Performance Analysis and Profiling Implementations of Languages

Joyesh Mishra
Department of Computer Science and Engineering
R.V. College of Engineering

0. Abstract

Language optimization and execution speed determine the efficiency of an application to satisfy certain specific requirements. With the advent of low-level as well as high level languages, it is certainly important to use the best suited language which is not only easy to use and understand, but also efficient from a system point of view.

Low Level languages have been devised to replace the problems faced in using, debugging and hardwiring assembly languages. High Level languages were devised due to some inherent problems in these low-level languages, which required modifications to the thinking ideologies of low-level language programmers. Hence, more complex languages were written so that ultimately the end-user who is an application developer can build and develop programs much faster and easier. It was also taken care to make the languages less prone to bugs so that debugging remains no more a tedious job.

We compared and contrasted highly popular application languages – C, C++, Java and scripting languages – Python and Perl. Standard Programs were written and executed to profile the implementation of these five languages. Their start-up times, which include the loading time, were compared and plotted. Basic algorithms were written and executed to evaluate the efficiency of language implementations.

1. Introduction

In earlier times, programming languages were often designed to reflect the internal hardware structure of a system. Low Level languages are therefore often described as machine-oriented languages. Most such languages have mnemonics, which depict the way the hardware operates. These languages are often not completely portable. The programmer has more control as the language is near to the hardware. Due to such intricacies involved, many-a-times developers find it difficult to learn, use and debug low-level languages.

High Level languages were designed keeping in mind the difficulties posed by low level languages. Portability and ease of programming were the primary concerns of developers of these languages. Program realization time i.e. the time required to implement an idea into an application was given importance. Thus, programmers could easily abstract the real world problem into distinct classes that made solving them easier.

Scripting languages were built to further ease the implementation. These languages are often interpreted and executed as batch operations. With the introduction of object-oriented concepts in scripting languages, they have become the language of choice for application developers.

But, the feature additions often come with compromises. Users and programmers remain oblivious to them due to which the application does not meet the stipulated requirements. For example, mission critical and real time applications will vary greatly from user-driven business applications. There is no one language that will solve all the purposes – but a collection, which has to be used scrupulously according to the requirement.

2. Profiling Implementations

Considering popular languages like C, C++, Java, Perl and Python helps in getting an insight on why these languages became popular over numerous others. It also brings out certain innate qualities of these languages for which they should always be preferred over others. We considered three important parameters to compare these languages

a. Loading Time – Time consumed in starting an application
b. Basic Algorithmic Computation
c. Program Size for typical applications

Start-up time often is critical to most real time applications. We measure the basic average time required by simplest applications written in these languages. Next, we implement two of the most commonly seen concepts in computer science viz. binary tree implementation and recursive algorithm in these languages. The programs are profiled and plotted against each other to show statistical analysis. The average execution time is calculated by executing all the implementations number of times. It is also observed that under certain conditions, the profile of a certain language differs according to the number of times it is executed, but we will discuss only for a fixed number of executions, which has been kept constant across the five languages.

Object oriented programming, abstraction of entities and re-usage of existing libraries have been key features of modern high level and scripting languages. Hence, the developed applications were compared against each other from the view of program size. It may be assumed that the number of lines required in writing a program in certain language has the highest weightage in determining the ease of using the language. It may also be argued that putting the solution in form of program may take significant time and hence makes the former case less effective. But we have considered that the resources implementing applications using such languages know all its attributes well and have optimized it to the best possible level.

The measurements to establish a comparative analysis have been tested on a **2.4 GHz dual-core Intel** processors operating on memory devices consisting of **1 GB RAM** and **80 GB SATA** disk drives. Linux distribution with kernel version **2.6.18.1** was chosen to carry out the performance evaluation. The programs have been compiled with standard available compilers. For C and C++, gcc and g++ were chosen.

3. Loading Time Analysis

The simplest program should ideally consume the least time to load. “Hello World” is considered to be the simplest

program that can be written. We measured the execution time of the hello world program. This time is though not equivalent to startup time. But, it is the nearest equivalent that could be achieved.

Each program should print "hello world" and exit. The program is executed n times in a loop. A simple shell wrapper is used to execute the program. The average CPU time consumption is measured. As most hello world programs are fast, it did not make much sense to profile the memory usages. Only CPU times (in seconds) were measured and plotted.

C++

No. of Executions	Avg. CPU Time
500	2.55

```
#include <iostream>
using namespace std;
int main() {
   cout << "Hello World" << endl;
   return 0;
}
```

C

No. of Executions	Avg. CPU Time
500	1.2

```
#include <stdio.h>
int main() {
   printf("Hello World\n");
   return 0;
}
```

Java

No. of Executions	Avg. CPU Time
500	106.8

```
public class HelloWorld {
   public static void main(String args[])
   {
    System.out.println("Hello World");
   }
}
```

Python

No. of Executions	Avg. CPU Time
500	15.75

```
#!/usr/bin/python
print "Hello World"
```

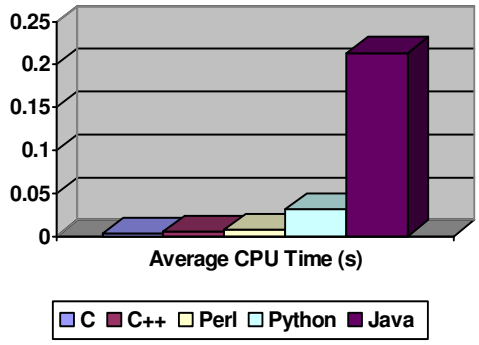
Perl

No. of Executions	Avg. CPU Time
500	3.8

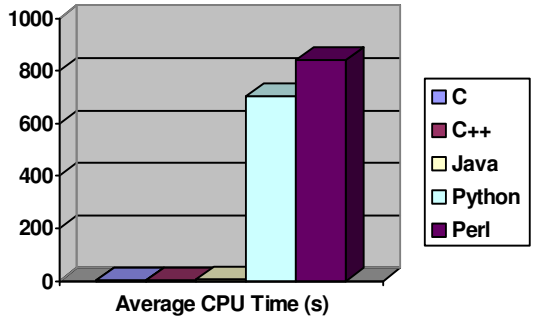
```
#!/usr/bin/perl
print "Hello World\n";
```

Due importance has been given to the flushing of output buffer. Hence, the statements, which print the output, have been terminated with '\n' or other syntactic equivalents.

Ratio	Program	Avg. CPU Time
1	C	0.0024
2.3	C++	0.0051
3.5	Perl	0.0076
14.6	Python	0.0315
97.7	Java	0.2136



Comparison Chart for Startup Time



Comparison Chart for Avg. CPU Time for Factorial Calculations

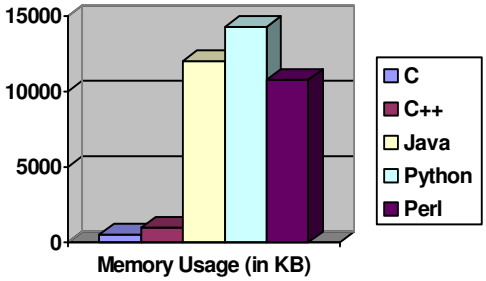
4. Recursive Algorithm Statistics

The most common example of recursive algorithm is the computation of factorial for a positive integer. The pseudocode is as follows:

factorial(n)
If n <= 1 → 1
*else → n * factorial(n-1)*

Other typical examples of recursion are Fibonacci series and Towers of Hanoi. The computation of factorial for an integer was implemented similarly on all the five languages.

Ratio	Program	Avg.CPUTime	Memory Usage (In KB)
1	C	2.65	501
1	C++	2.69	950
1.8	Java	4.94	12,009
260	Python	702.69	14,337
318	Perl	840.16	10,797



Comparison Chart for Memory Usage for Factorial Calculations

Python

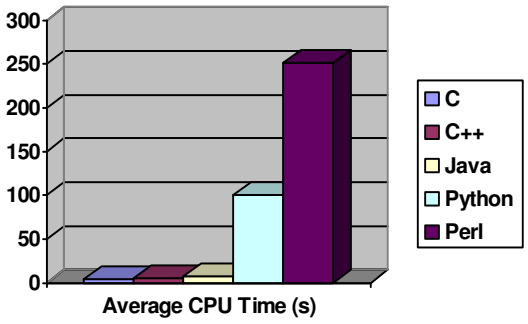
```
def Factorial(n):
    if n < 2: return 1
    return n * Factorial(n)

print "Factorial(10):", Factorial(10)
```

No. of Execs	Avg.CPUTime	Memory Usage (in KB)
16	702.69	14,337

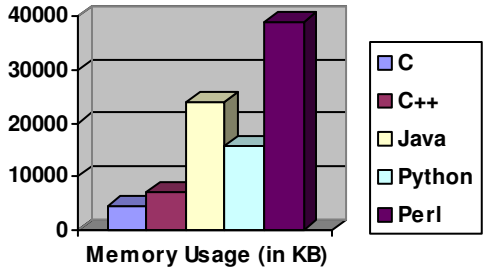
5. Binary Trees Implementation

The benchmark for binary tree implementation measures the allocation and de-allocation of binary trees. Binary trees of depth 16 were implemented. Regardless of the language chosen for implementation, each program defined node structures for tree, appropriate methods or procedures to manipulate the structure and allocate and de-allocate memory upon insertion and deletion of data. The following were the statistics recorded:

Ratio	Program	Avg.CPUTime	Memory Usage (In KB)
1	C	3.8	4,499
1.3	C++	4.5	7,023
2	Java	7.0	24,014
27	Python	100.06	15,804
66	Perl	251.7	38,882



Comparison Chart for Average CPU Time in execution of Binary Trees

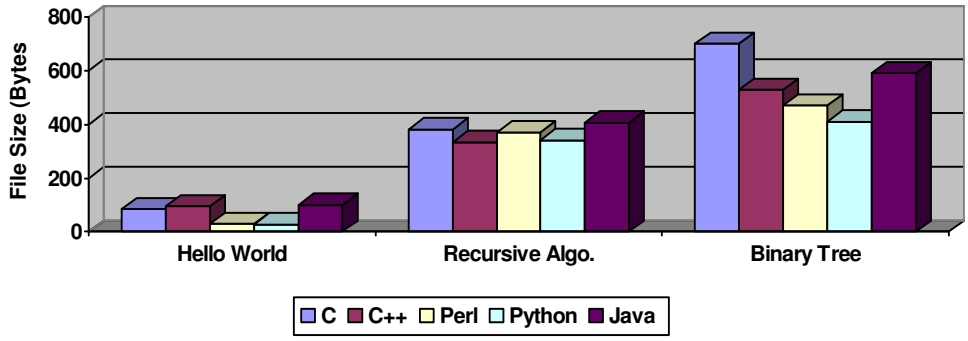


Comparison Chart for Memory Usage in execution of Binary Trees

6. Program Length/Size Comparison

As discussed earlier, often it is assumed that the shorter the program, the easier it tends to be to write it. Reusability tends to make the length and size of typical programs much smaller. This is exactly the reason why so many user-defined libraries exist. It also corroborates the existence of dll's and popularity of java as language of application development. The above three implementation were written using vi editor and saved in the Linux File System. The unnecessary comments, white spaces etc were removed and the file was compressed using gzip (one of the open source compression tools). Following are some of the observations (All file sizes are in bytes):

Program	HelloWorld	Factorial (Recursive)	Binary Tree
C	85	382	701
C++	95	334	529
Perl	30	371	471
Python	27	341	411
Java	101	407	592



Comparison Chart for Program Lengths in Various Languages

7. Discussions and Analysis

From the above results, we can observe few trends about the evolution of programming language. The results corroborate the fact that low or middle level languages like C are going to stay even when High Level Languages like Java or Scripting Language like Python have been adopted by many programmers. The profiled implementations for C programs have the least execution time (Average CPU Time) and lowest memory usage to carry out the necessary computations. An important point to be noted here is that both the execution time and memory usage include allocation and de-allocation of necessary memory to perform relevant calculations.

However, it should be noted that with the advent of scripting and interpreted languages like Perl and Python and with sophistications such as object-oriented concepts available in them, it is becoming more and more easier to quickly write programs efficiently from the user effort perspective. This fact is supported by the fact that the smallest programs (compressed in this case) are produced by perl and python. Though this fact is neutralized to a certain extent by the fact that programs using extensive third party library support will tend to increase in memory usage and CPU Time (because time to load and unload these extra libraries will be counted in), but proper caching mechanism and memory de-referencing techniques help to achieve a much faster execution speed.

8. Conclusion

Reiterating it again, there is no one language for making the best application possible. Each level of language has their merits and demerits. While C gives the best execution time and memory usage, it is not portable completely. Also, the programs tend to become bulkier and heavier with increase in complexity. On the other hand high-level languages such as C++ and Java are though easy to write and maintain, they consume memory and CPU time to a lot extent. Especially, in case of Java, due to the Virtual Machine, which is responsible for its portability, the startup

time is highest. Scripting languages are elegant to write and stand in between the two levels.

9. References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, 2/e, The MIT Press 2001.

[2] Aaron M. Tenenbaum, Yeddiyah Langsam, Moshe J. Augenstein. Data Structures using C, Prentice Hall 1989.

[3] Mark Allen Weiss. Data Structures and Algorithm Analysis in C++, 3/e, Addison Wesley 2006.

[4] Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language, 2/e, Prentice Hall PTR 1988.

[5] Bruce Eckel. Thinking in C++, 2/e, Prentice Hall 2000.

[6] Bruce Eckel. Thinking in Java, 4/e, Prentice Hall PTR 2006.

[7] Bjarne Stroustrup. The C++ Programming Language, 3/e, Addison-Wesley Professional 1997.

[8] John M. Zelle. Python Programming: An Introduction to Computer Science, Franklin Beedle & Associates 2003.

[9] Larry Wall, Tom Christiansen, Jon Orwant. Programming Perl, 3/e, O'Reilly Media 2000.

[10] Damian Conway. Object Oriented Perl: A Comprehensive Guide to Concepts and Programming Techniques, Manning Publications 2000.

[11] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools, Addison Wesley 1986.

[12] K. Reinholtz, Java will be faster than C++, ACM Sigplan Notices, 35(2): 25-28 Feb 2000.

[13] Benjamin Zorn, The Measured Cost of Conservative Garbage Collection Software - Practice and Experience 23(7): 733-756, 1992.

[14] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, Bill Pugh, P. Sadayappan, Jaime Spacco, and Chau-Wen Tseng. Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures.

[15] William Pugh. Evaluating Research on Software Design and Productivity, University of Maryland 2001.