Programming (OOP) that lends it immense power. This increases the programmer productivity as it allows code reuse. Although inheritance is a powerful feature in OOP, multiple inheritance creates various sorts of problems as can be seen in C++. Inheriting multiple implementations of the same method may lead to confusion as the two implementations might have very different behavior. In order to avoid confusion, some form of Dis-disambiguation is required. But having multiple behaviors in the same object is a desirable quality. To deal with this, both Java and Ruby take a similar approach. In both the languages, a class can inherit from one and only one class, but can have multiple behaviors by implementing *interface* in Java or adding *mixins* in Ruby.

Method Overloading : Method Overloading refers to having methods with the same name but taking a different set of arguments [15]. It is a form of implementing Parametric Polymorphism in Java and is also a very important feature of OOP. Ruby does not support method overloading per se. Ruby takes a somewhat different approach. In method overloading, either the number of arguments can be different or the type of the arguments can be different. As the variable in Ruby are not typed, so the arguments cannot differ in type. A Ruby method can by default take different number of arguments by having default arguments in the methods.

Operator Overloading : Operator overloading is the ability for a programmer to define an operator (such as +, or *) for user-defined types [15]. Java does not support operator overloading while Ruby does. Operator Overloading is more of a syntactical ornamentation rather than being an essential feature. When abused, this can lead to cryptic and obfuscated code, as it is upto the programmers to decide on how the operators will behave on the user-defined types. So, for example, one may define the + operator to perform multiplication of two numerical objects.

Garbage Collection : Garbage collection is a technique by which the language implementations can automatically obtain the memory allocated to objects that can no longer be referenced by the user code (Garbage). This relieves the programmer from the responsibility of memory management and also helps build a memory safe system where the system

Feature	Ruby	Java
Object Orientation	Pure	Hybrid
Static / Dynamic Typing	Dynamic	Static
Inheritance	Single Class, multiple mixins	Single Class, multiple interfaces
Method Overloading	No	Yes
Operator Overloading	Yes	No
Garbage Collection	Mark and Sweep	Mark and Sweep, Generational Incremental, Mark and Compact
Reflection	Yes	Yes
Access control	public, protected, private	public, protected, package, private
Multithreading	Yes	Yes
Regular Expressions	Built-in	Libraries
Built-in Security	Yes	Yes

Table 1: Comparison of the Features of Java and Ruby

performs all memory allocation and deallocation. The alternative is for the programmer to explicitly free any memory that is no longer needed. Most modern programming languages including Java and Ruby are Garbage collected. Ruby only uses the Mark and Sweep Garbage Collector. The Java Virtual Machine has various types of Garbage collectors as listed in Table 1.

Reflection : Reflection lends a program the ability to obtain information about an object at run-time [15]. This includes the ability to determine the type of the object, its inheritance structure, and the methods it contains, including the number and types of parameters and return types. Ruby has a very powerful and flexible reflection mechanisms. Java also supports reflection, but not in as flexible and dynamic fashion as in Ruby.

Access Control : As [15] defines, Access control refers to the ability for a modules implementation to remain hidden behind its public interface. It is closely related to the encapsulation/information hiding principle of object-oriented languages. Both Ruby and Java implement access control through reserved keywords. Members within a class can be classified as $private$ (visible only with the class), $protected$ (visible within the class and its sub classes) and $public$ (visible to everybody). In addition, Java brings a concept of $packages$ and an associated access specifier which is also known as the $default$ access specifier (visible withing the class and all classes within the package).

Multithreading : Multithreading is the ability for a program to perform multiple tasks concurrently. Most modern programming languages provide an easy interface to implement threads. Java too provides a Thread library. Ruby is somewhat unique in that its threading capabilities are built in to the interpreter itself, rather than wrappers around the operating system threading operations. This has both advantages as well as disadvantages. The advantage is that we can have Ruby threads even in Operating Systems like MS-DOS, that have no support for multi-tasking A disadvantage is that an OS call will block the entire interpreter [15].

Regular Expression : As defined in [15] Regular expressions are pattern matching constructs capable of recognizing the class of languages known as regular languages. They are used very often for text processing systems as well as for various general applications using pattern recognition. Java provides support for regular expressions by providing class libraries that help processing regular expressions. Ruby, like Perl, has regular expressions built into it. This is very evident is the huge difference in the text processing programs, namely Anagrams and Parser as seen in Table 2. These programs are explained in greater detail in Section 4.

Built-in Security : Whenever third party code is executed on a system, one has to sure that it comes from a "trusted" source. The runtime environments of most modern languages provide features that is used to determine whether the code to be executed can be executed without causing security hazards. With the advent of the Java applet paradigm, untrusted code is often downloaded into the local computers and executed. So, the runtime environments of the languages should be equipped with the ability to securely execute this code so that it does not jeopardize the system. Both Java and Ruby have security features that are part of the the language specification.

3.2 From a Programmer's Perspective

In this section, we compare how similar and different the two languages Java and Ruby are, from a Programmer's perspective. Ruby is a pure object oriented language whose syntax is greatly influenced by that of Perl. It is dynamic typed language and has a very compact syntax doing away with semicolons ending statements, braces wherever they are not necessary. The compactness of the coding style is evident in Table 2 (Similar indentation and comments have been used in programs of both the languages, files have been compressed using gzip in order to remove inadvertent white spaces as much as possible). Even though the gains in terms of code size

Program	Java	Ruby
Anagram	1028	423
Client/Server	1402	547
Factorial	268	204
Factorial (Multiple iterations)	365	242
Recursive Fibonacci	235	208
Insertion Sort	306	254
Iterative Function Calls	203	194
Library Sort	686	463
Linear Search	789	612
Multiply Matrix	601	511
Matrix Multiply (Multiple Threads)	810	558
Merge Sort	721	644
Parser	960	385
Polynomial Function	295	177
Polynomial Function (Using Method Calls)	335	230
Quick Sort	943	837

Table 2: Comparison of the Size of Code in Bytes for the same program in Java and Ruby

would not seem to be much, but we could better judge the gain if we look at it as percentage gain. As is evident from Table 2, the gain ranges from 25% to 200% in the programs considered. We can see that complex tasks such as finding Anagrams (which involves a complex set of tasks as described in Section 4) or parsing a text file require much less effort in Ruby from the programmer's perspective. Everything in Ruby is an object and this lends uniformity. Most of the built-in objects have overloaded implementation of the common operators, which gives great flexibility and ease in manipulating with the objects. When dealing with numbers, Ruby automatically determines objects of which class should be used to appropriately store the numbers. For example, if we have a small number, Ruby represents it using the $Fixnum$ class, but if the number is greater than a particular value, it is represented by the $Bignum$, and this transition is done automatic and is transparent from the programmer's view point. Ruby has regular expressions built into the language which eases the programmer's life when it comes to writing program for text processing. Ruby provides, as a part of the language specifications, Arrays and Hashtables that re-size as the elements are added or deleted from it. One of the most powerful feature of Ruby is *Code Blocks*. These are chunks of code that can be associated with method invocations, almost as if they are parameters. They are much simpler than the corresponding construct in Java i.e. Inner Classes. They can be used to implement Callbacks and to pass around chunks of code, or implement iterators. They lend great flexibility to the Ruby language. Another feature in Ruby is that it automatically creates the accessor methods of the fields of the classes as and when required (by the use of $attr_reader$). The way in which variables are named has different implications in Ruby. These may be listed as under:

- Local variables, method parameters and method names should all start with a lower case letter or with an underscore
- Global variables should be prefixed by a $
- Instance variables should start with a @
- Class variable (equivalent to static variables in Java) start with @@
- Class names, module names, and constants should start with an uppercase letter

In addition to the above features in Ruby, included to make the programmer's task easier, Ruby and Java also share various common features as well. These are listed as under:

- The control structures are very similar in the two languages (in addition to those in Java, Ruby provides some additional constructs)
- Java classes have constructors that perform the task of initializing the objects, similarly, classes in Ruby have the $initialize$ method
- The $toString()$ method in Java can be used as a uniform means of having a String representation of an object, the equivalent in Ruby is to_s
- Java implements multiple inheritance by using the concept of $interfaces$, the Ruby equivalent is $mixins$
- Ruby has $delayed\ binding$ like Java
- Like Java, Ruby also has a class named Object as the common ancestor for all classes. The Object class in Ruby has 35 methods while that in Java has 11 methods

In conclusion we can view Ruby as a language that has been designed to increase the programmer productivity.

4. EXPERIMENTAL EVALUATION

This section explains the various experiments performed to compare the performance of Ruby with Java. The experiments were performed on a Linux machine running Fedora Core 6 (kernel version 2.6.18-1.2869.fc6). The Java version used for executing the programs is $Java^{TM}$ *SE Runtime Environment (build 1.6.0-b105) Java* $HotSpot^{TM}$ *Server VM (build 1.6.0-b105, mixed mode)* [13]. and the version of Ruby used is *ruby 1.8.4 (2005-12-24) [i386-linux]*. All time measurements are done using the $time$ command in Linux(which is an internal command in $bash$), which gives as its output, the execution time of the program as $Real$ time (the total time from the start of execution to the end of execution), $User$ time and $System$ time. We take the $User$ times to plot our graphs, neglecting the real time because of the multitasking nature of the system; the real time might include the time taken by other processes as well.

Figure 1: Factorial of a number calculated Recursively

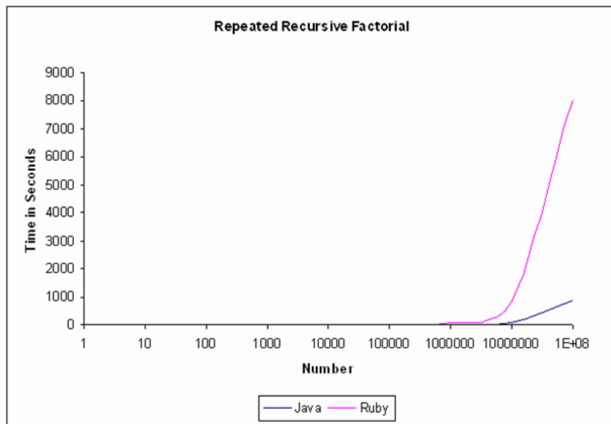

Figure 2: Factorial of 25 calculated Recursively repeated many times

4.1 Function Calls and Recursion

In these set of experiments, we compare how efficiently calls to functions/methods, and recursion is implemented in the two languages. The time reported in these graphs is the user time as obtained as output of the time command. The programs in Java and Ruby are identical from the algorithmic point of view. The analysis is as under:

4.1.1 Program that calculates the factorial of a number recursively:

In this experiment, we recursively calculate the factorial of a number. This program uses tail recursion and the number of function calls increases linearly with the number whose factorial we have to calculate. In Fig 1, the curve represented by Java is the time obtained when the *Java Virtual Machine* (JVM) was invoked without any of the client or server option set. On the other hand, curve represented by Java Server is the one obtained by invocation of JVM with the -server flag set while Java Client is that with the -client flag set. From the first view of the graph, one might conclude that Ruby is far more efficient compared to Java when it comes to implementing this recursive function. But one thing that

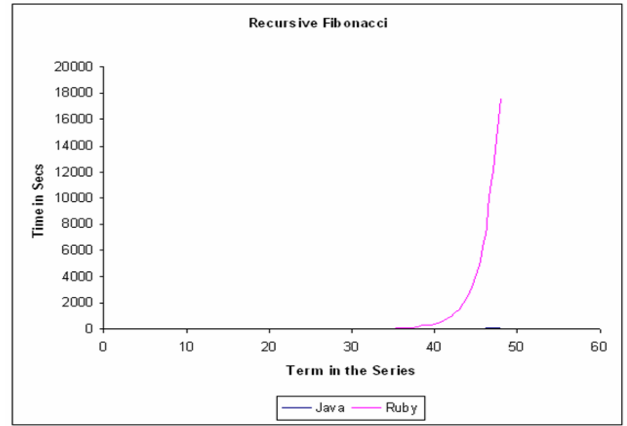

Figure 3: Calculating the nth term of Fibonacci Series using Recursion

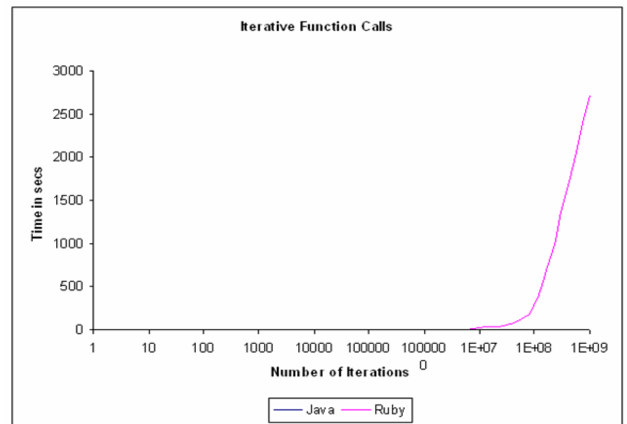

Figure 4: A function with empty body called repeatedly in a loop

should be considered is that there is no order of magnitude difference between Java or Ruby, it only seems that there seems to some kind of offset that is skewing the results. An explanation of this could be the initial overhead associated with a Java program (it is loaded into the JVM followed by bytecode verification, *Just In-Time* (JIT) compilation and other processing within the JVM). Evaluation of this aspect is the motivation for the experiments in the next sub-section.

4.1.2 Program that repeatedly calls the recursive factorial in a loop:

In this sub-section, we experiment to see whether the results of the previous section were somewhat skewed by the initial loading overhead in Java. To test this, instead of invoking the JVM repeatedly, we do the same task repeatedly in the same invocation of the JVM or the Ruby interpreter. In this experiment, we recursively calculate the factorial of 25. But the function is called repeatedly in a loop and we vary the number of iterations of this loop. As is evident in Fig 2, as the number of function calls increases exponentially, the time taken by Ruby also rises exponentially whereas the skew because of the

initial overhead in loading the Java programs vanishes. Hence we can assert with some confidence that when the job to be done is less time consuming, the initial overhead with loading Java leads to the reduced system performance in terms of time. In such a case, Ruby performs better as the Ruby Interpreter does not have this overhead at load time. But once the Java code has been loaded in the VM and the JIT compiler has compiled the code, things get much faster.

4.1.3 *Program to calculate the nth term in the Fibonacci Series:*

In this sub-section, we are trying to reassert the claim in the previous section. It is well known that in the recursive implementation of Fibonacci Series, the number of recursive calls grow exponentially because of duplication of work. In the previous set of experiments, there was a loop involved, and so one might argue that we might be including the overhead of loops that might skew our results. So, in this section we select this function that in itself makes a lot of recursive calls. Fibonacci series was one of choice. The result in Fig 3 reconfirms the findings in the previous sub-section.

4.1.4 *Program that makes repetitive calls to the same function that does nothing:*

In the experiments performed in all the previous subsections, we were using recursion. So it was not clear whether the problem lies with how recursion is implemented in the system or how function calls are handled in each of the language implementations. To comment on this aspect, in this sub-section, we concentrate only on the calls. The function called has an empty body and the function is repeatedly called in a loop. The number of iterations made in the loop is varied in the experiment. As is evident from the Fig 4, the time taken by Ruby increases rapidly as the number of function calls increase. Since very little coding has gone into this set of programs, the file sizes here do not vary considerably, Java source being 234B while Ruby code is 194B.

From the above set of experiments, we can conclude that calls to functions, and recursion as a result, are much slower in Ruby as compared to Java. But as is evident from Fig 2, in terms of code length, even in these small programs, Ruby gains over Java. This further asserts the argument that Ruby code is compact as compare to Java.

4.2 Searching, Parsing and Sorting

In this sub-section, we try to evaluate and compare the performance of the two languages when it comes to the very common problems of searching and sorting. We also look how these language constructs and libraries can be used for parsing large text files. We look at very simple and common techniques for this analysis.

4.2.1 *Searching and Parsing*

By comparing searching and parsing, we are concentrating more on evaluating File I/O and String manipulation techniques of the two languages. In searching, we simply do a linear search through the file using string comparison to look for the patterns. The file is first read into an array of strings and then the array is searched for the required string. In parsing, we use regular expressions to look for the patterns. In ruby, it is easier to code for these as the regular expression is built into the language, whereas in Java, it is part of the library. We have also evaluated how Perl performs in a similar scenario to have it as a baseline while performing the comparisons. Perl is very famous amongst programmers for these purposes of searching and parsing, and so with this analysis, we can not only compare Ruby and Java, but also have an idea of where they stand in comparison to Perl which is the present state of the art in parsing and searching. In these set of experiments, we have used somewhat larger files with size starting from 10MB going up to 100MB so that we get a feel how these programs perform when they are faced with considerable amount of data. In this sub-section we also consider a program that finds out the anagrams from a dictionary of English words. This involves complex tasks of signing every word, then sorting the words based on their signature and finally squashing and producing the output so that the anagrams are listed together in the same line.

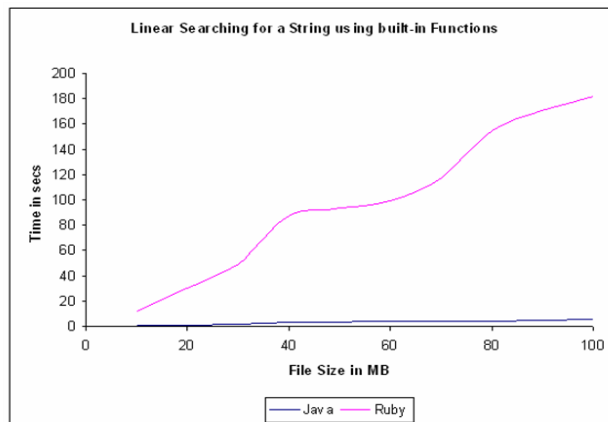

Figure 5: Linear Search through a text file

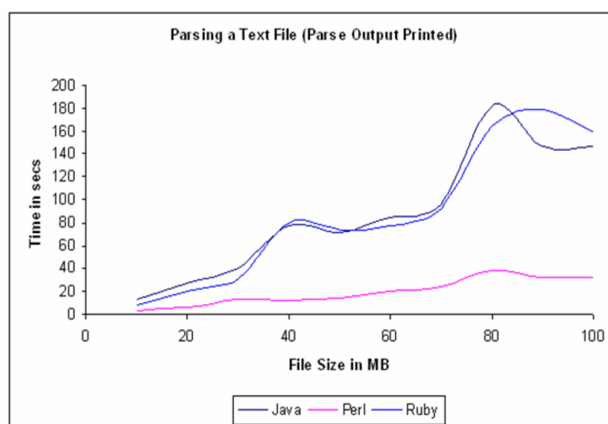

Figure 6: Parsing a text file

From Fig 5 it is evident that Java outperforms ruby by a huge margin even with file size of 10MB and the

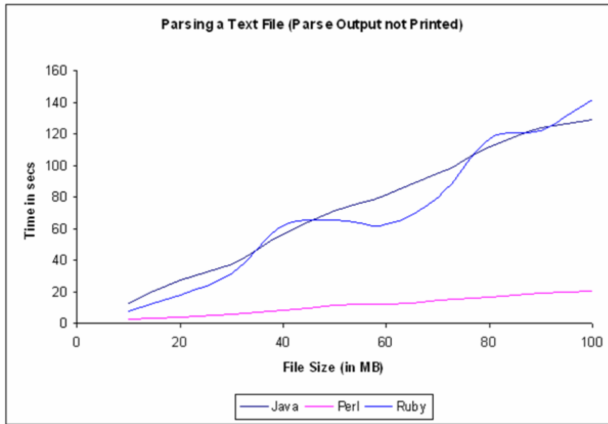

Figure 7: Parsing a text file with no output printed

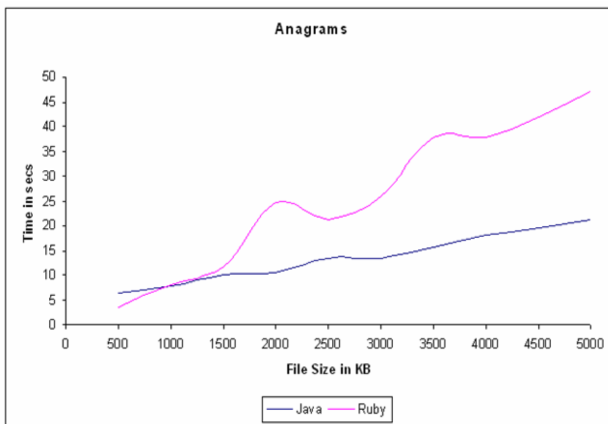

Figure 8: Anagrams from a dictionary of English Words

margin grows with the increase in file size. So, even though Ruby provides an easy approach towards solving the problem that eases the burden of the programmer, performance wise, Java outperforms Ruby by a considerable margin. Next we look at results of parsing a text file. We take Perl as the baseline. Again from Fig 6, it is evident that Java performs almost as good as Ruby and is way off when compared to Perl. In this program, the parsed output was printed. So just to find out whether the I/O is by any means offsetting the results, we re-run the same program with the output removed. The results can be seen in Fig 7. It is evident that the output didn't have much of a bearing in the results of Fig 6. So even though regular expressions are built into Ruby, still its performance w.r.t time is really poor. This is further supported by the anagram program whose performance can be seen in Fig 8. This program involves complex number of steps as discussed in the preceding paragraph. Programming this in Ruby is really easy as everything seamlessly integrates into the language. On the other hand, the program in Java requires more lines of code and heavily uses the library classes. But as evident from Fig 8, Java outperforms Ruby in most of the cases. One thing to be noted though is that the memory requirement of the Java Program is very high as compared to the Ruby

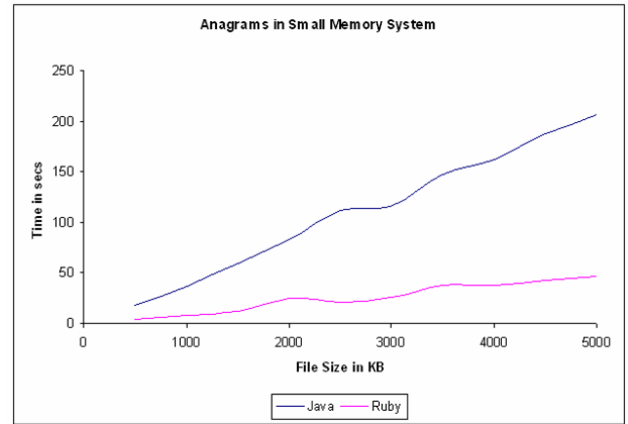

Figure 9: Anagrams in a system with small amount of memory

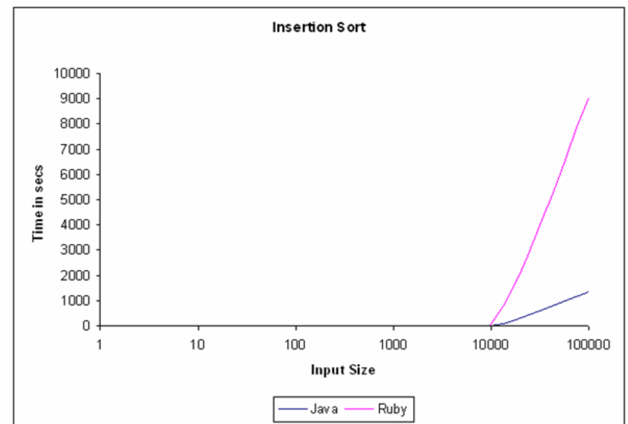

Figure 10: Insertion Sort on an Array of Integers

program, almost in all cases, about 9-10 times higher in average. As a result, when the same program is executed in a machine with less available memory, the scenario changes dramatically as is evident in Fig 9. This can be attributed to cache misses and the overhead associated with paging in and out. We tried explicitly invoking the Garbage Collector in Java after we finished using huge data structures, but that too did not help. As a general observation, we saw that when the programs require a large number of objects and data structures of huge size, the memory requirement of the Java programs increases a lot.

4.2.2 Sorting

For comparing implementation of sorting algorithms, we look at two algorithms with different time complexities. Insertion Sort ($O(n^2)$) although having a quadratic time, is popular because of the ease of insertion into a sorted array. We compare the Merge Sort as well as the Quick Sort algorithm for looking at algorithms with $O(n \log n)$ time. For the sake of completeness, we also compare the performance of the built-in sorting methods available with the language libraries. The version of Quick Sort we implemented selects the middle element

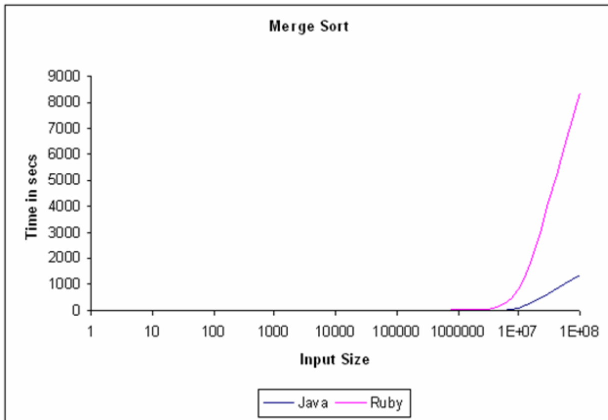

Figure 11: Merge Sort on an Array of Integers

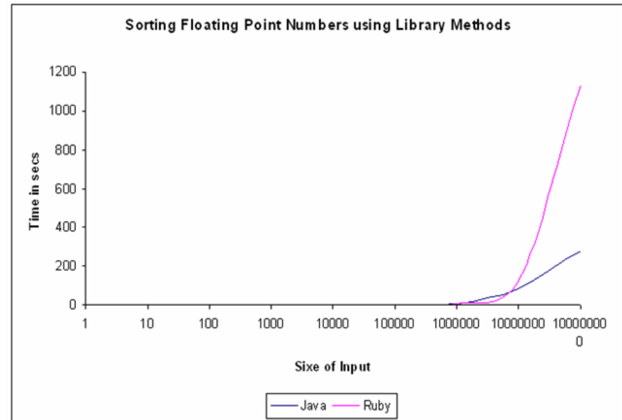

Figure 13: Sorting and array of Floating Point numbers using library routines

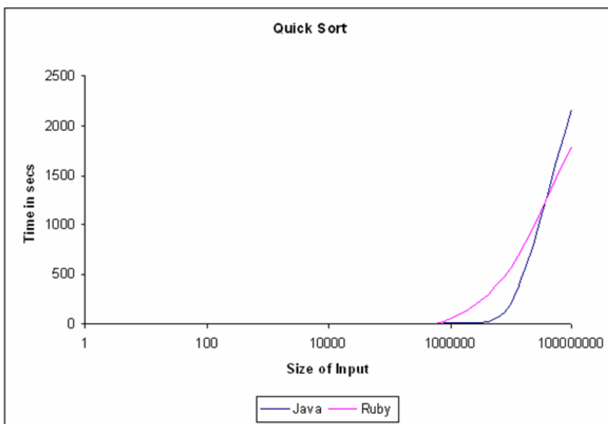

Figure 12: Quick Sort on an Array of Floating Point Numbers

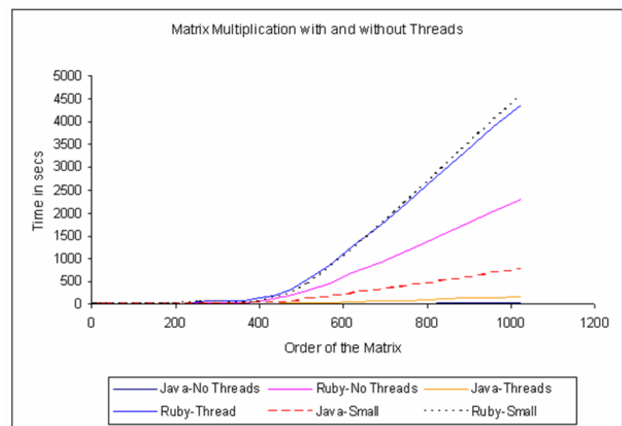

Figure 14: Matrix Multiplication with and without Threads

of the sub-array as the pivot. This is done so the Quick Sort does not reduce to an (O(n^2)) algorithm for already sorted and reverse sorted arrays.

The two algorithms chosen use two different approaches, Insertion Sort is iterative while Merge Sort is recursive. So we can test how the two languages perform when dealing with huge arrays of numbers using iteration and using recursion. From Fig 10 and Fig 11 it is evident that Java outperforms Ruby in both the cases and the difference becomes much more significant as the input size increases. A similar trend can be seen even in Quick Sort, as in Fig 12, which is also a recursive Divide-and-Conquer technique. Ruby relieves the programmer of the burden of implementing the sorting algorithms as the arrays have a built-in sort method. A similar sort method is also present in the Java library but it cannot deal with primitive types as it is part of the Collections framework. So with wrappers around primitive data types, we can sort arrays using the Java Library, but it is much more convenient the way this is integrated to the Ruby language. But from the performance perspective, Java again is the clear winner as is evident in Fig 13.

4.3 Threads

In this sub-section, we evaluate how the threads are handled in both Java and Ruby. Both the languages provide built-in support for threads that makes the life of a programmer much easier when dealing with multiple threads as compared to other languages such as C or C++. In this sub-section, we compare the overhead of adding threads in the two languages. To find this out, we use a simple matrix multiplication program where each element in the resulting matrix is calculated by a new thread. Therefore, we spawn as many threads as there are elements in the resulting matrix. Each thread simply multiplies the corresponding elements in the input matrices and adds the products. The work done by each thread is minimal and hence with this setting we can easily concentrate only on the overhead introduced by the spawning of threads. For the purpose of our experiments, we use square matrices of the same order as is seen in Fig 14 which plots the order of the matrices along the x-axis. As is evident, as the threads are created and introduced in the system, the increased overhead is significant in case of Java (increases the execution time by more than 5 times) whereas it is only less than twice

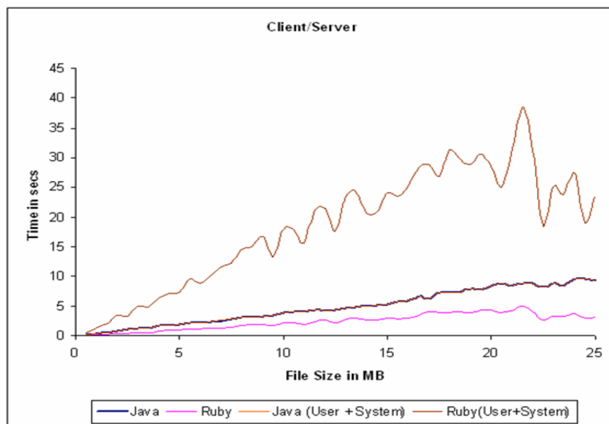

Figure 15: Client Server Program transferring a file from the server to the client using TCP Sockets

in Ruby. This is because the threads are built into the Ruby interpreter rather than being wrapper classes as in Java. As a result, the memory requirement in Java is more compared to that in Ruby. Hence, when the same set of programs are run in a system with smaller memory, the performance of Java decreases as seen in Fig 14.

4.4 Network Programming

In this sub-section, we aim to evaluate the networking features of the two languages under consideration. Both Ruby and Java provide libraries that helps write network programs with ease and comfort. To compare this feature, how they perform in terms of time, we use a very simple client-server program in which the client that connects to the server to obtain a file. Once the connection is set up with the server, the server reads the file, and sends it to the client and the connection is terminated. In order that the network characteristics doesn't affect the performance of the programs, we have the client and server running on the same host. In both the programs, the client and the server communicate using abstractions of BSD Sockets [5, 12]. We use TCP Sockets in both the languages. We take average of three runs of the programs. Fig 15 shows the results obtained when the time taken by the programs is plotted against the size of the file transferred. The results obtained are pretty interesting. As is evident from Fig 15, when we compare only the user and system times of Ruby and Java (marked in the fig simply as Ruby and Java), Ruby seems to perform better as compared to Java. But this is a very misleading result. In all the previous experiments, the system time is generally very small in comparison to the user time and so it has been neglected in all the previous graphs. But in this experiment, we found that the system time in the ruby programs considerably exceeds the user time. So, in Fig 15 we also plot the combination of user and system time (marked by the graphs user+system). As a result, we can see that Java again performs better than Ruby. The lines for Java and Java (User + System) almost overlap, but the one marked Ruby (User+System) is much higher. To delve into the reason for this anomaly, we obtained a detailed trace of the programs. The Ruby program shows very high number of context switches (about 1000 times more than that in Java), and it is this context switching overhead that increases the system time in case of the Ruby programs. But we could not find an explanation for the astronomically high number of context switches.

5. CONCLUSION

This paper compares Ruby with Java based on various aspects. Ruby is a purely Object Oriented programming Language whose syntax is influenced by Perl and is aimed at increasing programmer productivity. The syntax of the language is very terse and mellifluous. Various features such as built-in re-sizable arrays and hashes, regular expressions, and advanced high-level constructs such as code blocks, ease programming various complex tasks. Although Ruby provides various features that increase programmer productivity. But from the performance perspective, Java outclasses Ruby by a huge margin. Ruby is interpreted and hence it is fast when a code is executed once. Java on the other hand has an initial overhead of verification of classes, loading the classes into the JVM and just in time compilation. But once this is done, Java performs better when the same tasks are performed repeatedly. Hence, for small tasks, Ruby performs better than Java, but this gain vanishes in case of large tasks. From this study, we can conclude that Ruby is preferred from the programmer's point of view but Java is better suited performance wise for huge computational tasks. If some mechanism is developed that can optimize the performance of Ruby as it is now, then it could easily become the first choice language for the programmers.

6. SUMMARY

In this paper, we performed a comparison of Ruby with Java based on various approaches. As is pointed out in Section 2, there has been very little work till now comparing the two languages in a thorough and general manner. We would give a grade 1 to our project as it satisfies all the conditions as stated. Ruby is a new language that we learned for the purpose of this work. A lot of effort has to be put not only in learning the various aspects of the language but also writing programs in both the languages. For the sake of comparing the performance, the two implementations had to be made as similar as possible. After the programs were written, they were executed, performance results recorded and a thorough analysis of the results has been provided. We have tried to compare almost all the important features of the languages, like loops, function calls, recursion, various common tasks using the language features as well as the libraries available, threads and so on. We provide empirical results in all the comparisons. As an outcome, we have this paper, which gives a thorough analysis and comparison of the two languages which will help a Java programmer learn the differences between the two languages as well as compare the performance of two languages. This project provides some interesting insights that is beyond the scope of the course.

REFERENCES

[1] Ruby versus Smalltalk versus Objective-C versus C++ versus Java versus Python versus CLOS versus Perl5 versus XOTcl. *http://www.approximity.com/ruby/Comparison_rb_st_m_java.html*.

[2] Barry Burd. Ruby on Rails vs. Java: An Expert Roundtable. *http://javaboutique.internet.com/reviews/ruby/*.

[3] [dmh2000] - Essays. *http://javaboutique.internet.com/reviews/ruby/*.

[4] Dave Thomas et. al. *Programming Ruby - The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc., http://www.ruby-doc.org/docs/ProgrammingRuby/, second edition, October 2004.

[5] M. Tim Jones. Socket Programming in Ruby. *https://www6.software.ibm.com/developerworks/education/l-rubysocks/l-rubysocks-a4.pdf*.

[6] Claude Kaiser, Pradat-Peyre Jean-Fran, Sami, and Pierre Rousseau. Comparing Java, C# and Ada Monitors Queuing Policies: a Case Study and its Ada Refinement. *Ada Lett.*, XXVI(2):23–37, 2006.

[7] Eric Potratz. A Practical Comparison between Java and Ada in Implementing a Real-time Embedded System. In *SigAda '03: Proceedings of the 2003 annual ACM SIGAda international conference on Ada*, pages 71–83, New York, NY, USA, 2003. ACM Press.

[8] Lutz Prechelt. An Empirical Comparison of Seven Programming Languages. *Computer*, 33(10):23–29, 2000.

[9] Evaluation: Moving from Java to Ruby on Rails for the Centernet Rewrite. *http://rewrite.rickbradley.com/pages/moving_to_rails/*.

[10] Ruby - a Programmer's Best Friend. *http://www.ruby-lang.org/en/*.

[11] Ruby on Rails - Web Development that doesn't hurt. *http://www.rubyonrails.org/*.

[12] Lesson: All about Sockets - The Java Tutorial. *http://java.sun.com/docs/books/tutorial/networking/sockets/index.html*.

[13] Java technology - the Source for Java Developers. *http://java.sun.com/*.

[14] Bruce Tate. From Java to Ruby: Risk. *http://www.infoq.com/articles/From-Java-to-Ruby–Risk*.

[15] Jason Voegele. Programming Language Comparison. *http://www.jvoegele.com/software/langcomp.html*

[16] David A. Wheeler. Ada, C, C++, and Java vs. the Steelman. *Ada Lett.*, XVII(4):88–112, 1997.