PROJECT REPORT

Topic: Writing a code for bench-marking of hardware (CPU) using the C programming language.

Course Code: CSE360

Course Title: Computer Architecture

Section: 03

Submitted By

Gazi Shahrear

ID: 2019-1-60-217

Azizul Islam Tushar

ID: 2019-1-60-213

Sifat Fatima Ferdous

ID: 2018-1-60-169

Submitted To

Dr. Md. Nawab Yousuf Ali

Professor

Department of Computer Science & Engineering.

Abstract:

Benchmarking is a frequently used approach in experimental computer science, particularly for evaluating tools and algorithms in comparison. As a result, a number of problems must be addressed in order to assure effective benchmarking, resource assessment, and result presentation, which is critical for academics, tool developers, and users, as well as tool competitions. We outline a set of requirements that are required for reliable benchmarking and resource assessment of automated solvers, verifiers, and similar tools' time and memory utilization, as well as the limits of existing methodologies and benchmarking tools. On Linux systems, the only way to meet these criteria in a benchmarking framework is to use the kernel's cgroup and namespace features.

Introduction:

A Benchmark is a point of reference by which something can be measured. Hardware benchmarks can be used to compare the performance of different hardware such as CPUs. CPU benchmark usually measures the performance and efficiency of a CPU by running both simple arithmetic and computationally heavy algorithms and measuring their time to compare the performance of different processors of different architectures and brands. Using thread specific benchmark, we can compare the single threaded performance and multi-threaded performance of different CPU.

State-of-the-art, high-performance microprocessors contain hundreds of millions of transistors and operate at frequencies close to 3-5 gigahertz (GHz). These processors are deeply pipelined, execute instructions in out-of-order, issue multiple instructions per cycle, employ significant amounts of speculation, and embrace large on-chip caches. These microprocessors are true marvels of engineering. Designing and evaluating these microprocessors are major challenges especially considering the fact that 1 second of program execution on these processors involves several billions of instructions, and analyzing 1 second of execution may involve dealing with hundreds of billions of pieces of information. The large number of potential designs and the constantly evolving nature of workloads have resulted in performance evaluation becoming an overwhelming task. Designing benchmarks deals with such task of accurate performance evaluations so that the architecture improvements and effect of clock speed change can be easily measured. Current day processors and task are also highly parallel and features multiple cores that can work simultaneously so designing multi-threaded benchmarks that takes advantage of every single core and reflect their performance improvement is also challenging.

Related Works:

"Reliable benchmarking: requirements & solution" this article was published on 3rd November,2017, was authorized by Dirk Beyer, Stefan Löwe & Philipp Wendler. In that article they explained some requirements for benchmarking. These are -

1) Measuring and limiting resources accurately is the most important requirement. The CPU time of all processes, including the time of all child processes a tool starts must be measured very accurately, and limited accordingly.

2) Terminating processes reliably is the next requirement. If a resource limit is violated it is necessary to reliably terminate the tool including all of its child processes. Or else, the child process will keep running and occupying CPU memory and resources.

3) Assigning cores deliberately is an important step for benchmarking. If a resource limit is violated, it is necessary to reliably terminate the tool including all of its child processes. Because if all threads are heavily accessing the memory at the same time, it could have a large impact on performance, and we need to avoid such performance influences as far as possible in order to achieve accurate measurements.

4) Respecting nonuniform memory access is another requirement to be careful about while benchmarking. Systems with several CPUs often have an architecture with nonuniform memory access. So, once a process has to access remote memory, this leads to a performance decrease depending on the load of the inter-CPU connection and the other CPU. Hence, a single tool execution should be bound to memory that is local to its assigned CPU cores, in order to avoid nondeterministic delays due to remote memory access.

5) To avoid swapping memory is another thing to be cautious about. It must be avoided during benchmarking, because it may degrade performance in a nondeterministic way. It can negatively affect the benchmarking, which is something that should not happen.

6) Isolating individual runs is the last requirement for benchmarking. If multiple processes or tools are executed at the same time, or even started sequentially but run at the same time, it can interfere with the benchmarking of the processes. Such interference could influence performance and change the results. The benchmarking environment should isolate the benchmarked processes to prevent such interference.

Problem Statements:

Writing a tool for bench-marking of hardware (CPU) using the C programming language.

Objective:

The objective of this project is to find and compare the single-threaded and multi-threaded performance of different CPUs from different generations having different clock frequency and micro-architectures.

Proposed Work:

This simple benchmarking C program consists of Integer arithmetic test,Floating point arithmetic tests and Prime number calculation test.

Algorithms used in the program:

1. Simple Arithmetic (Integer and Floating-point arithmetic operations)
    a) Integer: Calculates 100,000,000,000 instances of increment, modulus, multiplication, subtraction operations
    b) Float: Calculates 5,000,000,000 instances of division, multiplication, subtraction
2. Prime number calculation: Run for 2 to 10,000,000 instances

START

Step 1 -- * Take integer variable A

Step 2 -- * Divide the variable A with (2 to square root of A)

Step 3 -- * If A is divisible by any value (2 to sqrt(A)), then it is not Prime number.

Step 4 -- * Else it is prime

STOP

Implementation

This program is written in ANSI C using code::blocks and compiled with gee mingw- w64 compiler for 64-bit windows. It uses 'posix' threads (pthread) for creating threads for testing algorithms in single threaded and multi-threaded mode.

System Requirements:

CPU: 64-bit x86_64 microprocessor.

Compiler: Mingw-w64 (built in with code::blocks 20.03)Operating System: Windows 7/10 64-Bit

The code for thread creation using pthread:

```
61  void stbench(){
62      printf("\t\tRunning Benchmark...\n");
63      struct timeval start, end;
64      pthread_t thread[16]; // Create an array of threads to be created
65
66      gettimeofday(&start, NULL); // Get Thread Starting Time
67      pthread_create(&thread[0], NULL, benchInt, (void*)1); // Create a single thread
68      pthread_join(thread[0], NULL); // Wait for thread termination
69      gettimeofday(&end, NULL); // Get Thread Ending Time
70      // Print total thread time
71      printf("\t\tSingle Threaded Integer Arithmetic Time:\t %.6fs\n", timeMS(end,start));
72
```

The code for multiple thread creation using for loop:

```
139     gettimeofday(&start, NULL);
140     for(int i=0; i<8; i++)  // For loop for creating threads
141         pthread_create(&thread[i], NULL, benchInt, NULL);
142     for(int i=0; i<8; i++)  // For loop for terminating threads
143         pthread_join(thread[i], NULL);
144     gettimeofday(&end, NULL);
145     printf("\t\tMulti Threaded Integer Arithmetic Time(8t):\t %.6fs\n", timeMS(end,start));
146
```

The code for Integer arithmetic calculation:

```
15
16  void *benchInt(void* vargp)
17  {   if (vargp!=NULL)    // Lock thread to a single Core
18          SetThreadAffinityMask(GetCurrentThread(), GetCurrentProcessorNumber());
19      uint64_t i = 0;     // Declare 64 bit integer
20      for(i = 1; i < 100000000000; i++){
21          222%73*(i-9999);    // Integer operations: Increment, modulus, multiplication, subtraction
22      }
23      return NULL;
24  }
25
```

The code for Floating point arithmetic calculation:

```
26  void *benchFloat(void* vargp)
27  {
28      if (vargp!=NULL)
29          SetThreadAffinityMask(GetCurrentThread(), GetCurrentProcessorNumber());
30      double i;   // Declare 64 bit floating point number
31      for(i = 0.5; i < 5000000000; i++){
32          222/77.777777*(i-999);      // Float operations: Increment, division, multiplication, subtraction
33      }
34      return NULL;
35  }
```

The code for Prime number calculation:

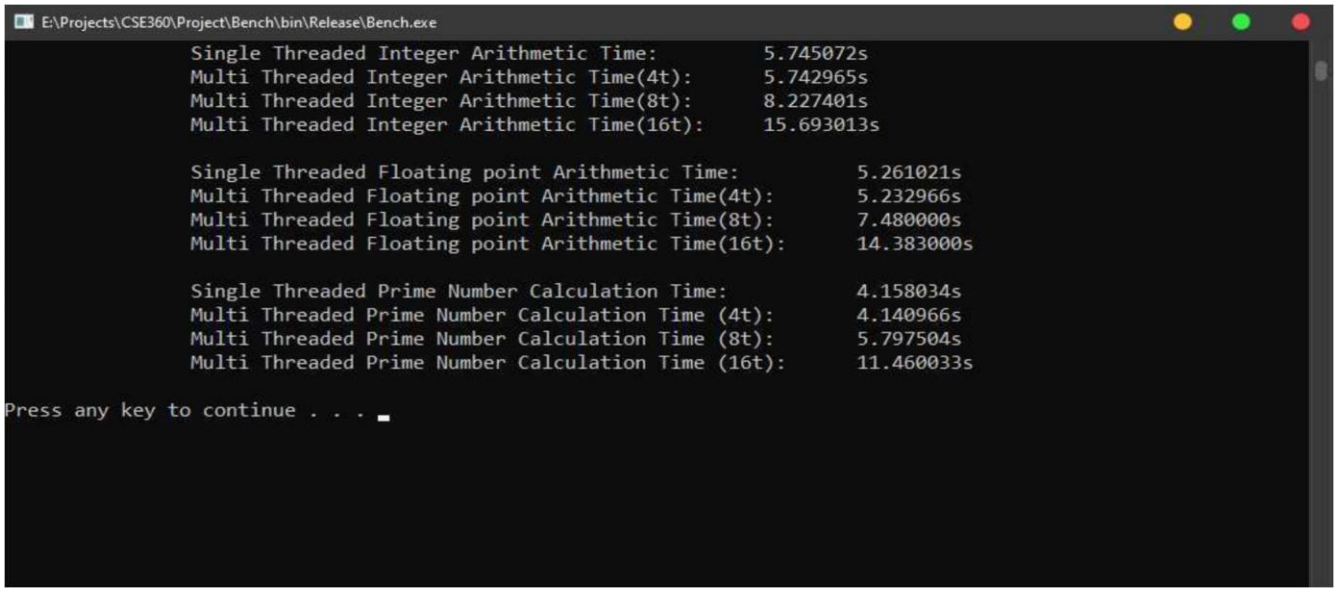
```
37  void *benchPrime(void *vargp){      // Prime number calculation algorithm
38      if (vargp!=NULL)
39          SetThreadAffinityMask(GetCurrentThread(), GetCurrentProcessorNumber());
40      int flag;
41      uint32_t i, j;      // Declare 64 bit integers for for loop
42      for(i=2; i<10000000; i++){  // All prime numbers upto 10000000
43          flag = 1;   // flag for deciding if prime or not
44          for(j = 2; j<sqrt(i)+1 ; j++){
45              if(i%j == 0){
46                  flag = 0;
47                  break;      // break if number is divisible
48              }
49          }
50      }
51      return NULL;
52  }
```

Function for calculating time from two times values:

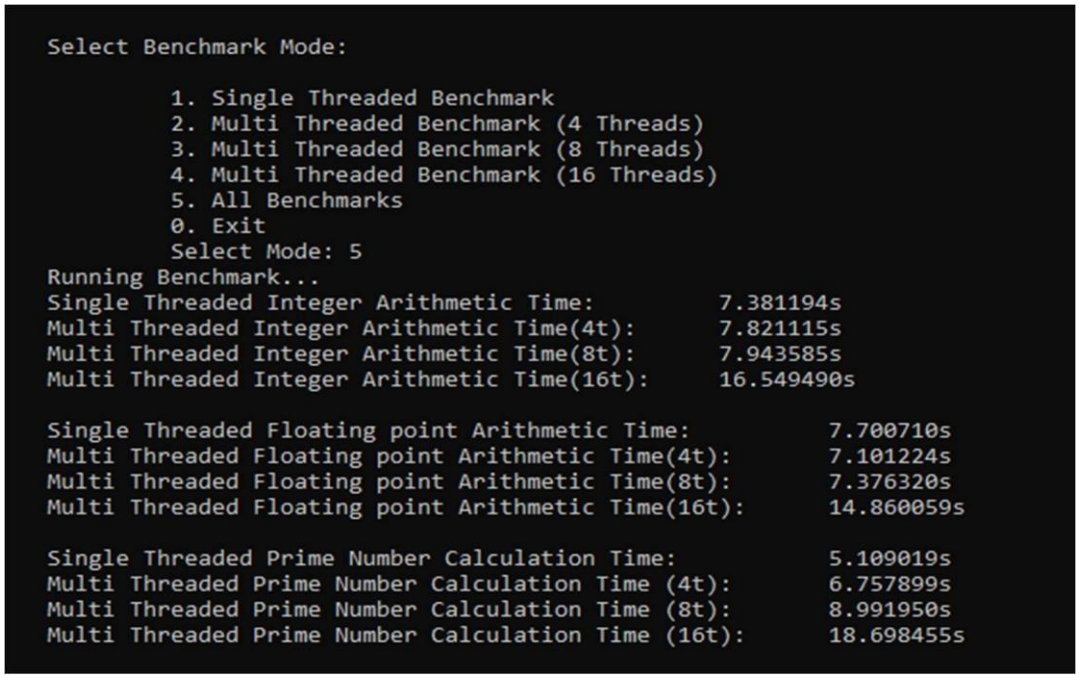
```
55  double timeMS(struct timeval  tv1, struct timeval tv2){
56      double s = (double)(tv1.tv_sec - tv2.tv_sec);   // Calulate time difference in seconds
57      double us = (tv1.tv_usec - tv2.tv_usec)/1000000.0; // Calclulate time difference in Micro-seconds
58      return (s+us);  // Add and return time
59  }
```

Test run:

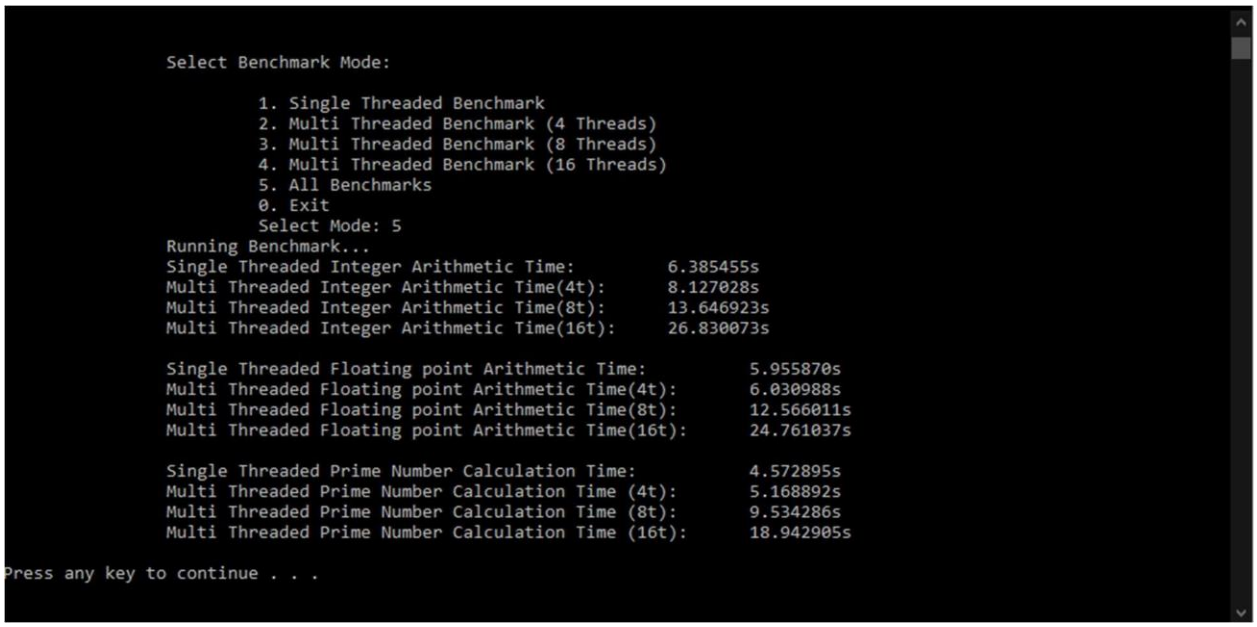
Test run on Intel(R) Core(TM) i5-9400F CPU @ 4.10GHz (6 Cores, 6 Threads)



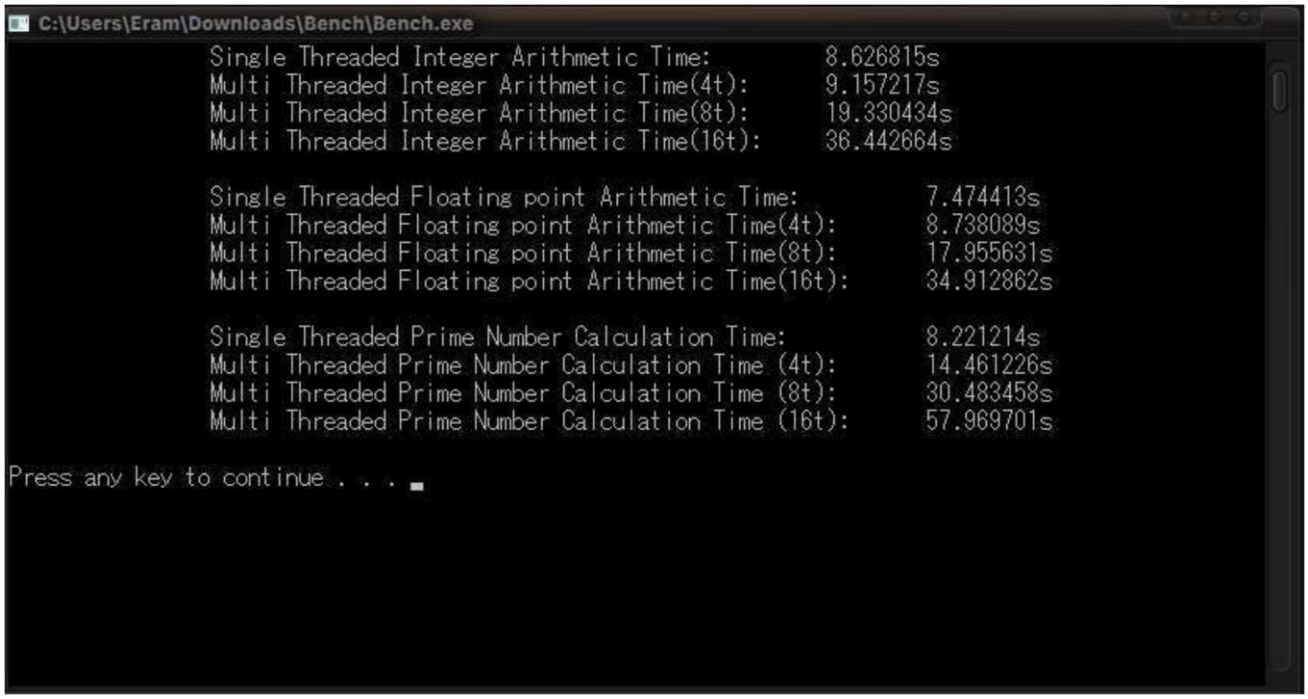
Test run on Intel(R) Core(TM) i5-8265U CPU @ 3.90GHz (4 Cores, 8 Threads)



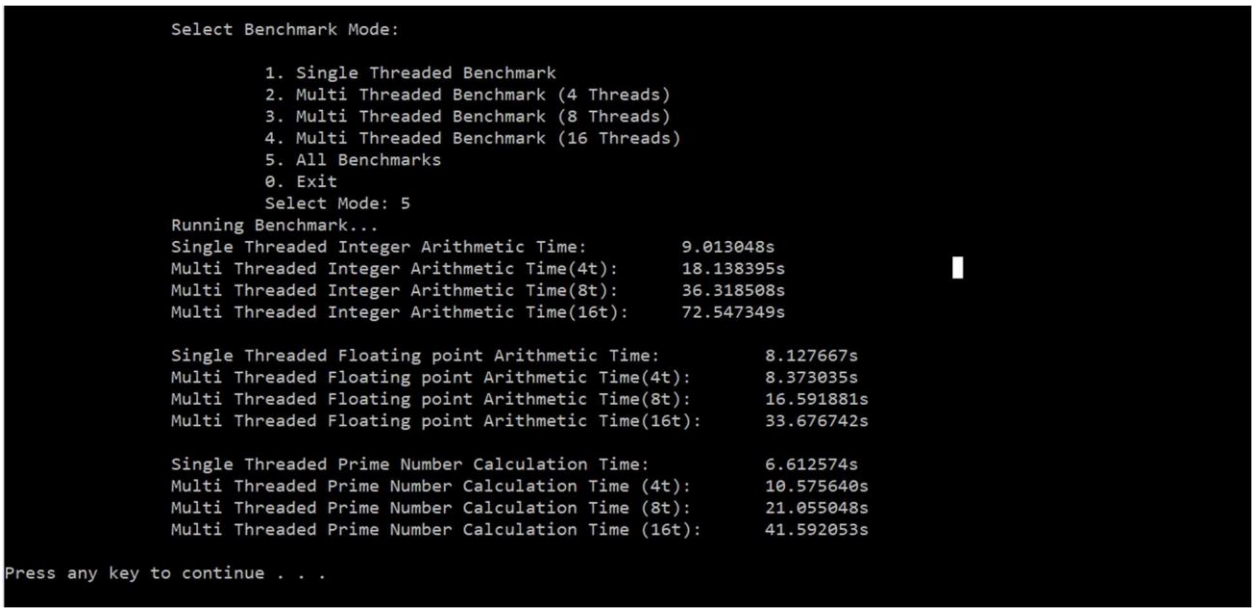
Test run on Intel(R) Core(TM) i3-8100 CPU @ 3.60GHz (4 Cores, 4 Threads)



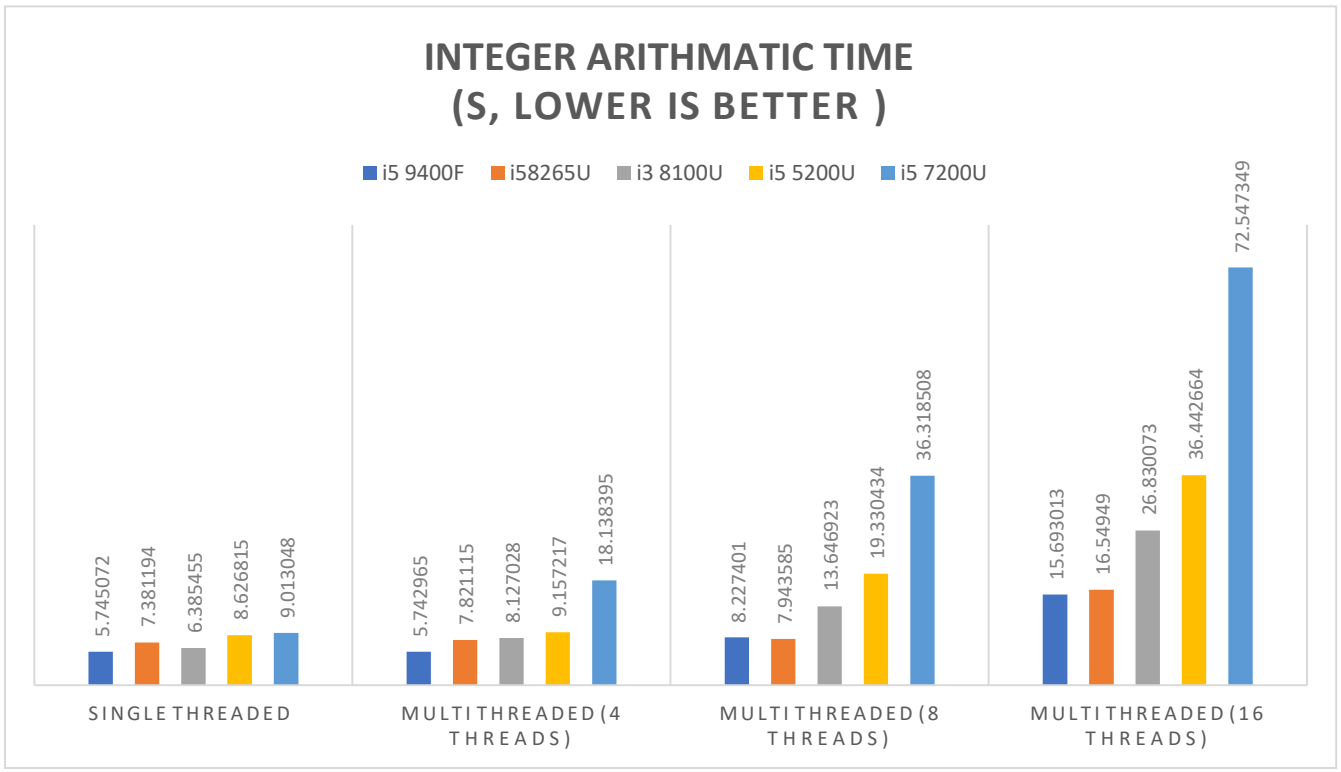
Test run on Intel(R) Core(TM) i5-5200UCPU @ 2.70GHz (2 Cores, 4 Threads)

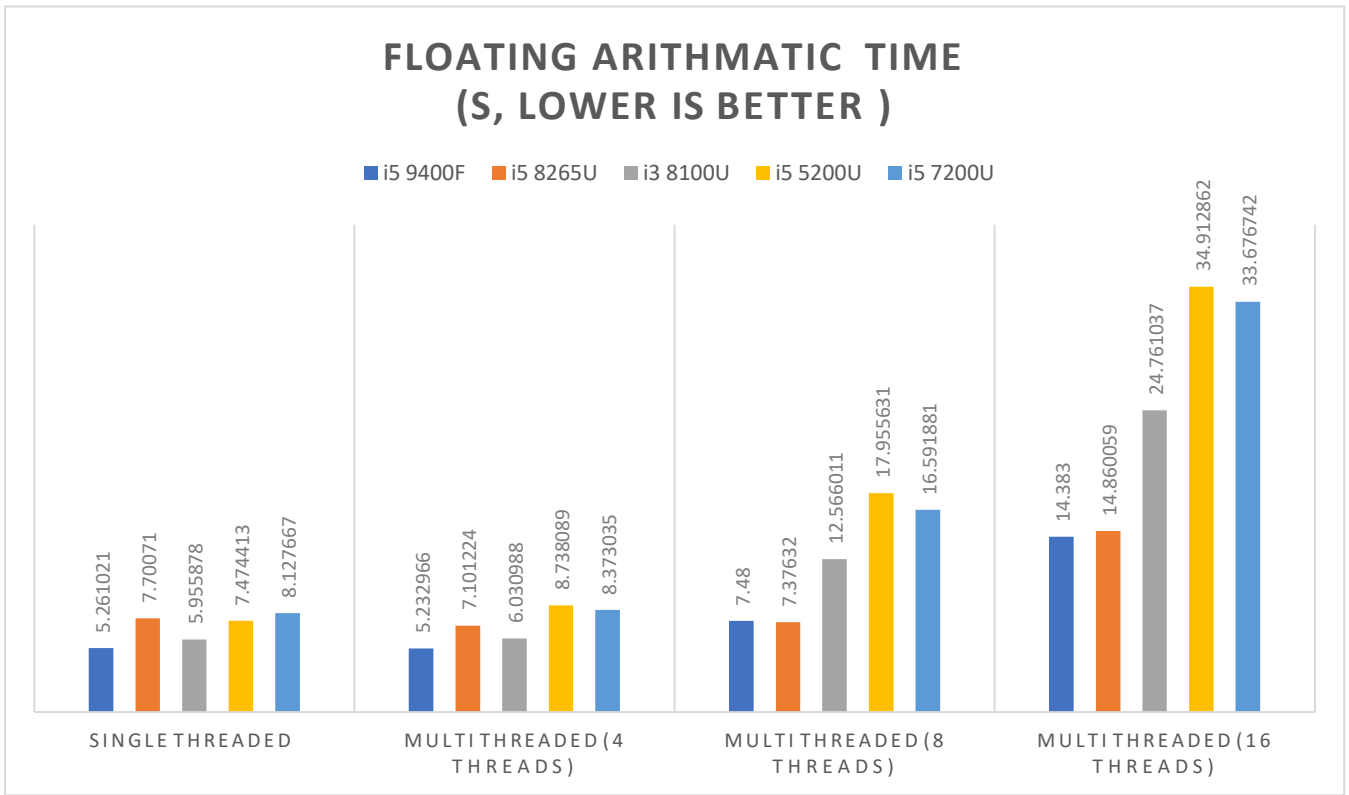


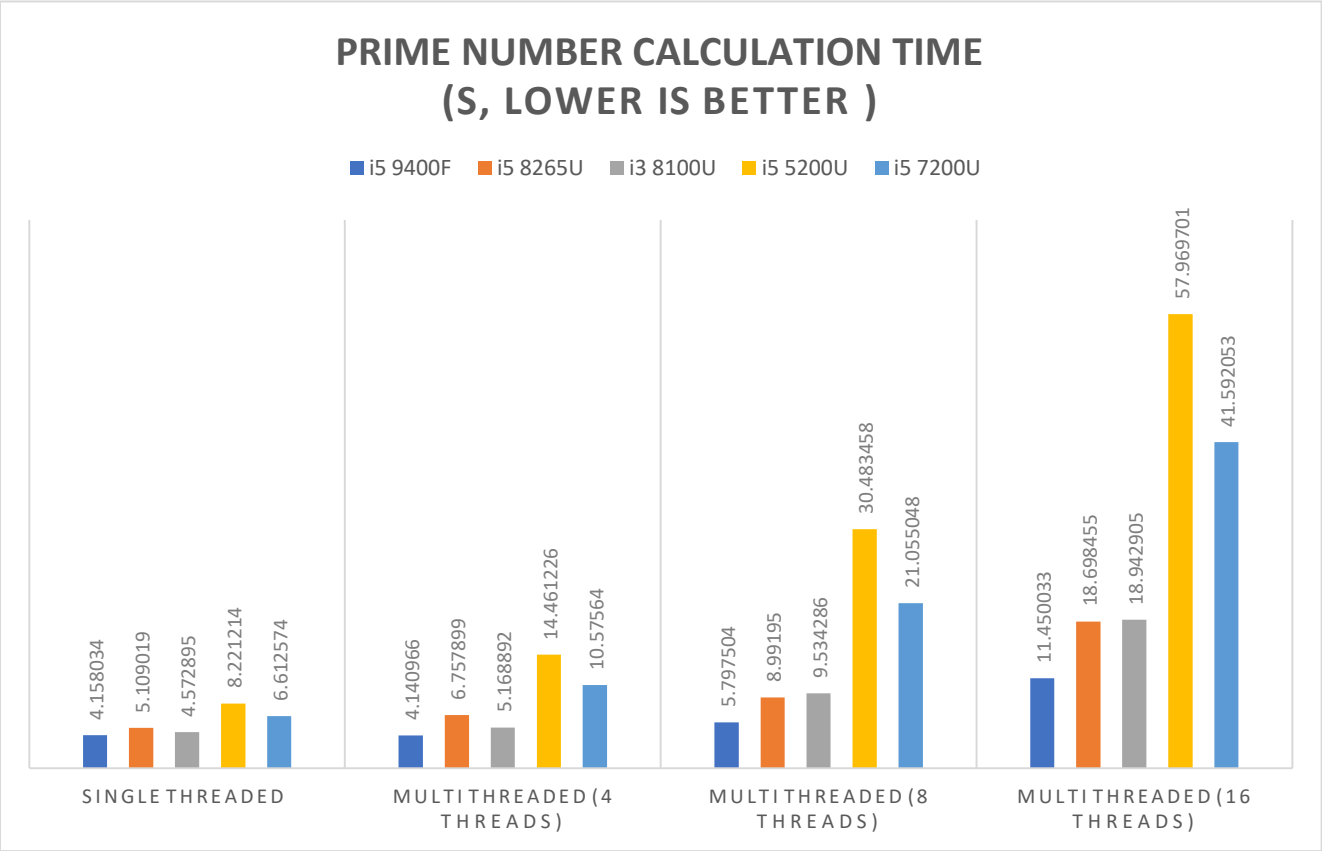
Test run on Intel(R) Core(TM) i5-7200UCPU @ 3.10GHz (2 Cores, 4 Threads)



Result Analysis:







From the tests runs in different CPUs of different micro architecture and frequency and comparing and their benchmark results it can be said that this benchmark successfully compares CPU hardware and their improvement. The integer and floating-point arithmetic test scales quite well with clock speed. The higher the clock speed of a processor the lower time it takes to complete any test.

The Core i5 9400F processor @4.1 GHz completes 100,000,000,000 64-bit integer modulus, multiplication and subtraction operations in just 5.745 seconds using just 1 thread. While the i5 8265U @3.9 GHz takes 7.38 seconds, i3 8100 @3.6 GHz takes 6.38 seconds, i5 5200U @2.7GHz takes 8.623 seconds and i5 7200U @3.1GHz takes
9.013 seconds. Similar results can be seen from the floating-point arithmetic test too.

But clock speed doesn't decide everything, the generational architecture improvement and the number of cores also come into play when it comes to more complex tests like the prime number calculation test. The test finds all prime numbers from 2 to 10,000,000 and uses the *sqrtO* function and uses nested loop and if statements. From this benchmark result we can see that the newer the generation the lower time it takes to complete the

benchmark. Here from the multi-threaded test, we see that the 5th gen i5 5200u takes longer on average than the 7th gen i5 7200u in the test despite having the same number of cores and higher clock speed.

Another noticeable thing from the benchmarks is that the single threaded benchmarks take almost the same time as the multi-threaded tests as long the number of physical CPU cores/threads are greater than the number of virtual threads. Here, we can see the only one different for core i5 7200U which takes almost double time to calculate the integer time calculation value. The 6 core Core i5 9400F takes almost the same time for the single threaded test and the 4 threaded test. But it starts taking longer in the 8 threaded and 16 threaded tests due to running out of available CPU cores to assign the threads to. The 2 core i5 7200U and 5200U suffers heavily in the multi-threaded tests due to having less CPU cores.

Summary of Findings:

From the result analysis part, we can say that the performance of CPU not only depends on its clock frequency but also depends on the number of core and thread. Those have more cores and threads they are performing better than others. Also, the generational architecture development plays a big role here. Like different CPU has different bus speed, maximum memory bandwidth and turbo boost technology to speed up the performance of a CPU.

Conclusion:

We can see a clear trend from the benchmark results. As long as the numbers of virtual thread are lower than the number of physical CPU cores/threads, the total time for multi- threading doesn't take much longer compared to the single threaded program. But as soon as the numbers of thread are greater than the number of cores of that CPU, the multi- threaded benchmarks start to take a lot longer.

And another thing can be noticed from the benchmarks across multiple CPUs is that the latest or the higher clock frequency a CPU has, the lower time it takes for completing any particular benchmark. Thus, we think this program is fairly successful in comparing the speeds of various CPU and therefore it should work as a nice standard benchmark for comparing various x64 CPUs.

Future Works:

As for future improvements, we can add additional CPU heavy tasks into our benchmark programs such as ray-tracing, Fourier calculation, Matrix inversing etc. This benchmarking is not for cross platform. So, there is a chance to make this benchmark for cross platform and allow people to run it on arm / other RISC processors too and validate their performance.

References:

1. *Measuring time: https://stackoverflow.com/questions/5248915/execution%C2%AD%20time- of-c-program*

2. *Reliable Benchmarking: requirements and Solution:*

   https://link.springer.com/article/10.1007/s10009-017-0469-y