"Do Virtual Machines or Compilers Perform Better in Terms of Processing Time for Executing Code?"

By: Robert Knowles

Words: 3,700 (approximately)

INTRODUCTION

After nearly 50 years of annual improvements in the processing power of computers, it is understandable that many people assume that these improvements will simply continue.

In this essay, I examine the hardware and software limitations that the computer industry is now facing. If these limitations are not overcome, Moore's "Law" may finally be broken.

My essay investigates the improvements in CPU performance that we can achieve by changing the nature of the instructions that a CPU is required to process, rather than by trying to improve the power of the CPU itself.

My particular area of interest is the difference in the performance of Virtual Machines and Compilers when executing computer code. If we can optimize the nature of the tasks that we require a CPU to perform, we can extract higher performance from it.

Table of Contents

1 Are We Hitting a Brick Wall after 50 Years?

1.1 Moore's Law and Physical Bottlenecks

In the year 1965, electrical engineer Gordon Moore published the article "Cramming More Components onto Integrated Circuits" in the April edition of *Electronics* magazine. This article would become part of the bedrock of the computer processor industry[1].

In the article he forecasted that "The number of components on a given square centimeter of silicon would double every year"[2].

His forecast was uncannily accurate. **Figure 1** below illustrates Moore's "law" by plotting the how the number of transistors in microprocessors has increased over the last 40 years.

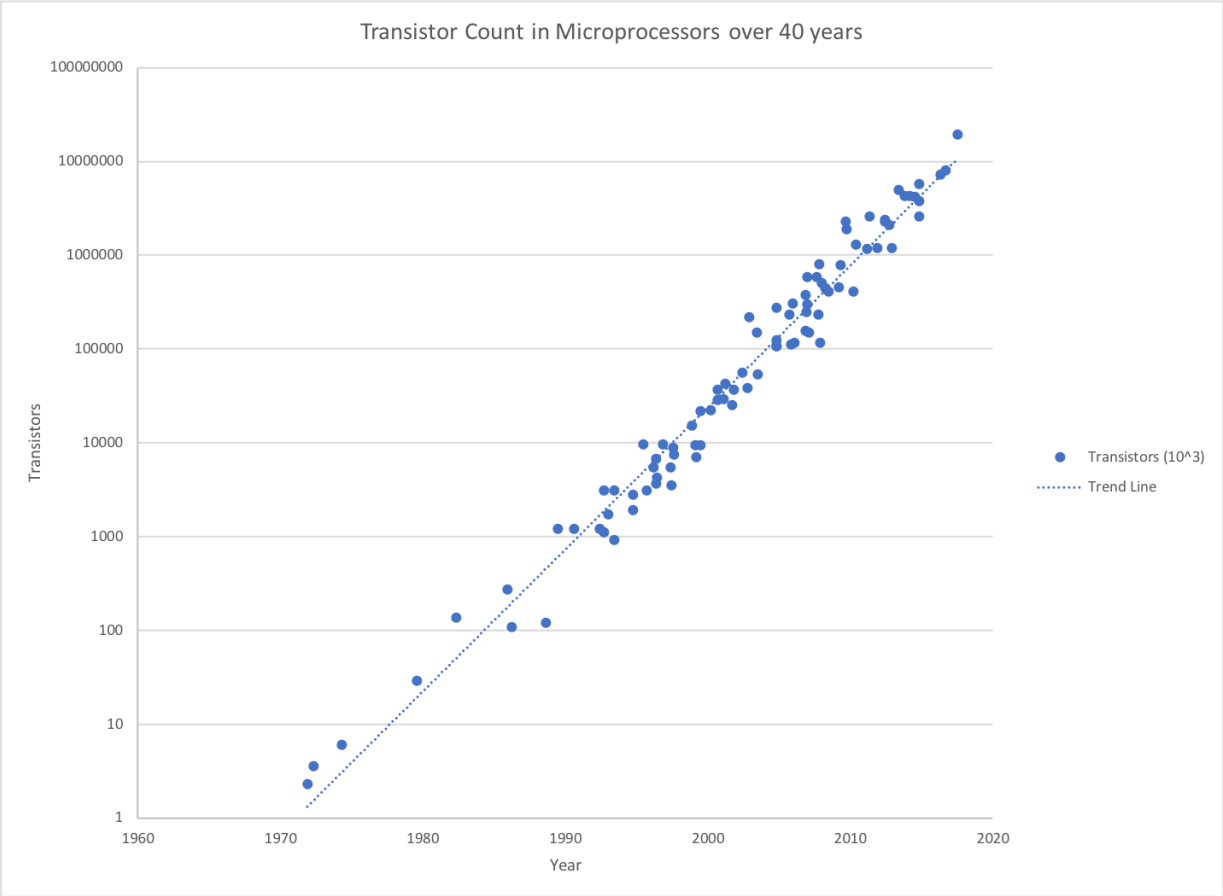

Figure 1: Transistor Count over the Period 1970 to 2018[3]

[1] Kelion, Leo. "Moore's Law: Beyond the First Law of Computing." *BBC News*, BBC, 17 Apr. 2015, www.bbc.com/news/technology-32335003.
[2] Moore, G.e. "Cramming More Components Onto Integrated Circuits." *Proceedings of the IEEE*, vol. 86, no. 1, 19 Apr. 1965, pp. 82–85., doi:10.1109/jproc.1998.658762.
[3] Rupp, Karl. *Github*, 2 Apr. 2016, github.com/karlrupp/microprocessor-trend-data.

Moore's Law is central to the computer processor industry because the number of transistors that you can fit in a CPU determines the processing power of that CPU[4]. **Figure 2** demonstrates the relationship between transistor count and processing power (using the industry standard SpecINT bench mark) for processors in the last 40 years:

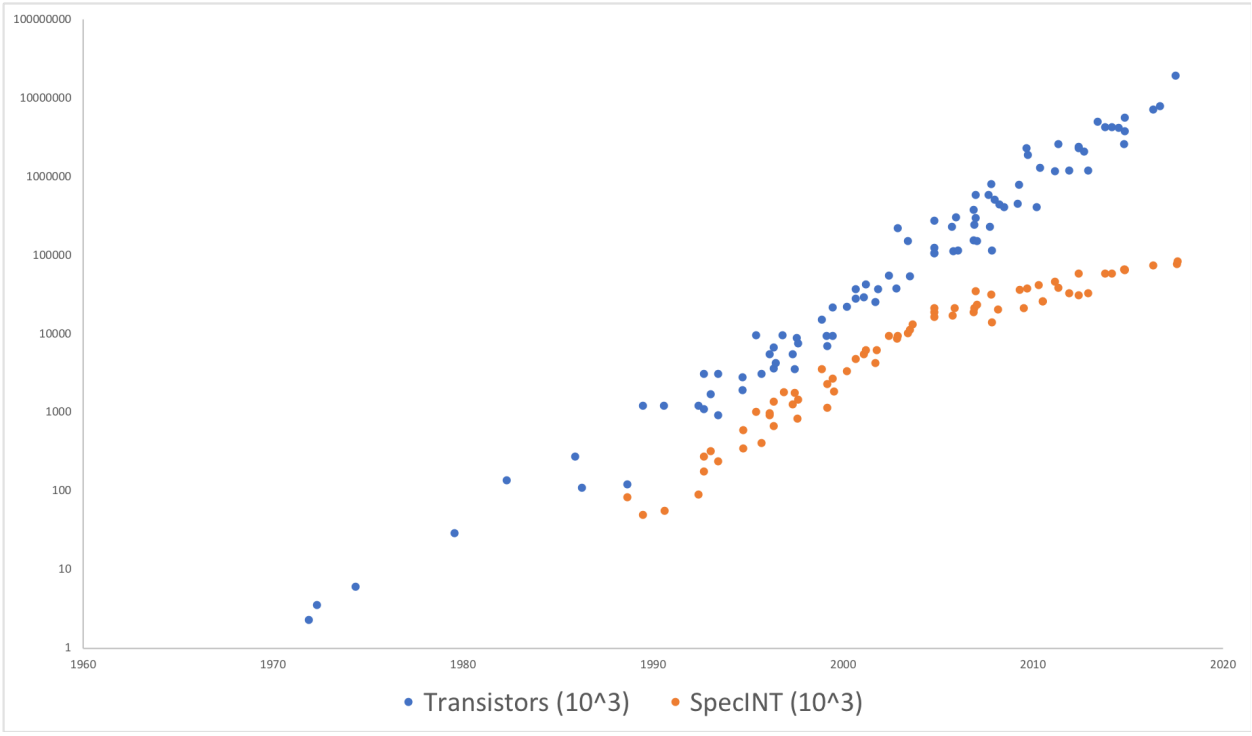

Figure 2: Transistor Count and Processing Speeds Over the Period 1970 to 2018

We are now reaching the limits of what can physically be achieved. In 2018 Samsung, Intel, AMD, Qualcomm and other companies were producing transistors that are only 100 atoms apart[5]. The enormous investments required to develop semiconductors of this complexity is one of the reasons why the top 10% of companies in this industry capture close to 60% of the revenues and more than 70% of the profits[6].

[4]Wirth, N. "A Plea for Lean Software." *Computer*, vol. 28, no. 2, 3 Feb. 1995, pp. 64–68., doi:10.1109/2.348001.
[5] Courtland, Rachel. "Intel Now Packs 100 Million Transistors in Each Square Millimeter." *IEEE Spectrum: Technology, Engineering, and Science News*, IEEE Spectrum, 30 Mar. 2017, spectrum.ieee.org/nanoclast/semiconductors/processors/intel-now-packs-100-million-transistors-in-each-square-millimeter.
[6]Partners, Alix. "Getting Moore out of Chips | Driving R&D Efficiency after 50 Years of Moore's Law." *Results-Driven Global Consulting Firm*, 2016, legacy.alixpartners.com/en/Publications/AllArticles/tabid/635/articleType/ArticleView/articleId/1663/Getting-Moore-out-of-Chips.aspx.

Further increases in the physical density of transistors may soon not be possible because, as the size of transistors approaches the atomic level, their performance suffers due to the properties of electrons such as Quantum tunneling[7] that results in insufficient heat dissipation.

Gordon Moore himself believes that the days of his "Law" may be numbered, observing:

> *"I think that's inevitable with any technology [that] it eventually saturates out. I guess I see Moore's law dying here in the next decade or so, but that's not surprising"*[8]

1.2 Software Compounds the Problem

Even if we were not facing hardware challenges to Moore's Law, we still might not be able to improve computing performance. This is because software is getting slower more rapidly than hardware is getting faster[9]. Industry professionals refer to this phenomenon as May's Law which specifically states that "Software Efficiency halves every 18 months, compensating Moore's Law"[10].

A good example of May's Law comes from the gaming industry - StarCraft (1998)[11] required a 90MHz processor while StarCraft 2 (2010)[12] which requires 2.8GHz processor. This represents a 3,000% increase in the processing power requirements of the game over a span of only 12 years!

If we had a better approach to software design, we might not need better hardware.

1.3 Three Variables to Consider

Processing time is a function of three factors - *cycle speed, cycles/instruction,* and *instructions:*

$$\text{CPU Time (Processing Time)} = \text{Instructions} * \text{Cycles/Instructions} * \text{Cycle Speed}$$[13]

Note that one of these a <u>hardware</u> constraint (because *cycle speed* depends on the number of transistors in the CPU) but the other two are <u>software</u> constraints.

---

[7] Kumar, Suhas. *Fundamental Limits to Moore's Law*. Stanford, 2015, p. 3, *Fundamental Limits to Moore's Law*.
[8] Courtland, Rachel. "Gordon Moore: The Man Whose Name Means Progress." *IEEE Spectrum: Technology, Engineering, and Science News*, IEEE Spectrum, 30 Mar. 2015, spectrum.ieee.org/computing/hardware/gordon-moore-the-man-whose-name-means-progress.
[9] Wirth, N. "A Plea for Lean Software."
[10] Ibid.
[11] Blizzard. "About Starcraft." 5 Nov. 2017, starcraft.com/en-us/.
[12] Blizzard. "StarCraft II System Requirements." *Blizzard Support*, 2016, us.battle.net/support/en/article/27575.
[13] Duluth, Minnesota. "The Performance Equation." *The Performance Equation*, 2014, www.d.umn.edu/~gshute/arch/performance-equation.xhtml.

The area of interest of this essay is how a more efficient approach to software could reduce the number of required *instructions* and the required *cycles per instruction*. In particular, how we might optimize the compiling of the code rather than the design of the code itself? This is an overlooked topic - optimization is a well-established topic in Operations Research and Algorithmic Computer Science, but is rarely discussed in Computer Architecture.

I compare the performance of a Complier against two types of Virtual Machine (one stack-based, the other register-based) to see which approach performs best on tasks with varying degrees of complexity.

2 Structuring My Experiment

2.1 Bytecode and Machine Language

The first thing to note is that computers do not inherently "understand" coding languages. Any coding language such as C or Java must first be converted into Machine Language.

The method by which a coding language is converted into an executable machine language depends on how high a level of language it is[14]. For example, Java is considered a high-level language and C is considered a mid-level language because Java contains feature such as garbage collection and automatic bounds checking that C lacks.

The high-level languages compile differently from their simpler counterparts. As **Figure 3** below illustrates, Java source code is translated into Java bytecode by a Java Compiler in order to make it readable by the Java Virtual Machine (an emulation of a computer system). The Java Virtual Machine (JVM) then renders this code into the machine code required by the computer to execute the desired function[15]:

[14] Etiemble, Daniel. "45-Year CPU Evolution: One Law and Two Equations." *University Paris Sud*, 2013, p. 6.
[15] Carroll, Jeff. "How Java Works." *How Java Works*, CMU, 2009, www.cs.cmu.edu/~jcarroll/15-100-s05/supps/basics/history.html.

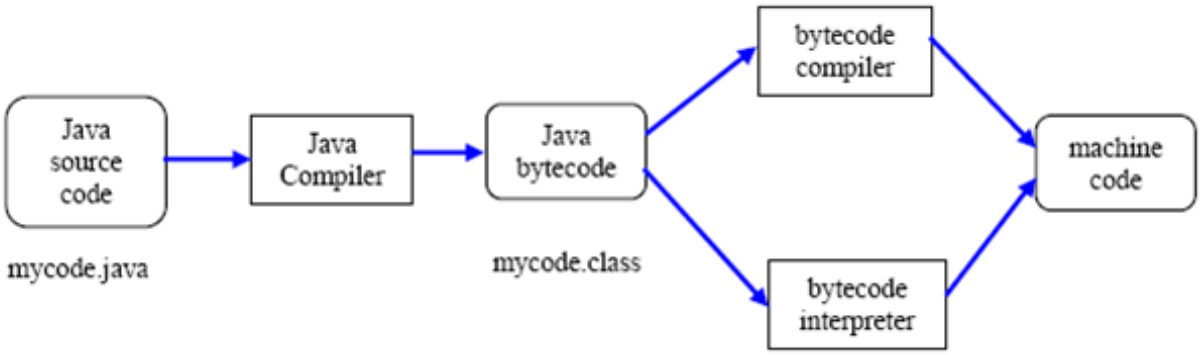

Figure 3: Conversion of Java Code into Machine Code[16]

C follows a similar process. The source code is initially compiled to assembly (similar to bytecode) and this is then interpreted into Machine code does. **Figure 4** demonstrates the process of how C/C++ code is compiled:

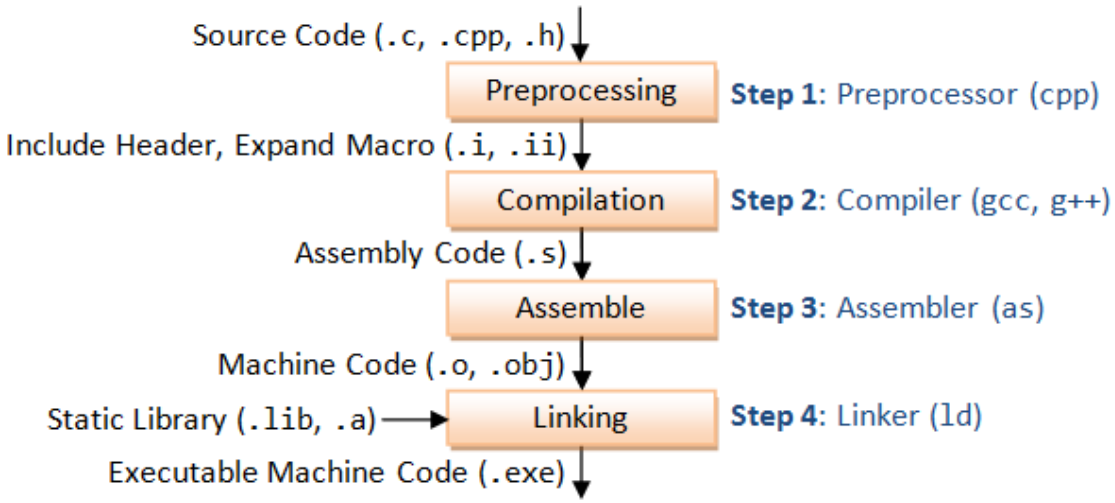

Figure 4: Conversion of C++/C Code into Machine Code[17]

Note that the key difference between the two processes is that Java requires both a Complier and a Virtual Machine to translate the source code into executable machine code. C requires just a Complier and an Assembler.

[16] https://i.stack.imgur.com/FARhm.png
[17] https://www3.ntu.edu.sg/home/ehchua/programming/cpp/images/GCC_CompilationProcess.png

2.2 *Stack and Register Virtual Machines*

There are actually two types of Virtual Machines. A Java Virtual Machine is a popular example of what is known as a "stack-based" virtual machine[18] (so called because it organizes the data in stacks). The other type of virtual machine is "register-based" (so called because it keeps the data in a single registry). A popular example of a register-based virtual machine is Lua 5.0[19].

Because of the way that these two virtual machines arrange the data, they operate differently and can have very different processing times for the same type of task. A stack machine assumes the location of operands by taking the two most recent values on the stack and appending the result to the same stack[20]. It also stores and loads separately which results in the CPU having to run twice as many instructions as a register-based machine. **Figure 5** demonstrates the syntax of Conception (our stack-based virtual machine):

Sample High-Level Code	Stack Machine (Conception)
int a,b,c; a = a+b; a = a*c	Iconst a; Iconst b; Iconst c; ladd; imul;

Figure 5: Stack Syntax - Operations need no implicit reference of operands

A register-based virtual machine requires the specific location for the operands and the result. This means that while there will be less instructions in comparison to a stack machine, but each instruction will take up more cycle time because it is more complex. **Figure 6** demonstrates the syntax of Inertia (our register-based virtual machine):

Sample High-Level Code	Register Machine (Inertia)
int a,b,c; a = a+b; a = a*c	set t1, t2 set t3, t4 set t5, t6 add t1, t5, t2 mul t1, t5. t2

Figure 6: Register Syntax - Operations require location for operands and results

[18] Javapoint. "JVM | Java Virtual Machine - Javatpoint." *Www.javatpoint.com*, 2012, www.javatpoint.com/internal-details-of-jvm.
[19] Lua. "Lua 5.0 Documentation ." 2016, www.lua.org/doc/jucs05.pdf.
[20] Leyse, Carl F. "Preliminary Investigations of Stack Height." 1950, doi:10.2172/12467735.

2.3 *Turing Equivalent Machines and Timing Specifics*

In order to ensure a fair comparison between the Complier and Conception and Inertia (our two Virtual Machines), we need to define the coding environment.

Specifically, we want to ensure that the environment in which we run our algorithms is the same in terms of processing power and time complexity. To do so, we mathematically model them as Turing Equivalent Machines (named after Alan Turing's discovery of how to use math to run programs of any complexity[21]). By using Turing Equivalent Machines with the same processing bandwidth[22] and similar time complexities, we can be certain that differences in runtime are solely due to differences in the method of compilation.

We also want to measure performance in terms of CPU time (the number of clock ticks the processor takes to run the operation) rather than elapsed time (because the latter can be affected by factors such as I/O or multitasking delays[23]). To observe CPU time, we use the C library's built-in clock() method and the clocks_per_second from time.h. **Figure 7** demonstrates sample C code to measure time of function run():

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

int main () {
   start_t, end_t, total_t;
   start_t = clock();
   run();
   end_t = clock();
   total_t = (long)(end_t-start_t)*1000/CLOCKS_PER_SEC;
   printf("Total time taken by CPU: %ld\n", total_t  );
   printf("Exiting of the program...\n");
   return(0);
}
```

Figure 7: Sample C code to measure the time in milliseconds that the the CPU takes to process the run() function

[21] Hodges, Andrew (2012). *Alan Turing: The Enigma* (The Centenary Edition). Princeton University Press. ISBN 978-0-691-15564-7.
[22] Smith, Carl H. "Introduction." *A Recursive Introduction to the Theory of Computation*, 1994, pp. 1–2., doi:10.1007/978-1-4419-8501-9_1.
[23] "CPU Time Accounting." *IBM DeveloperWorks : Linux : Linux on Z and LinuxONE : Tuning Hints & Tips : CPU Time Accounting*, 3 Nov. 2017, www.ibm.com/developerworks/linux/linux390/perf/tuning_cputimes.html.

2.4 *Queries and Environments*

The final requirement is to develop test cases that differ both in number of instructions and cycles/instructions in order to assess the performance of our three environments.

The queries that I have chosen to run consist of 3 functions, each with 4 levels of complexity. Each test is repeated 5 times to ensure accuracy among the results.

Function 1 is to find the first 1,000 Fibonacci numbers.
Function 2 is to perform some high value addition.
Function 3 is a function that calls itself numerous times.

The syntax for these three functions is as follows:

FibFun

A function that finds the first (10^2, 10^3, 10^4, 10^5) Fibonacci Numbers

Sample High-Level Code (C)	Register Machine (Inertia)	Stack Machine (Conception)

void FibFun (int val[],int curr, int max){
  if (curr != max){
  val[curr] = val[curr-1] + val[curr-2]
  FibFun(val, curr +1, max)
  }
}
int main(){
  int fibresults[1001];
  int fibresults[0] = 1;
  int fibresults[1] = 1;
 	
  FibFun(fibresults, 2, 1000);
 	
}
	load R1 #0
	load R2 #500
	load @0 #0
	load @1 #1
	1:
	itn R0 R1 R2
	if R0 P2
	add @0 @0 @1
	add @1 @0 @1
	inc R1
	goto P1
	return
		iconst 0
		gstore
		iconst 1
		gstore
		iconst 0
		dup
		iconst 1000
		swap
		ilt
		if_icmple 20
		inc
		gload
		gload
		swap
		iadd
		gstore 5
		gload
		ret

Figure 8: Function for first 1000 Fibonacci numbers with similar time complexity, but different methods of memory allocation (which simulate compiling)

Addition:

A function that adds up to (10^8,10^7… 10^5)

Sample High-Level Code (C)	Register Machine (Inertia)	Stack Machine (Conception)

int main(){
  long val = 0;
  for (long i = 0; i < 100000000; i++){
  val++;
  }
}
	load R1 #0
	load R2 #100000000
	1:
	itn R0 R1 R2
	if R0 P2
	inc R1
	goto P1
	2:
	return
		iconst 0
		dup
		iconst 100000000
		igt
		if_imcple 7
		inc
		goto 1
		ret

Figure 9: Function that increments itself from 0 to 10^8

Recursion:

A function that calls itself (10^5, 10^5… 10^3) times

Sample High-Level Code (C)	Register Machine (Inertia)	Stack Machine (Conception)

void recursion (long rec){
  if (rec != 0){
  recursion(rec-1);
  }
}
int main(){
 	
  recursion(1000);
 	
}
	load R1 #0
	load R2 #1000
	call P1
	return
	1:
	2:
	itn R0 R1 R2
	if R0 P2
	dec R2
	call P2
	3:
	return
		iconst 1000
		gstore
		call rec
		ret
		procedure rec
		gload
		dup
		iconst 0
		ieq
		if_icmple 6
		ter
		dec
		gstore
		call rec
		ret

Figure 10: Function to recursively call itself 1000 times, each implementation has a similar time

These functions are run and compiled on each of our three machines - Conception (our stack-based virtual machine); Inertia (our register-based virtual machine); and a GNU Compiler (our C compiler that builds directly to assembly). Each function is run at four different levels of complexity (e.g. 10^2, 10^3, 10^4, 10^5 for FibFun). We compare the average run time across five observations for each task in each environment to determine which approach minimizes the processing time.

3 Experiment & Analysis

3.1 Processing Time Between Functions

Since all of our experiments are being run on the same computer, we can assume that each of the functions will have the same Cycle Speed (the hardware). So the differences in CPU Processing time will be purely a function of the two software variables:

$$\text{CPU Time (Processing Time)} = \text{Instructions} * \text{Cycles/Instructions} * \text{X (Constant)}$$

Since the batches vary in terms of the number of instructions, we can say:

$$\text{CPU Time (Processing Time)} = \text{Batch size} * \text{K (Constant)} * \text{Cycles/Instructions} * \text{X (Constant)}$$

This means that the slope of the line formed by the four observations for each function at each level of complexity will show how time increases as a function of the complexity of the task.

The slope is the change in processing time over change in batch size. Therefore, by dividing both sides by Batch size we isolate the slope as:

$$\frac{\text{Processing Time}}{Batch\ Size}\ (\text{slope}) = \ \text{Cycles/Instructions} * \text{XK (Constant)}$$

The steepness of the slope of the line for each function will reveal the relative complexity of that function.

Figure 11 below shows the processing time that Conception required for each of the three functions (Addition, FibFun, and Recursion) at each of the four levels of complexity.

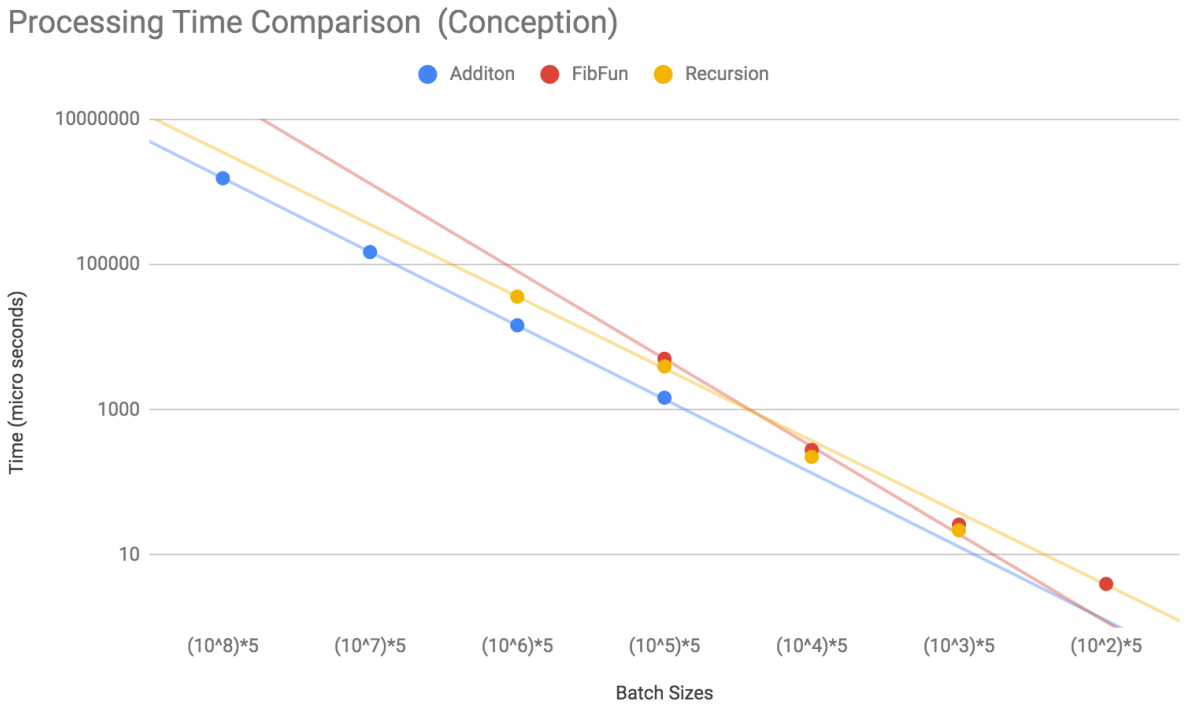

Figure 11: Graph demonstrating the processing time for each batch level of the three functions for the Conception virtual machine.

We can see that FibFun has the highest slope and is therefore the most complex of the three functions (note that we have excluded the result for the 10^2 version of FibFun (the red dot in the bottom right) because the processing time was so small $2*10^{-6}$ that the computer rounded the result, producing a value that was well off the trend line created by the other three FibFun data points).

This result seems intuitively reasonable given the amount of code FibFun has in comparison to Addition (the function with the lowest slope).

The reason why the observation of complexity is important as it allows us to determine which approach (Compiler, Stack-based VM, Register-based VM) will be best for tasks with varying levels of complexity.

3.2 Processing Time Between Methods for the Addition Function

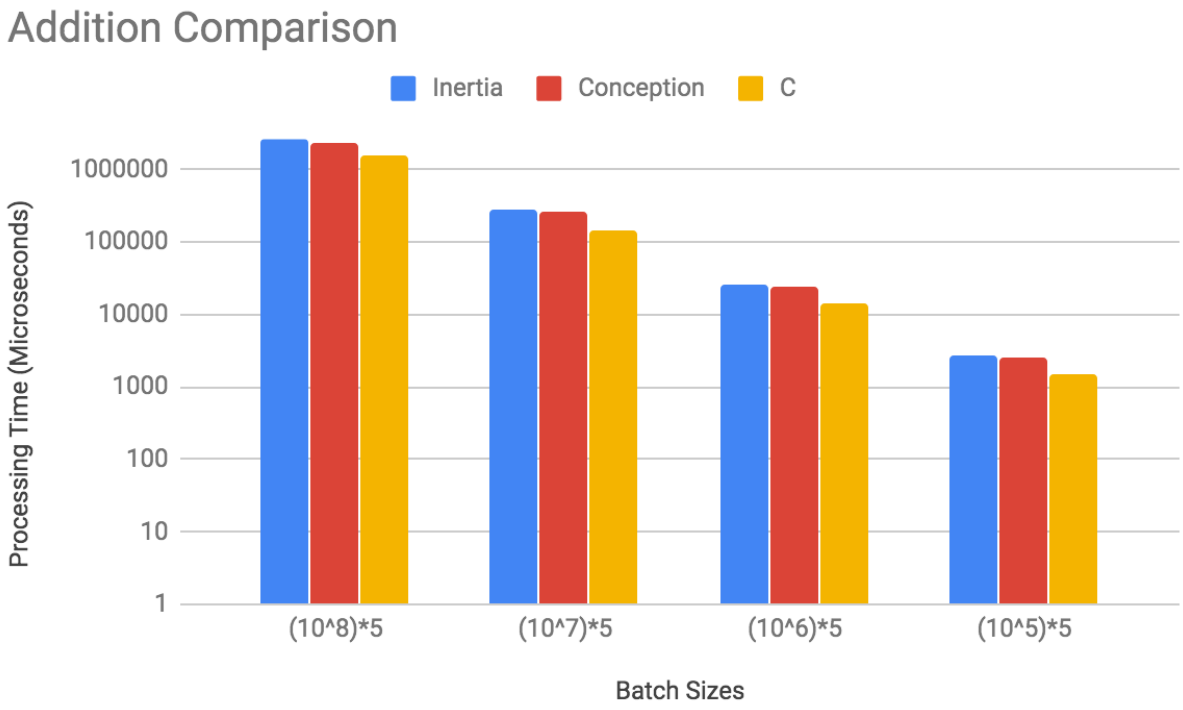

Figure 12: Graph demonstrating distribution of processing time between the 3 different methods for the Addition function.

Figure 12 shows the processing time distribution for a function that is not complex. These results reveal that Inertia is slower than Conception by 10%, and slower than the C + GNU compiler method by 50%. The Graph above uses a logarithmic scale such that it can show all the results of the batch sizes, but **Figure 13** the magnitude of the differences in processing times to be seen more clearly:

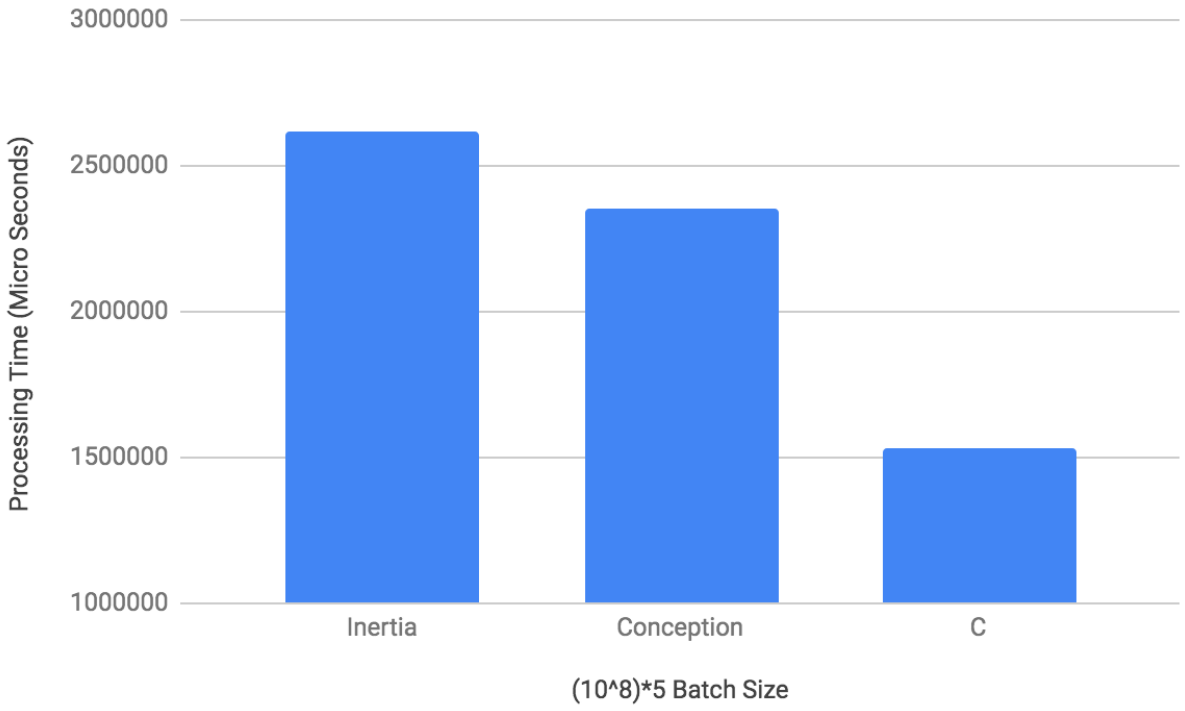

Figure 13: Processing Time for a Batch size of (10^8)*5 - FibFun

From this we can conclude that if a machine needs to do a lot of simple queries it is best to use the C and GCU compiler. However, if the instructions are being provided in a high level language that requires the use of a virtual machine, a stack-based virtual machine will perform better than a register-based virtual machine.

3.3 *Processing Time between Methods for the Recursion Function*

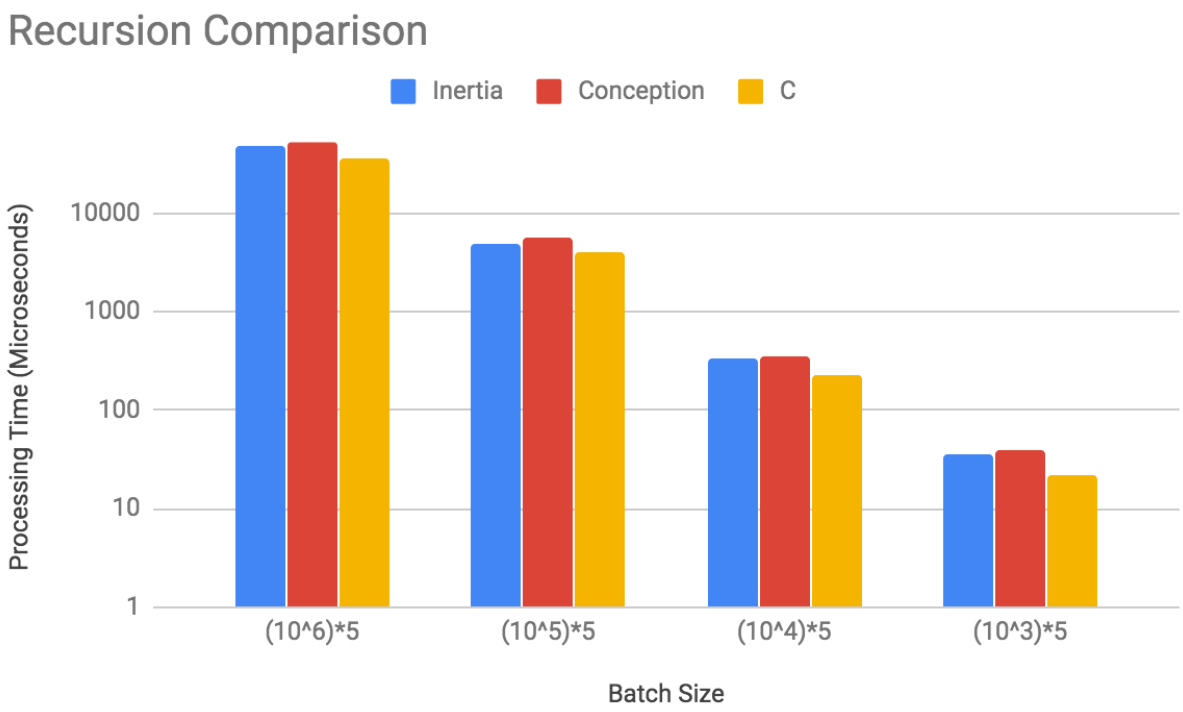

Figure 14; Graph demonstrating distribution of processing time between the 3 different methods for the Recursion function.

Figure 14 represents the processing speed for the Recursion function between the three different methods. In contrast to Figure 12 Conception preforms worse than Inertia by a factor of 7.9%. However, our C implementation is still faster by 30% in comparison to Inertia. Again, this graph also uses a log scale such that the processing time for the smaller batches can be shown on the same axis as the processing time for the bigger batches.

From this, we can conclude that if we want to optimize processing time we should convert to a lower level language and then just interpret into machine code (what our C code does), but if that's not an option we should use a register-based virtual machine for semi-complex operations.

3.4 *Processing Time between Methods for the FibFun Function*

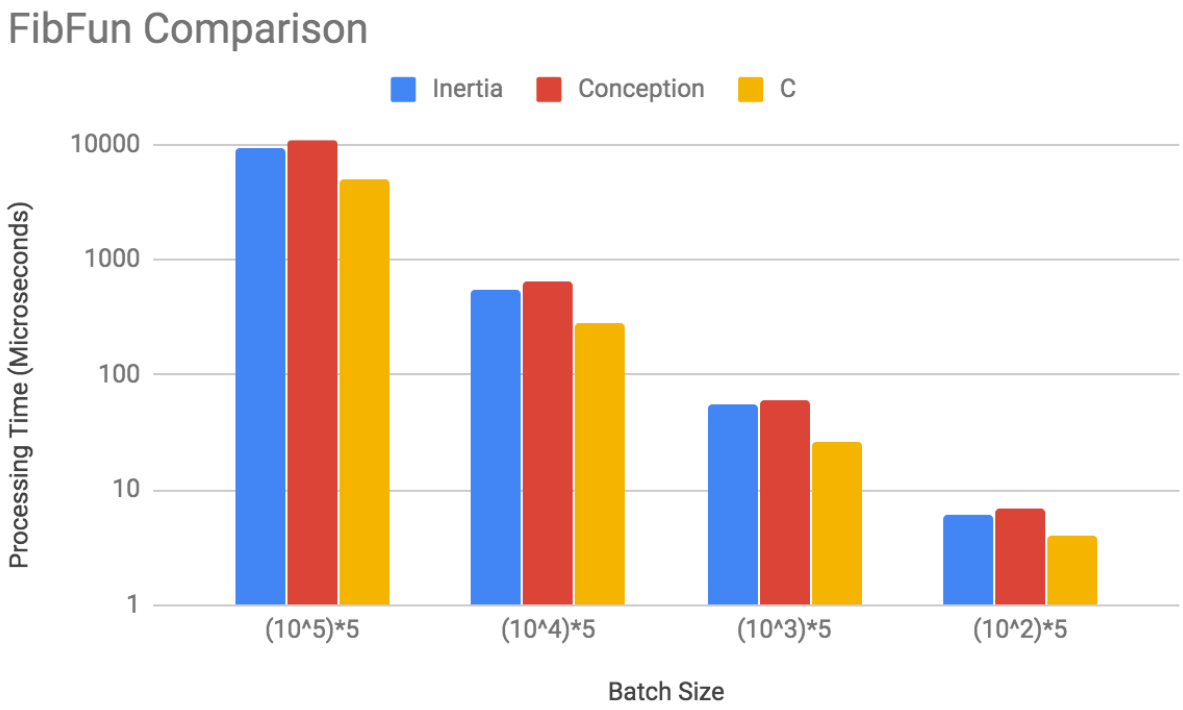

Figure 15; Graph demonstrating distribution of processing time between the 3 different methods for the FibFun function.

As established previously (see page 15), FibFun is the most complex of our three functions. Even so, the C method remains the fastest, followed by Inertia and then Conception. The margin by which Inertia outperforms Conception has now increased to 15% (versus 8% for the Recursion function). This illustrates the point that as the task becomes more and more complex, a register-based machine will increasingly outperform a stack-based machine.

However, our C interpretation is still faster than either virtual machine. The conclusion is that, until we are dealing with a function that is considerably more complex than just finding Fibonacci numbers, interpreting to machine code remains the fastest method for processing code.

4 Conclusion

4.1 Limitations of the Experiment

As stated in Section 2 above, my goal was to create an environment in which I had standardized everything in the experiment other than the method of data compilation. That way, I could be sure that any differences in the time taken to run the functions was solely caused by how the three environments approached the tasks.

Unfortunately, this was easier said than done. While it is reassuring that I observed a constant slope of the lines based on the different batches, I got less consistent results when I repeated the experiment for the same batch size multiple times.

Figure 16 below shows the fluctuations in the results I achieved for the Inertia virtual machine when running the addition function at 10^8*5 batches:

Figure 16; a demonstration of fluctuations in results for one of the tests

The results from these five tests varied by up to 10%. This may not be sufficient to call into question our key conclusions (processing time increases linearly as a function of complexity; and compiling code straight to assembly is the optimal choice), but it is still crucial to explain how this variability occurs.

When running all of the tests I made sure only to have open the CLion application. However, it is possible that the computer was also running other background tasks that I was not aware of (especially since it is a computer that wasn't made to run only one task at a time).

A second issue may be that the compiler varies in the time taken to read libraries or to output and this would add a further source of random error.

If I wanted to eliminate both of these sources of error, I would need to create a large set of Turing Equivalent machines that would only run the desired functions. This is something far outside the scope of this paper and my budget!

4.2 Result's Implications for Modern Technology

As mentioned at the start of this paper, my interest is in how we can improve the performance of existing hardware now that we are nearing the point at which greater density of transistors may be hard to achieve.

My focus was on the optimal method for translating coding languages into machine readable code. What my experiments have shown is that the best way to compile is to code in a lower level language and then convert to assembly. Doing so means that you can compile processes an average of 40% faster than when you use a virtual machine (whether stack-based or register-based).

The implication of this is that we should be focusing on creating applications using lower level languages. However, the TIOBE index shows that Java is the most used coding language and other higher-level languages such as Python will overtake C in the next 4 years[24].

There are a couple of reasons why I think this may be the case. First is that we have assumed that computing power is going to continue to get cheaper so we do not need to optimize our use of it. Second, there is the cost of the programmers. It may be cheaper for the computer to process lower-level languages, but if these languages reduce the speed at which the programmers can work, then the overall costs may be higher.

For a variety of reasons, we don't always have the option to code in lower level languages and compile straight into assembly. Given this we must use virtual machines. My FibFun function was more complex than the Addition and Recursion functions but it was not sufficiently complex to illustrate the degree to which register-based virtual machines have been shown to

[24] "Latest News." *TIOBE*, 2019, www.tiobe.com/tiobe-index/.

outperform stack-based virtual machines. It is interesting that LUA 5.0 is the only major example that compiles using a register based virtual machine[25]. Java Virtual Machine, Python and Ruby all use stack-based compilers.

This raises the prospect of some significant disruption in the industry. As interest grows in the optimization of software, we should expect register-based virtual machines to capture a major share of the market.

4.3 Future Research Paths

Since we can optimize the processing capacity of our existing hardware by coding in lower level languages that do not requires virtual machines, we should conduct research into the costs of implementing such a solution.

As mentioned above, our use of higher-level languages may reflect our desire to reduce the time taken by programmers to create code. If the cost of processing power was to start to rise relative to the cost of programming, we may need to rethink how we optimize the relationship between these two sources of cost. For this, we should measure how much longer it takes to learn/code in a lower level language.

The same rationale applies to how we think about register-based virtual machines - while they compile faster than stack-based virtual machines, they are often much more complex to use[26] and this complexity may outweigh the value of their performance advantage.

Another important experiment should analyze the memory usage of the three environments studied. While it is true that a register-based virtual machine is less intensive in terms of CPU load, it has a higher memory requirement than its stack-based counterpart. If memory becomes a constraint, then stack-based virtual machines may be the optimal solution.

[25] "Lua 5.3 Bytecode Reference¶." *Lua 5.3 Bytecode Reference - Ravi Programming Language 0.1 Documentation*, the-ravi-programming-language.readthedocs.io/en/latest/lua_bytecode_reference.html.
[26] Ibid.

5 Bibliography

5.1 Works Cited

Kelion, Leo. “Moore's Law: Beyond the First Law of Computing.” *BBC News*, BBC, 17 Apr. 2015, www.bbc.com/news/technology-32335003.

Moore, G.e. “Cramming More Components Onto Integrated Circuits.” *Proceedings of the IEEE*, vol. 86, no. 1, 19 Apr. 1965, pp. 82–85., doi:10.1109/jproc.1998.658762.

Rupp, Karl. *Github*, 2 Apr. 2016, github.com/karlrupp/microprocessor-trend-data.

Wirth, N. “A Plea for Lean Software.” *Computer*, vol. 28, no. 2, 3 Feb. 1995, pp. 64–68., doi:10.1109/2.348001.

Kumar, Suhas. *Fundamental Limits to Moore's Law*. Stanford, 2015, p. 3, *Fundamental Limits to Moore's Law*.

Courtland, Rachel. “Gordon Moore: The Man Whose Name Means Progress.” *IEEE Spectrum: Technology, Engineering, and Science News*, IEEE Spectrum, 30 Mar. 2015, spectrum.ieee.org/computing/hardware/gordon-moore-the-man-whose-name-means-progress.

Blizzard. “About Starcraft.” 5 Nov. 2017, starcraft.com/en-us/.

Blizzard. “StarCraft II System Requirements.” *Blizzard Support*, 2016, us.battle.net/support/en/article/27575.

Duluth, Minnesota. “The Performance Equation.” *The Performance Equation*, 2014, www.d.umn.edu/~gshute/arch/performance-equation.xhtml.

Etiemble, Daniel. “45-Year CPU Evolution: One Law and Two Equations.” *University Paris Sud*, 2013, p. 6.

Carroll, Jeff. “How Java Works.” *How Java Works*, CMU, 2009, www.cs.cmu.edu/~jcarroll/15-100-s05/supps/basics/history.html.

Lua. “LUA 5.0 Documentation .” 2016, www.lua.org/doc/jucs05.pdf.

Leyse, Carl F. “Preliminary Investigations of Stack Height.” 1950, doi:10.2172/12467735.

Smith, Carl H. “Introduction.” *A Recursive Introduction to the Theory of Computation*, 1994, pp. 1–2., doi:10.1007/978-1-4419-8501-9_1.

“CPU Time Accounting.” *IBM DeveloperWorks : Linux : Linux on Z and LinuxONE : Tuning Hints & Tips : CPU Time Accounting*, 3 Nov. 2017, www.ibm.com/developerworks/linux/linux390/perf/tuning_cputimes.html.

“Latest News.” *TIOBE*, 2019, www.tiobe.com/tiobe-index/.

“Lua 5.3 Bytecode Reference¶.” *Lua 5.3 Bytecode Reference - Ravi Programming Language 0.1 Documentation*, the-ravi-programming-language.readthedocs.io/en/latest/lua_bytecode_reference.html.

5.2 Images Used

https://i.stack.imgur.com/FARhm.png

https://www3.ntu.edu.sg/home/ehchua/programming/cpp/images/GCC_CompilationProcess.png

Appendix A

Conceptum is primarily the work of Ruijie Fang. Conceptum's source code is opensourced under the GNU General Public License v3.0. The complete source code for the Conceptum virtual machine can be found at the URL: https://github.com/Conceptual-Inertia/Conceptum.

Inertia is mainly the work of Siqi Liu. Inertia's source code is also opensourced under the GNU General Public License v3.0. The complete source code for the Inertia virtual machine can be found at the URL: https://github.com/Conceptual-Inertia/Inertia.

All of the test codes presented in this paper can be accessed online via the following URLs: https://github.com/Conceptual-Inertia/testcodes, and https://github.com/Conceptual-Inertia/conceptum_testcodes.

Appendix B

```
#include <iostream>
#include <time.h>
void recursion(long rec){
   if (rec != 0){
      recursion(rec-1);
   }
}
void fibfun (int val[], int curr, int max){
   if (curr != max) {
      val[curr] = val[curr - 1] + val[curr - 2];
      fibfun(val, curr + 1, max);
   }
}

int main() {
   // ---------- //
   clock_t start_time = clock();
   for (int i = 0; i < 50; i++) {
      recursion(100000);
```

```
}
clock_t exe_time;
exe_time = clock()-start_time;
printf("Batch 1 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
// ---------- //
start_time = clock();
for (int i = 0; i < 5; i++) {
   recursion(100000);
}
exe_time = clock()-start_time;
printf("Batch 2 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
// ---------- //
start_time = clock();
for (int i = 0; i < 5; i++) {
   recursion(10000);
}
exe_time = clock()-start_time;
printf("Batch 3 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
// ---------- //
start_time = clock();
for (int i = 0; i < 5; i++) {
   recursion(1000);
}
exe_time = clock()-start_time;
printf("Batch 4 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
printf("END OF RECURSIVE BATCHs \n");
// Def of the fib sequence
int fib[100001];
fib[0] = 1;
fib[1] = 2;
start_time = clock();
for (int i = 0; i < 5; i++){
   fibfun(fib, 2, 100000);
}
exe_time = clock()-start_time;
printf("Batch 1 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
// ---------- //
start_time = clock();
```

```
for (int i = 0; i < 5; i++){
   fibfun(fib, 2, 10000);
}
exe_time = clock()-start_time;
printf("Batch 2 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
// ---------- //
start_time = clock();
for (int i = 0; i < 5; i++){
   fibfun(fib, 2, 1000);
}
exe_time = clock()-start_time;
printf("Batch 3 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
// ---------- //
start_time = clock();
for (int i = 0; i < 5; i++){
   fibfun(fib, 2, 100);
}
exe_time = clock()-start_time;
printf("Batch 4 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
printf("END OF FIB BATCHs \n");

start_time = clock();
for (int x = 0; x < 5; x++) {
   long val = 0;
   for (long i = 0; i < 100000000; i++) {
      val++;
   }
}
exe_time = clock()-start_time;
printf("Batch 1 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
// ---------- //
start_time = clock();
for (int x = 0; x < 5; x++) {
   long val = 0;
   for (long i = 0; i < 10000000; i++) {
      val++;
   }
}
```

```
    exe_time = clock()-start_time;
    printf("Batch 2 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
    // ---------- //
    start_time = clock();
    for (int x = 0; x < 5; x++) {
        long val = 0;
        for (long i = 0; i < 1000000; i++) {
            val++;
        }
    }
    exe_time = clock()-start_time;
    printf("Batch 3 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);
    // ---------- //
    start_time = clock();
    for (int x = 0; x < 5; x++) {
        long val = 0;
        for (long i = 0; i < 100000; i++) {
            val++;
        }
    }
    exe_time = clock()-start_time;
    printf("Batch 4 executed in %lu \n", exe_time*1000000/CLOCKS_PER_SEC);


    return 0;
}
```