Research Article

Tosin P. Adewumi*

Inner loop program construct: A faster way for program execution

https://doi.org/10.1515/comp-2018-0004
Received March 16, 2018; accepted May 18, 2018

Abstract: Loops are repetitive control structures in programming languages. They are used extensively in many algorithms. The for-loop and while-loop exist, where the former is repeated a number of times while the latter is repeated until a condition is met. Some have asked if re-arranging loops in certain ways can change a program’s speed to produce machine-independent optimisation. Therefore, this research sought to find out if there is any speed difference in a single loop of computations and a loop with an inner loop of same computations. Greater focus is on inner for-loop. The research used a comparative study method in order to evaluate the primary data obtained from running several tests in four popular programming languages: C, C#, Python and R. The Python implementations were further tested on Ubuntu 16 for comparison with results from Windows 10. Results established that, across all languages, there were more computations performed per unit time with an inner for-loop than no inner loop, meaning, given the same number of computations to perform, a loop with an inner for-loop will finish faster. The inner while-loop didn’t perform so well, though. This study will help developers in making better choices with programming language and style.

Keywords: for loop, while loop, programming languages, Windows 10, optimisation

1 Introduction

Speed is an advantage when it comes to computing and it is a very important factor computer and smart phone manufacturers are quick to showcase when marketing their latest products. The choice of programming language by developers or programmers is also heavily influenced by speed, though it is by no means the only criterion for selecting programming languages [1]. Besides choice of programming language in developing a software program, how lines of codes are implemented play critical role in determining the speed of the program. Two equivalent programs, in terms of purpose written in the same language but implemented in different styles can perform at considerably different speeds on the same platform. This research work focuses on a tiny but significantly influential aspect of programming, which are loops. The research answers the question “Is there any speed difference in a single loop of computations and one with an inner loop of same computations?” Hence, the objectives are to determine if a nested inner loop of one level makes a difference in terms of speed and therefore, computational attempts and whether there are noticeable differences across programming languages and operating systems.

The main contribution of this research is the provision of confirmation that there is general better speed performance or optimisation with inner for-loops than otherwise. This is true for the programming languages and operating systems tested, though Windows 10 is the predominant focus in this research. Some programming languages performed relatively better than others. This will help programmers in making better choices with programming style and programming language among other things. The research uses the method of comparative study to critically analyse the results from the various tests. The scope of the work will be restricted to a brief review of programming languages, loops, optimisation techniques and tests conducted in some popular programming languages and operating systems.

2 Literature review

Von Neumann computer architecture influenced programming languages available today since all-purpose computers in use today and many years ago are based on such architecture [2]. Step-wise program execution reflects the adaptation of programming languages to this architecture. Although there are many programming languages, they
are categorized into different paradigms based on their design features and they are suited for addressing different problem domains [1]. The TIOBE index, which rates popularity of programming languages rated C as 2\textsuperscript{nd} in popularity as of March 2018, Java came 1\textsuperscript{st}, Python was 4\textsuperscript{th} while C# was 5\textsuperscript{th}. It ranked R as 20\textsuperscript{th} on the popularity scale [3]. This ranking tends to have some changes from month to month and is therefore not constant. C is part of the procedural paradigm, C# is object-oriented while Python and R belong to the scripting paradigm [2, 4]. C is a compiled language, as C# is, though C# has some difference by being compiled into an intermediate language [1]. Python, like R, is an interpreted language, which are inherently slower than compiled languages by design [1].

In comparing different programming languages used in Economics, Aruoba and Fernandez-Villarverde [5] asked “Can we re-arrange loops in ways that change relative speed?” (p. 12). This question arose out of their research implementing and comparing the same value function iteration with grid search in 8 languages and tested on two operating systems. Answer to this question is essential but resources or research into this question appears to be scanty. Indeed, Nanz and Furia [4] pointed out the unsubstantiated answers to questions about properties of programs of programming languages in their in-depth research of comparative study of popular programming languages across different paradigms in Rosetta Code. A point corroborated by Prechelt [6] in his research comparing several languages empirically. The Rosetta Code repository contains program solutions of various tasks in many different languages, contributed to by thousands of users [4].

Loops, in programming languages, are control structures which make repetitive execution possible [2]. Loops are used extensively in programs and play a major role in the implementation of particle swarm optimisation (PSO) described by Kennedy and Eberhart [7, 8]. Loops are executed repeatedly for a number of times or until a given condition is met, as with for-loop or while-loop, respectively [2]. Both for-loop and while-loop exist in popular programming languages and may be used interchangeably. The while-loop has a variant in some languages, do-while-loop, which has a slight difference in logic or order of execution [2]. Since some languages are interpreted while others are compiled to machine-specific codes, there are advantages and disadvantages to such a system [1, 2]. Compiled languages have the advantage of fast execution, compared to interpreted languages like the scripting languages [1].

Loop optimisation techniques exist and Jubb [8] pointed out two of such: machine-independent and machine-dependent techniques [8, 9]. Identifying loop invariants, inverting the loop, loop unrolling and removing induction variables are some of the techniques under the machine-independent type [8, 10, 11]. That is, they can be applied to optimize loops, regardless of the hardware architecture in use. Machine-dependent techniques take advantage of hardware-related features of a computer to optimize loops. Flags, registers and memory addressing are some of the hardware features utilized in machine-dependent techniques [8].

Both machine-independent and machine-dependent optimisation techniques for loops have their disadvantages [8, 12, 13]. For instance, as noted by Jubb [8], large file binaries are generated when doing aggressive machine-dependent optimisation. On the other hand, with regards to machine-independent techniques, high optimisation costs can result from analysis of control flow within a program, according to [13]. Also, based on research by Booshehri et al. [12], loop unrolling has slight noticeable effect on power consumption. There have been ongoing efforts in cutting down or managing the various disadvantages each offers [8, 13].

3 Methodology

This research adopted a comparative study method in order to evaluate quantitative primary data obtained from running several tests. Tests were carried out on Windows 10 for each of the four programming languages used in the research: C, C#, Python and R. These languages were selected to implement the study based on their popularity, as indicated by the TIOBE index for March, 2018, and representation of some paradigms. C was compiled with two versions for greater validity, namely, version 6.0.34.1 of Orange C compiler and version 4.9.2 of GCC-like compiler of DevC++, C# and Python, version 3.6, were written and compiled or interpreted in Visual Studio 2017 community edition, respectively. In addition to the Python 3.6 version, version 2.7 was also implemented for further analysis. R, version 3.4.3, was written in R Studio. All tests were carried out using HP 2000 laptop with Intel Pentium 2.4Ghz clock speed, 4GB RAM and 64bit Windows 10 operating system. The dual boot facility of the machine provided Linux Ubuntu 16 to run exact Python 3.6 tests. The system uses only one processor. Running the experiments in four programming languages, with additional Ubuntu 16 tests for Python, helped to minimize threats of external validity to this work [4].

In the first instance, three similar programs of loops were written for all four languages: a while loop without
any nested inner loop, a while loop with one inner for-loop and a while loop with one inner while-loop. In order for ease of replication of this study, complete codes used for the four languages, for the three different programs, are presented in the appendices. Lines of code inserted for measuring elapsed time where put outside the loop so as not to interfere with the execution time of the loop being measured, thereby minimizing threats to internal validity, an indication to whether measuring things right. Further to that, while each program executed, it was ensured no other unrelated application was open and running. The research focused on elapsed time because this is what users generally experience when using systems and it is a common performance evaluation technique in computer science [14]. On the sideline, in the course of tests to discover best approach to measure elapsed time, it was discovered that adding an extra line of code impacted on the elapsed time, even when the line was a comment, though a comment caused lesser delay than a line to be executed.

Each loop was made to run over lengthy iterations and the time elapsed (being around 120 seconds on average) after loop termination was recorded. This was carried out twice for each case, making a total test of 240 runs, and an average calculated in each case. Measuring the average elapsed time this way enabled deductions of the average iterations and computations per second. The number of iterations (or outer iterations) each test is given is only important in letting each test run for considerable time for consistency so as to standardize all tests over a second through calculations. Each loop had a line of computation which is a square computation. The line of computation did not involve invariant calculation or induction variable whose value changes by a fixed figure in each iteration, as explained by Jubb [8]. Any type of simple or complex computation could also have been used. Equation 1 below gives the formula for deducing total computations from number of iterations per unit time. In the case of no inner loop, total computations equals outer iterations (or simply iterations).

\[ \text{Computations} = \text{Outer Iterations} \times \text{Inner Iterations} \]

In the second instance, a more practical program with real-world basis was tested in Python 3.6. The example may be considered a simplified lottery system, where millions of tickets have been sold and there's a jackpot number to be matched. One version of the program searches through tickets with only one while loop whereas the second program uses an inner for-loop to optimize the search. This simplified example conducts search in a sort of ordered list, though a binary search is well-suited for ordered lists [15]. Three instances of jackpot numbers were tested and conducted twice for each program, making a total of 12 runs. Mean values were taken in each case and tabulated in the next section in Table 3. The codes for the simplified lottery system, in Python, to win a jackpot is given below.

**Code Listing 1 - No Inner Loop**

```python
import time
def spin(jackpotno,maxticketno):
    ticketno = 0
    while ticketno < maxticketno:
        if ticketno==jackpotno:
            return
        ticketno+=1
    if __name__ == "__main__":
        starttime = time.time()
        spin(jackpotno=800000,maxticketno=90000000)
        print("Time ", time.time() - starttime)
```

**Code Listing 2 - Inner For-Loop**

```python
import time
def spin(jackpotno,maxticketno,innermax):
    ticketno = 0
    while ticketno < maxticketno:
        for j in range(0, innermax):
            if (ticketno*innermax+j)==jackpotno:
                return
        ticketno+=1
    if __name__ == "__main__":
        starttime = time.time()
        spin(jackpotno=800000,maxticketno=90000000, innermax=200)
        print("Time ", time.time() - starttime)
```

## 4 Data presentation, interpretation and discussion

Results of the tests conducted and averaged, in the first instance, are summarized in Table 1 2 and a line graph representation of the ratios is given in Figure 1 below. It can be observed that per unit time or in a second, all inner for-loop iteration totals for Python (both versions), except 2, did more computations than the benchmark of no inner loop. This is confirmed in the ratios expressed, as an average ratio slightly above 1.4 was maintained steadily from 100 inner iterations onward. Python 2.7 had the highest ratio relative to its no inner loop of all the languages, climb-
A somewhat steady advantage ratio is reached and main-
required to achieve the inner for-loop speed advantage and
ally, it is clear a certain minimum inner iteration number is
1 in some instances, compared with no inner loop. Gener-
languages by performing quite below it and was below ratio of
different from its inner for-loop counterpart across all lan-
er iterations became worse. Inner while-loop was clearly
same trend as Python 3.7 but its ratio steadied around 1.5.
ashigh as 1.5655 at 22 for Orange C compiler. R followed the
quite differently at 2 and 22 inner iterations, having ratios
as high as 2.7595. C# behaviour was similar in many
ways that to Python, however, an average ratio around
1.2 was maintained after 200 inner iterations. C is similar to
C# in maintaining an average ratio close to 1.2 but behaves
quite differently at 2 and 22 inner iterations, having ratios
as high as 1.5655 at 22 for Orange C compiler. R followed the
same trend as Python 3.7 but its ratio steadied around 1.5.
Python’s performance on Linux Ubuntu 16 was very simi-
lar to its behaviour in Windows 10 except that ratio for 2 in-
er iterations became worse. Inner while-loop was clearly
different from its inner for-loop counterpart across all lan-
guages by performing quite below it and was below ratio of
1 in some instances, compared with no inner loop. Gener-
ally, it is clear a certain minimum inner iteration number is
required to achieve the inner for-loop speed advantage and
a somewhat steady advantage ratio is reached and main-
tained over a wide range of inner iteration values. In the
second instance example, Table 3, displaying results of the
simplified lottery system, shows that the loop with inner
for-loop found the jackpot numbers faster than the no in-
ner loop counterpart in all cases.
Additionally, it can be observed that Python ran faster
on Linux Ubuntu 16 than Windows 10, judging from the
number of computations done per second. On Windows
10, Python was the slowest of the languages. C and C# were
neck and neck as the fastest of the languages. R ranked
third of the four languages on speed. Python’s and R’s
slow pace may be attributed to the nature of interpreta-
tion of the languages by design, as opposed to compilation
done by C and C# [2]. Related work done by prominent re-
searchers have shown similar tendencies like the immense
speed of C, which C# closely emulates [4, 16]. Nanz and Fu-
ría [4] also highlighted similar result in research done by
Prechelt [6].

Table 1: Computation and ratio comparison.

<table>
<thead>
<tr>
<th>Windows 10</th>
<th>Python 3.6 - VS</th>
<th>Python 2.7</th>
<th>C#</th>
<th>C - Orange</th>
</tr>
</thead>
</table>
|            | Mean Computa-
|            | tions/s | Ratio to No Inner Loop | Mean Computations/s | Ratio to No Inner Loop | Mean Computations/s | Ratio to No Inner Loop | Mean Computations/s | Ratio to No Inner Loop |
| No Inner Loop | 165,143 | 1.000 | 5,862,500 | 1.000 | 293,125,000 | 1.000 | 291,666,666 | 1.000 |
| 2X | 134,815 | 0.8164 | 3,235,556 | 0.5519 | 248,947,538 | 0.8693 | 404,444,000 | 1.3867 |
| 22X | 219,731 | 1.3306 | 11,863,704 | 2.0237 | 342,221,846 | 1.1675 | 456,595,989 | 1.5655 |
| 100X | 235,150 | 1.4239 | 14,979,422 | 2.5551 | 311,110,769 | 1.0614 | 337,036,667 | 1.1556 |
| 200X While | 155,870 | 0.9438 | 7,353,527 | 1.2543 | 323,555,200 | 1.038 | 323,555,200 | 1.1093 |
| 400X | 237,052 | 1.4354 | 16,177,760 | 2.7595 | 344,220,766 | 1.7143 | 330,158,367 | 1.1320 |
| 600X | 237,906 | 1.4406 | 16,177,760 | 2.7595 | 344,224,974 | 1.7143 | 332,419,726 | 1.1397 |
| 1000X | 231,275 | 1.4005 | 15,555,538 | 2.6534 | 337,036,667 | 1.1498 | 334,251,240 | 1.1460 |
| 2000X | 234,484 | 1.4199 | 15,555,538 | 2.6534 | 351,690,435 | 1.1998 | 335,638,174 | 1.1508 |
| 4400X | 234,167 | 1.4180 | 15,610,119 | 2.6627 | 348,932,078 | 1.1904 | 338,993,338 | 1.1623 |
| 44000X | 238,615 | 1.4449 | 15,275,268 | 2.6056 | 352,421,399 | 1.2023 | 342,221,846 | 1.1733 |

Table 2: Computation and ratio comparison.

<table>
<thead>
<tr>
<th>Windows 10</th>
<th>C - DevC++</th>
<th>R - RStudio</th>
<th>Python 3.6 - Ubuntu 16</th>
</tr>
</thead>
</table>
|            | Mean Computa-
|            | tions/s | Ratio to No Inner Loop | Mean Computations/s | Ratio to No Inner Loop | Mean Computations/s | Ratio to No Inner Loop | Mean Computations/s | Ratio to No Inner Loop |
| No Inner Loop | 291,666,666 | 1.000 | 6,948,529 | 1.000 | 13,400,000 | 1.000 |
| 2X | 449,999,996 | 1.5429 | 5,235,576 | 0.7535 | 4,375,564 | 0.3265 |
| 22X | 439,999,956 | 1.5086 | 9,268,508 | 1.3339 | 14,833,734 | 1.1070 |
| 100X | 272,058,684 | 0.9328 | 10,240,729 | 1.4738 | 18,383,818 | 1.3719 |
| 200X | 305,349,459 | 1.0469 | 10,372,064 | 1.4927 | 19,494,107 | 1.4548 |
| 200X While | 311,110,769 | 1.0667 | 6,913,573 | 0.9950 | 14,068,681 | 1.0499 |
| 400X | 311,288,729 | 1.0673 | 10,372,064 | 1.4927 | 19,728,976 | 1.4723 |
| 600X | 307,894,429 | 1.0556 | 10,550,713 | 1.5184 | 19,491,591 | 1.4546 |
| 1000X | 317,540,005 | 1.0887 | 10,240,729 | 1.4738 | 19,307,952 | 1.4409 |
| 2000X | 333,330,000 | 1.1428 | 10,304,724 | 1.4830 | 19,259,238 | 1.4373 |
| 4400X | 326,370,363 | 1.1190 | 10,467,859 | 1.5065 | 19,342,974 | 1.4435 |
| 44000X | 338,427,692 | 1.1603 | 10,176,078 | 1.4645 | 18,066,821 | 1.3483 |
Inner for-loop enhances program speed

5 Conclusion

This research focused on a small but significant aspect of programming: loops. Aruoba and Fernandez-Villarverde [5] had asked “Can we re-arrange loops in ways that change relative speed?” (p. 12) in their research comparing 8 programming languages used in Economics. To a large extent that question has been answered in this research. The objectives were to determine if a nested inner loop of one level makes a difference in terms of execution speed and, therefore, computational attempts, and whether there are noticeable differences across programming languages and operating systems. The research adopted a comparative study method in order to evaluate the primary data obtained from running several tests which were carried out for the four programming languages used: C, C#, Python and R.

Intuitively, one might have thought the ratio of the execution time of the inner for-loop to no inner loop programs would have been somewhat 0.9 because of the extra “for” line of code in the inner for-loop program, since extra code lines impact on execution time. However, this proved not to be the case as we can clearly see from the results. We observed that there are more computations performed per unit time with an inner for-loop than no inner loop. This also means, given the same number of computations to perform, an inner for-loop will finish earlier than one without it. This machine-independent approach to loop optimisation will likely have even more impressive performance when combined with machine-dependent optimisation techniques. The story of the inner while-loop is clearly different from its counterpart, in that it performed quite below it and was below ratio of 1 in some instances. The inner for-loop steady ratio for the C family of compiled languages (C and C#) was around 1.2 while that for the scripting languages was around 1.4, except Python 2.7, though the C family performed far better in execution speed, being compiled languages, as noted by Ghezzi and Jazayeri [2]. The worst ratio was 0.3265 for Python on Linux Ubuntu for 2 inner for-loop iterations.

This research has mainly provided confirmation that there is better performance, generally, in terms of execution speed with inner for-loops than no inner loop and this is true for the programming languages and operating systems tested. Some programming languages performed relatively better than others, however. The inner for-loop can be used in many instances to enhance execution speed of programs. Further work can be carried out on inner while-loops and over wider range of inner iterations. More tests can also be carried out on Ubuntu Linux and the Mac, besides Windows 10, to have a definitive study on which system is the fastest. Additional programming languages may be used to test this program construct for further external validity analysis. This will help programmers in making better choices with programming style and programming language.

References

A Appendix - Python codes

No Inner Loop

```python
import time

def spin():
    nonce = 0
    while nonce < 46900000:
        nonce += 1
    if _name_ == "_main_":
        starttime = time.time()
        spin()
        print("Time ", time.time() - starttime)
```

Inner For-Loop

```python
import time

def spin(num_particles):
    i = 0
    while i < 404:
        for j in range(0, num_particles):
            val = i^2
            i+=1
    if _name_ == "_main_":
        starttime = time.time()
        spin(num_particles=44000)
        print("Time ", time.time() - starttime)
```

Inner While Loop

```python
import time

def spin(num_particles):
    i = 0
    while i < 404444:
        j = 0
        while j < num_particles:
            val = i^2
            j+=1
        i+=1
    if _name_ == "_main_":
        starttime = time.time()
        spin(num_particles=200)
        print("Time ", time.time() - starttime)
```

B Appendix - C codes

No inner loop

```c
#include <stdio.h>
#include <time.h>

int main(void) {
    int op1, op2, op3;
    op1 = 0;
    op2 = 0;
    time_t starttime, passtime;
    starttime = time(NULL);
    while(op1 < 4044440)
    {
        op2 = op1^2;
        op1 = op1 + 1;
    }
    passtime = time(NULL);
    printf("Time %d", passtime - starttime);
}
```

Inner For-Loop

```c
#include <stdio.h>
#include <time.h>

int main(void) {
    int op1, op2, op3;
    op1 = 0;
    op2 = 0;
    time_t starttime, passtime;
    starttime = time(NULL);
    op1 < 4044440)
    {
        for(int i=0; i<200; i++)
        {
            op2 = op1^2;
            op1 = op1 + 1;
        }
    }
    passtime = time(NULL);
    printf("Time %d", passtime - starttime);
}
```

Inner While Loop

```c
#include <stdio.h>
#include <time.h>

int main(void) {
    int op1, op2, op3;
    op1 = 0;
    op2 = 0;
    time_t starttime, passtime;
    starttime = time(NULL);
    while(op1 < 4044440)
    {
        for(int i=0; i<200; i++)
        {
            op2 = op1^2;
            op1 = op1 + 1;
        }
    }
    passtime = time(NULL);
    printf("Time %d", passtime - starttime);
}
```
int main(void) {
    int op1, op2, op3;
    op1 = 0;
    op2 = 0;
    time_t starttime, passtime;
    starttime = time(NULL);
    while (op1 < 40444400) {
        op3 = 0;
        while (op3 < 200) {
            op2 = op1^2;
            op3 = op3 + 1;
        }
        op1 = op1 + 1;
    }
    passtime = time(NULL);
    printf("Time %d", passtime - starttime);
}

C Appendix - C# codes

No inner Loop
using System;
namespace Archicheck
{
    class Program
    {
        static void Main(string[] args)
        {
            UInt64 n = 0;
            UInt64 val = 0;
            var starttime = DateTime.Now;
            while (n < 4044440)
            {
                val = n ^ 2;
                n = n + 1;
            }
            var elapsed = DateTime.Now - starttime;
            Console.WriteLine("Time "+ elapsed);
        }
    }
}

Inner For-Loop
using System;
namespace Archicheck
{
    class Program
    {
        static void Main(string[] args)
        {
            UInt64 n = 0;
            UInt64 val = 0;
            var starttime = DateTime.Now;
            while (n < 4044440)
            {
                for (int i=0;i<200;i++)
                {
                    val = n ^ 2;
                    n = n + 1;
                }
                var elapsed = DateTime.Now - starttime;
                Console.WriteLine("Time "+ elapsed);
            }
        }
    }
}

D Appendix - R codes

No Inner Loop
nonce = 0
res = 0
Archicheck <- function()
{
  starttime = Sys.time()
  while (nonce < 4044440) {
    res = nonce^2
    nonce = nonce + 1
  }
  endtime = Sys.time()
  print(endtime-starttime)
}
CallArchicheck = Archicheck()

Inner For-Loop
nonce = 0
res = 0
Archicheck <- function(){
  starttime = Sys.time()
  while (nonce < 44440) {
    #ce = 0
    for (i in 1:200){
      res = nonce^2
      #ce = ce + 1
    }
    nonce = nonce + 1
  }
  endtime = Sys.time()
  print(endtime-starttime)
}
CallArchicheck = Archicheck()

Inner While-Loop
nonce = 0
res = 0
Archicheck <- function(){
  starttime = Sys.time()
  while (nonce < 4044440) {
    ce = 0
    while(ce < 200){
      res = nonce^2
      ce = ce + 1
    }
    nonce = nonce + 1
  }
  endtime = Sys.time()
  print(endtime-starttime)
}
CallArchicheck = Archicheck()