Creating a Benchmark

Stanciu Alexander
Group 30433

Table of Contents

1. <u>Description</u>

A benchmark is a “Standard tool for the competitive evaluation and comparison of competing systems or components according to specific characteristics, such as performance, dependability, or security”[1]. For this project, I will try to create a benchmark which will evaluate the performance of the CPU and give an overall score.

2. <u>Goals</u>

The goal of this project is to create a desktop benchmark which will measure and evaluate the performance of the CPU. The language used for the project will be Java with the intention of making the benchmark portable. The program will be easy to use with the possibility of including your own program for “application testing”.

Some of the main objectives that I want to reach are:

- Repeatability: running the same tests on the same machine should give about the same score
- Modularity: test can be changed without much effort and the user can put in his own programs to test their performance.
- Relevance: the end result should be easy to interpret.

3. Specifications

3.1. Types of tests that will be performed:

- **Integer Test:** aims to measure how fast the CPU can perform mathematical integer operations. An integer is a whole number with no fractional part. This is a basic operation in all computer software and provides a good indication of 'raw' CPU throughput. The test uses large sets of an equal number of random 32-bit and 64-bit integers and adds, subtracts, multiplies and divides these numbers [3].
- **Floating Point Test:** performs the same operations as the Integer Maths Test however with floating point numbers, using an equal amount of single precision (32-bit) and double precision (64-bit) values. A floating-point number is a number with a fractional part (ie. 12.568). These kinds of numbers are handled quite differently in the CPU compared to Integer numbers as well as being quite commonly used, therefore they are tested separately [3].
- **The Prime Number Test:** aims to test how fast the CPU can search for Prime numbers, reported as operations per second. A prime number is a number that can only be divided by itself and 1. For example, 1, 2, 3, 5, 7, 11 etc. This algorithm uses loops and CPU operations that are common in computer software, the most intensive being multiplication and modulo operations. All operations are performed using 64-bit integers [3].
- **String Sorting Test:** uses the quick sort algorithm to see how fast the CPU can sort strings (single byte characters). A very common task in many applications [3].

3.2. Metric

The metric used will be similar to the SPEC metric: there will be a few benchmark programs which will be started and timed. Then, the time of a test system is compared to the reference time and a ratio is computed: testTime/refTime. That ratio becomes the Score for that test[5]. The reference time will be the time a test will take to complete on my PC which will have by definition a score of 100. After all the tests have been run, a Geometric Mean(GM) will be calculated since is the one that SPEC chose after extensive debates[2].

3.3. Actual Tests

- **Integer Tests:** consists of 10 operations (addition, subtraction, multiplication, division, bitwise shifts, AND, OR, XOR, NOT) made with both 32-bit and 64-bit randomly generated numbers and we measure the total time needed for each test to be completed.
    - Tests 1 through 10 will be each operation used by its own on a total of 5000000 numbers/test
    - Test 11 will use all the arithmetic operations on a total of 5000000 numbers

  The final result will be the GM of all the partial results.
- **Floating Point Test:** similar to integer tests, we will use the arithmetic operations with randomly generated floats and doubles
- **The Prime Number Test:** find the prime first 20000 prime numbers.
- **String Sorting Test:** will use 500000 strings of 1000 characters and qsort to sort them and measure the time needed.
- **Final Result:** GM of all the tests.

3.4. Components

- **GUI:** The Graphical User Interface will be the way the user will be able to interact with the program. The possible actions will be:
    - Run a specific test
    - Run all the tests
    - View results
    - Select what 3rd party program to be measured and get the time of execution not a score

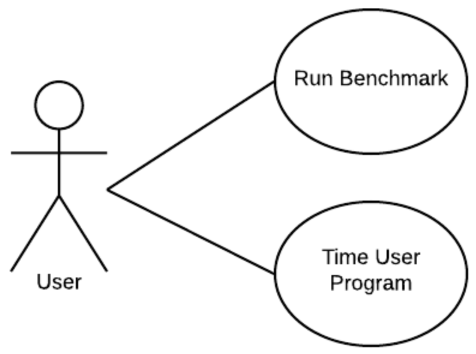
  The GUI will be built with JavaFX.
- **Test modules:** The part of the benchmark that puts the CPU to the test by requesting numerous operations to be executed pushing the CPU to the limit so it can be measured. The test modules could in theory be built using any programming language, but for the sake of simplicity, they will be built in java as well.

  Using the Random class, we can generate with the method nextX where X is :
    - Int for 32 bit integers
    - Long for 64 bit integers
    - Float for 32 bit floating point numbers
    - Double for 64 bit floating point numbers
    - Bytes to generate byte arrays

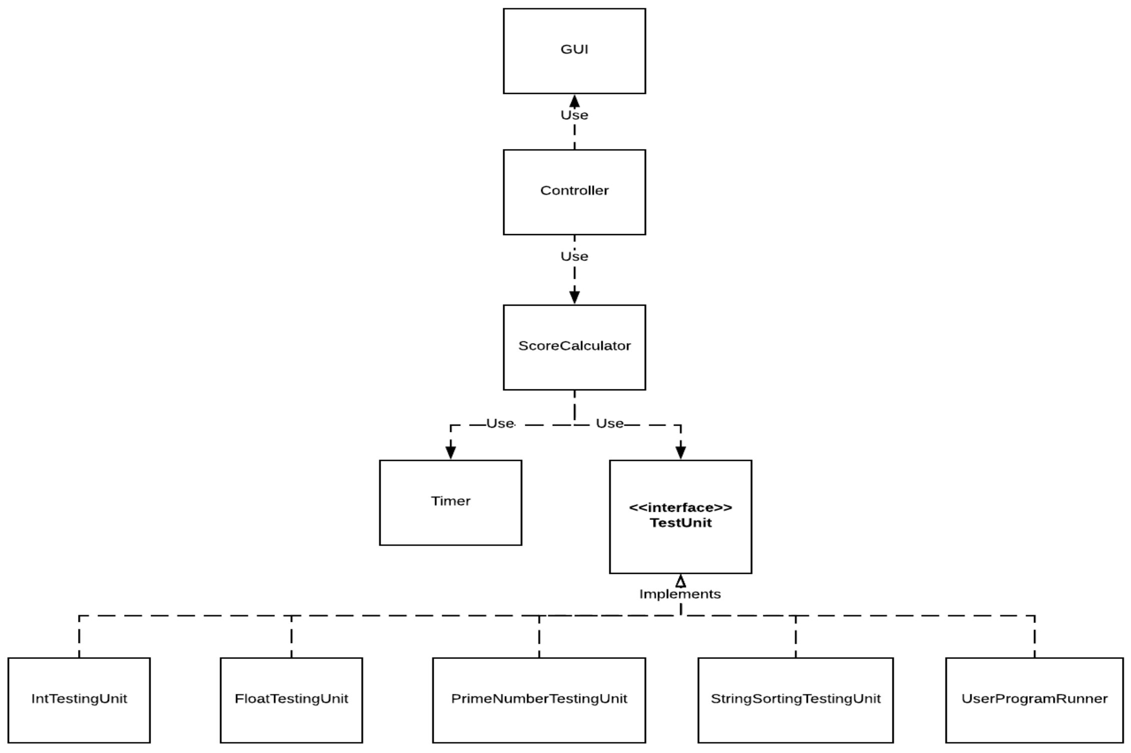
- **Timer:** Part of the program that measures the time needed for the tests to finish. Will use the System.nanTime() to get the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds before and after the test execution to obtain the total time needed for the execution of the test.[4]
- **Controller:** Will make the connections between the GUI, Test modules and the timer

4. Diagrams

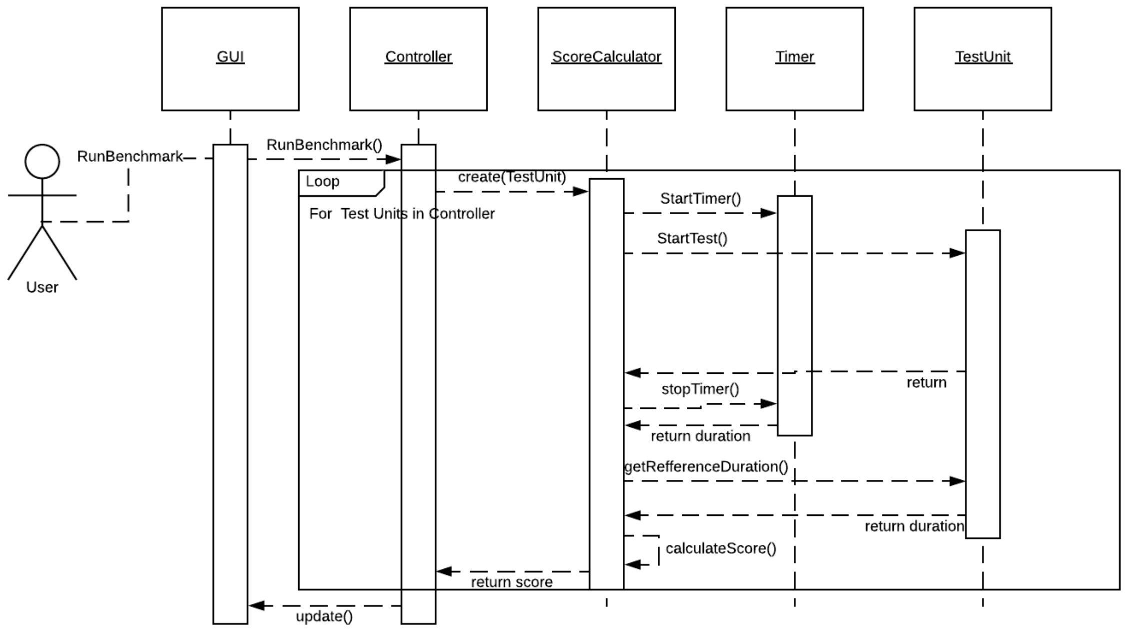
Use-case diagram:



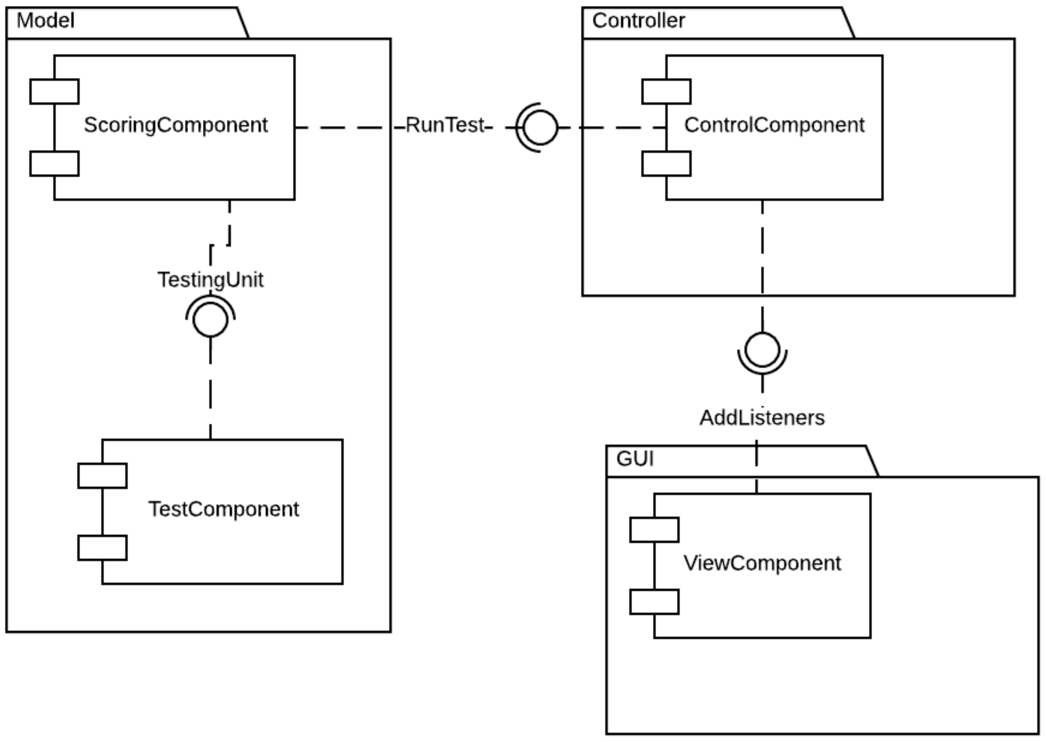
Class diagram:



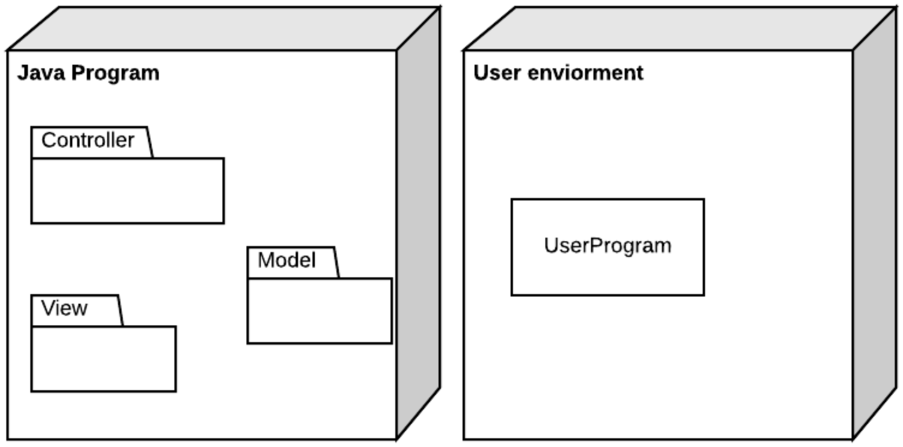
Sequence Diagram:



Component Diagram:



Deployment Diagram:



5. Bibliographic study

"How to Build a Benchmark", by Jóakim v. Kistowski, Jeremy A. Arnold , Karl Huppler, Klaus-Dieter Lange and John L. Henning, Paul Cao.[1]

This general ideas of this article are the following:

A benchmark should have these key characteristics:

- **Relevance:** The results of the benchmark should be relevant to the one who performed the test and wanted to draw conclusions from it
- **Reproducibility:** Running the benchmark on the same system should produce similar results
- **Fairness:** The benchmark should not favor one producer or another, it should give equal chances to all producers and systems
- **Verifiability:** Providing confidence that a benchmark result is accurate
- **Usability:** Avoiding roadblocks for users to run the benchmark in their test environments

There are three types of benchmarks:

1. **Specification-based:** begin with a definition of a business problem to be simulated by the benchmark. The key criteria for this definition are the relevance topics.
2. **Kit-based:** provides a near "load and go" implementations that greatly reduce the cost and time required to run the benchmarks
3. **Hybrid:** the benchmark can be provided in a kit, but there is a desire to allow some functions to be implemented at the discretion of the individual running the benchmark

Kaivalya M. Dixit "Overview of the SPEC Benchmarks"[2]

In this article we identify three types of popular benchmarks: kernel, synthetic, and application.

- **Kernel benchmarks** are based on analysis and the knowledge that in most cases 10 percent of the code uses 80 percent of the CPU resources. Performance analysts have extracted these code fragments and have used them as benchmarks.
- **Synthetic benchmarks** are based on performance analysts' experience and knowledge of instruction mix. Examples of synthetic benchmarks are Dhrystone and Whetstone. New technologies and instruction set architectures make some older assumptions regarding instruction mix obsolete.
- From a user's perspective, the best benchmark is the **user's own application program**. Examples of application benchmarks are Spice (circuit designers) and GNU compiler (software developers using GNU environments).

6. Implementation:

The implementation was done according to the class diagrams with the mention that Score Calculator was named Benchmark and the Timer is the built in System.nanoTime() before and after running the test and subtracting them to get the time needed for the program to run the tests. Another mention is that due to actual times not being displayed, the user program test was not implemented.

Because the GUI is a thread and running the benchmarks from it would block the GUI until the tests were done, the benchmarks have to be ran on a different thread. Because of this, Benchmark implements runnable and when a test is run, a new thread is created for that benchmark. This results in the GUI not being blocked, but it also introduces another problem which is that multiple benchmarks can be ran at the same time which results in lower performance for both benchmarks thus running each benchmark is recommended and running all the standard tests starts the first and when it finishes, the other tests start one after another.

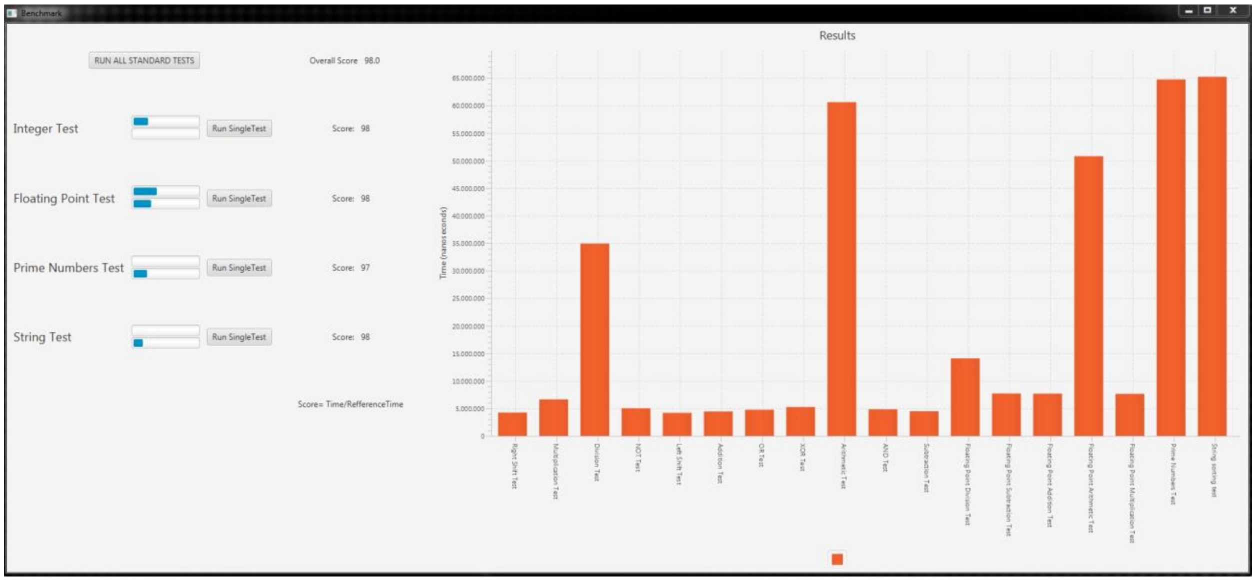
Once the GUI starts a benchmark, the Benchmark, which has a reference to a Test Unit, gets the number of tests in the test unit and starts timing each test a set number of times. The times for each batch of a single test are sorted and the first and last few times are eliminated and the rest of the times are summed and divided by their number. Doing this for each test yields a list of times in nanoseconds for each test. To get a score for each test, the times are divided by the Reference time of each test. This reference time is the time obtained by running the benchmark on my PC (CPU: AMD FX-6100 Core:3.3GHz Buss: 200 MHz, RAM: 16GB Dual-Channel DDR3 665MHz) thus giving my PC a definition score of 100. To get an overall score, a geometric mean is performed on the scores obtained previously. The benchmark also extends Observable and whenever an iteration of a single test is finished or a new test starts running or all the tests are finished, the observers are notified with a list of arguments which contains as the first element the percentage of tests that we have already tested, the second argument is the percentage of iterations we have finished for the current test and the third argument is 1 if the benchmark finished. The test units are implemented as they were specified earlier.

7. Testing:

The testing was done manually by running the benchmarks multiple times and observing the results with no other program running. The maximum variation between scores was of at most 3% of the first score for individual testing units, but most of the time it was of 1%. This also is true for when running all the tests. The variance is larger when other programs are running because the operating system interferes and may stop the running test and let other processes use the cpu while the timer is still running.

At the beginning “run all standard tests” was starting all the threads/benchmarks at once.

In the following image a normal test (each test run separately) was performed and after that, the “run all standard tests” was pressed. The test was done on the PC so the scores are close to 100, but a little below it due to a few other programs running in the background, but the results are still in the 3 point range.



Overall Score 98.0

Integer Test Score: 98

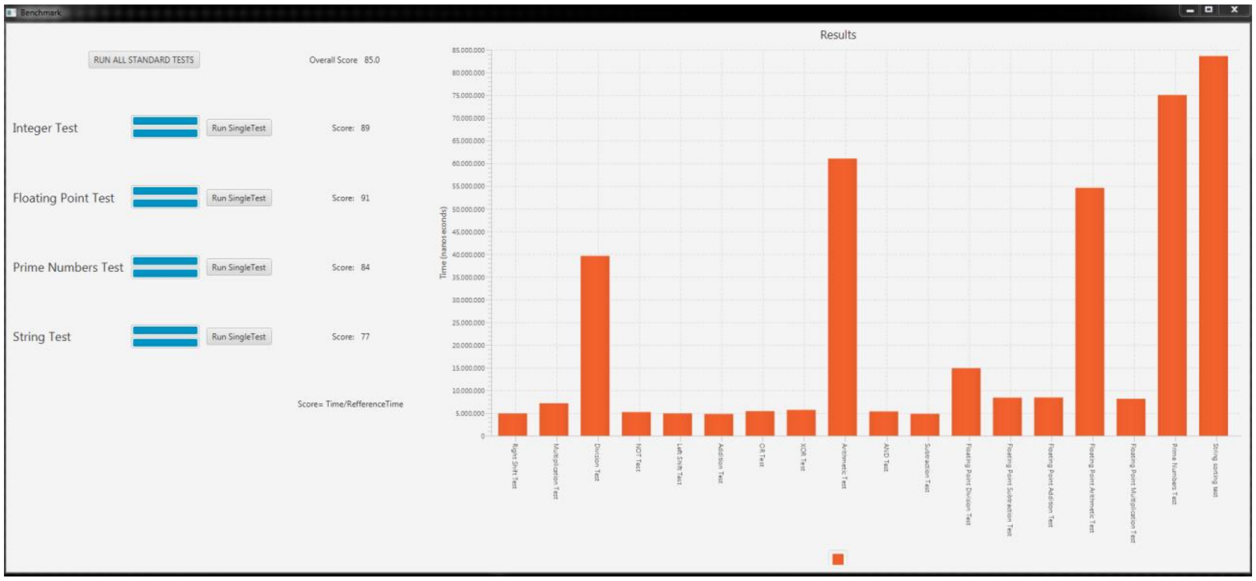
Floating Point Test Score: 98

Prime Numbers Test Score: 97

String Test Score: 98

Score= Time/RefferenceTime

After all the tests finished, the results were significantly lower than expected. This is because overall the whole program runs faster, but the individual tests are slower because they are overlapping when using the CPU.



Overall Score 85.0

Integer Test Score: 89

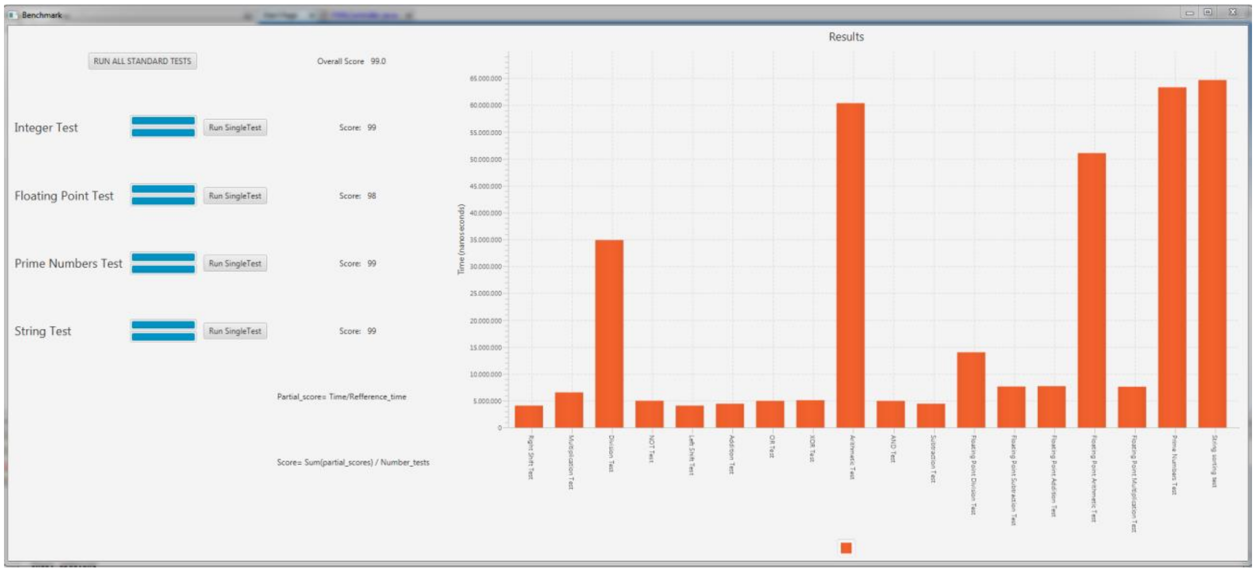
Floating Point Test Score: 91

Prime Numbers Test Score: 84

String Test Score: 77

Score= Time/RefferenceTime

Because this is not very convenient, running “run all standard tests” was modified to run the first test and when it finishes, all the other tests will start one after the other, bringing the result in the expected range.



Overall Score 99.0

Integer Test Score: 99

Floating Point Test Score: 98

Prime Numbers Test Score: 99

String Test Score: 99

Partial_score= Time/Refference_time

Score= Sum(partial_scores) / Number_tests

8. References:

[1] Jóakim v. Kistowski, Jeremy A. Arnold , Karl Huppler, Klaus-Dieter Lange and John L. Henning, Paul Cao. (February 2015) “How to Build a Benchmark”

[2] Kaivalya M. Dixit “Overview of the SPEC Benchmarks”

[3] PassMark Software – CPU Benchmarks

[4] Java Doc

[5] Wikipedia SPECint

[6] Stack Overflow