CO PROJECT TITLE

Project realised by *FileList*

Students:

Căuneac Răzvan Virtual Memory

Dohr Egon FOPS/MOPS

Goga Florin LOOP UNROLLING

Goldiş Răzvan ORGANISATION/DOCUMENTATION

Hepcal Lucian PIBBP

Timişoara
May, 2023

Chapter 2

Introduction

This chapter should not contain more than one page in this context. Under other circumstances, the introduction should contain more aspects and should be more detailed, but, in this case, even one-half of one page is perfect. :)

2.1 Context

- We have decided to implement CPU and RAM benchmarks, namely calculating the digits of PI, calculating MOPS and MFOPS, checking the stack performance by finding prime numbers using recursive loop unrolling and checking the RAM performance by writing to memory a 1 GiB file.

2.2 Motivation

- We decided to implement various benchmarks so that we could cover more use cases of the computer raw power with Pi benchmark, floating point operations, stack capabilities, and RAM speed;

Chapter 3

State of the art

2 benchmarks that inspired our project were the GMP benchmark for calculating the digits of Pi and Prime95 for calculating the prime numbers.

Compared to GMP Pi , our benchmark uses the **Bailey–Borwein–Plouffe formula** based on the series

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

While GMP uses a variation of **Ramanujan** algorithm namely **Chudnovsky**

$$\frac{1}{\pi} = 12 \sum_{q=0}^{\infty} \frac{(-1)^q (6q)! (545\,140\,134q + 13\,591\,409)}{(3q)!(q!)^3 \, (640\,320)^{3q+\frac{3}{2}}}$$

For the prime numbers algorithm we are using it to find the last prime number before the stack if full with various levels of loop unrolling, while the Prime95 is used to stress test the reliability of computations under prolonged usage

Chapter 4

Design and implementation

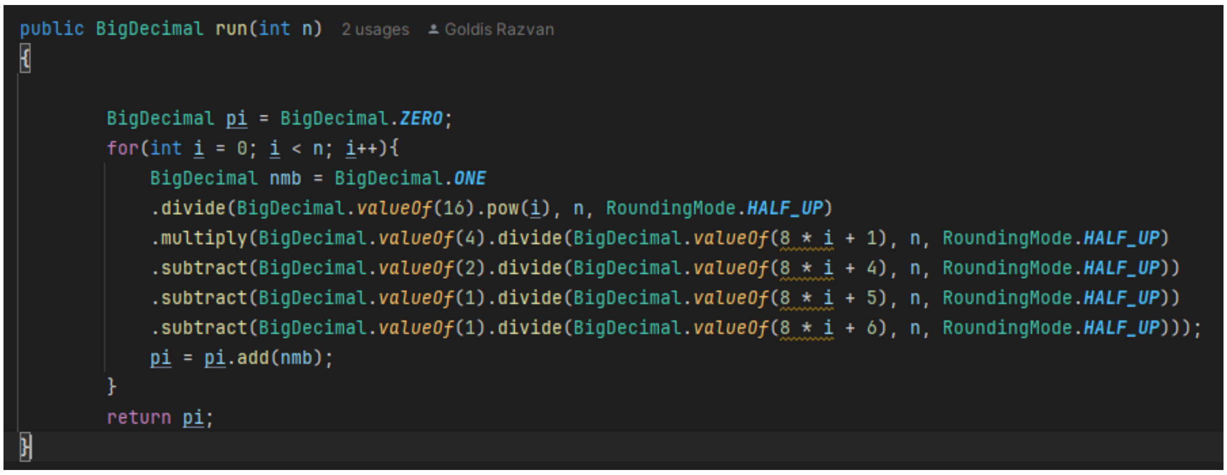
Pi BBP

Benchmark Features and Measurements

- **Precision Control:** The user can specify the number of decimal places to which π should be computed.
- **Performance Measurement:** The benchmark measures the time taken to compute π to the specified precision, providing insights into the computational efficiency of the hardware.
- **Warm-Up Phase:** Includes a warm-up method to ensure that the JVM is optimized before actual benchmarking starts.

Implementation

The core algorithm used in this benchmark is the BBP formula, which is implemented as follows:



```
public BigDecimal run(int n)   2 usages   Goldis Razvan
{

        BigDecimal pi = BigDecimal.ZERO;
        for(int i = 0; i < n; i++){
            BigDecimal nmb = BigDecimal.ONE
            .divide(BigDecimal.valueOf(16).pow(i), n, RoundingMode.HALF_UP)
            .multiply(BigDecimal.valueOf(4).divide(BigDecimal.valueOf(8 * i + 1), n, RoundingMode.HALF_UP)
            .subtract(BigDecimal.valueOf(2).divide(BigDecimal.valueOf(8 * i + 4), n, RoundingMode.HALF_UP))
            .subtract(BigDecimal.valueOf(1).divide(BigDecimal.valueOf(8 * i + 5), n, RoundingMode.HALF_UP))
            .subtract(BigDecimal.valueOf(1).divide(BigDecimal.valueOf(8 * i + 6), n, RoundingMode.HALF_UP)));
            pi = pi.add(nmb);
        }
        return pi;
}
```

This loop iterates **n** times, each time computing a portion of π and adding it to the cumulative sum. The **BigDecimal** class is used for high-precision arithmetic.

Floating point Operations

- **Execution Time**: Using **System.nanoTime()**.

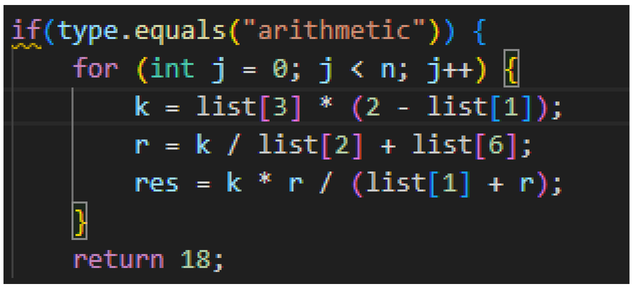
Implementation

Modes of operation:

- **Integer arithmetic test:** Use a wide array of operators between local variables.
- **Branching test**: Jump to different spots in an array.
- **Array access**: jump randomly to different spots in an array

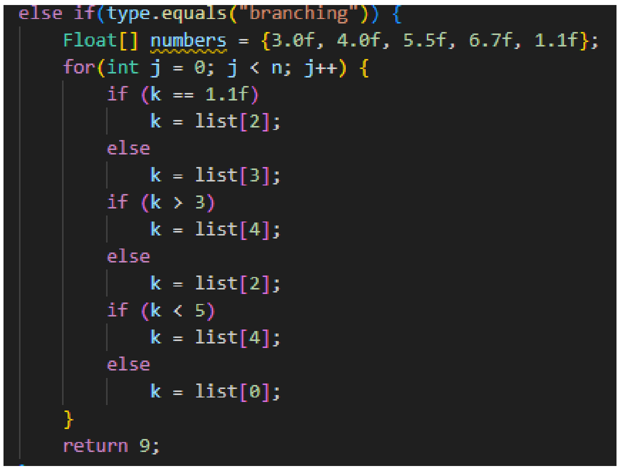
Example Code

Integer arithmetic test



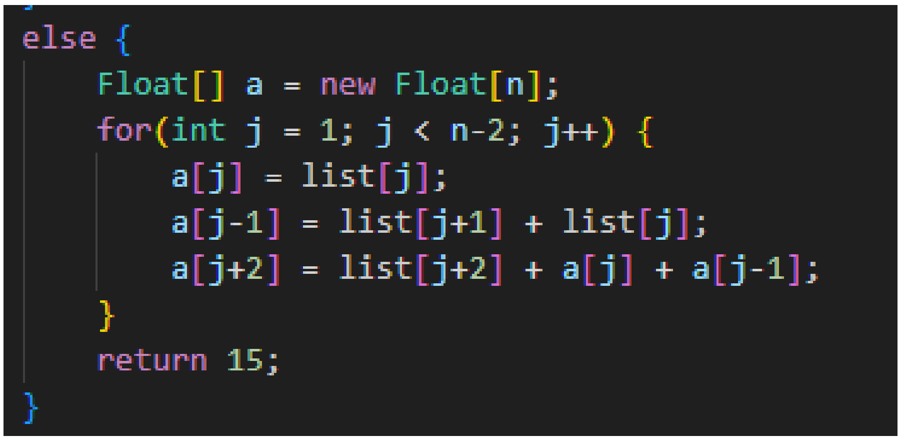
```
if(type.equals("arithmetic")) {
    for (int j = 0; j < n; j++) {
        k = list[3] * (2 - list[1]);
        r = k / list[2] + list[6];
        res = k * r / (list[1] + r);
    }
    return 18;
```

Branching test:



```java
else if(type.equals("branching")) {
    Float[] numbers = {3.0f, 4.0f, 5.5f, 6.7f, 1.1f};
    for(int j = 0; j < n; j++) {
        if (k == 1.1f)
            k = list[2];
        else
            k = list[3];
        if (k > 3)
            k = list[4];
        else
            k = list[2];
        if (k < 5)
            k = list[4];
        else
            k = list[0];
    }
    return 9;
```

Array access:



```java
else {
    Float[] a = new Float[n];
    for(int j = 1; j < n-2; j++) {
        a[j] = list[j];
        a[j-1] = list[j+1] + list[j];
        a[j+2] = list[j+2] + a[j] + a[j-1];
    }
    return 15;
}
```

Loop Unrolling

Execution Time: Using **System.nanoTime()**.

- **Prime Numbers Found**: Counted during recursion.
- **Performance Score: (primesFound / elapsedTime) * 1000**.

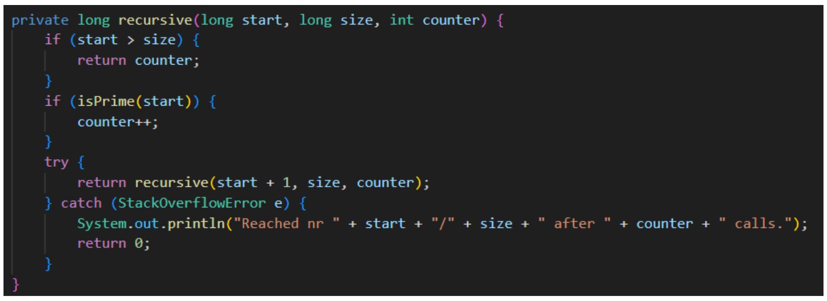
Implementation

Modes of operation:

- **Regular Recursion**: Checks each number individually.
- **Unrolled Recursion**: Checks multiple numbers per call to reduce function calls.

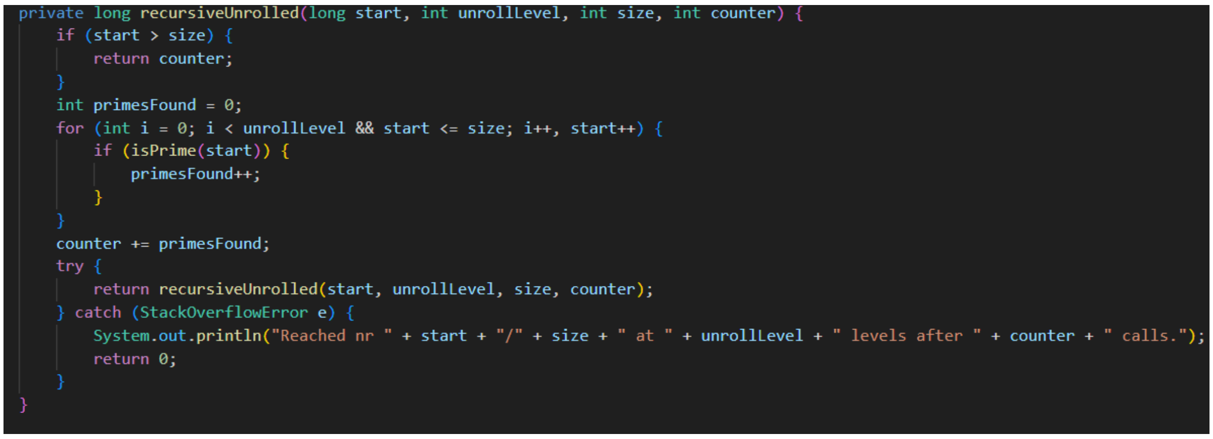
Example Code

Regular Recursion:



```java
private long recursive(long start, long size, int counter) {
    if (start > size) {
        return counter;
    }
    if (isPrime(start)) {
        counter++;
    }
    try {
        return recursive(start + 1, size, counter);
    } catch (StackOverflowError e) {
        System.out.println("Reached nr " + start + "/" + size + " after " + counter + " calls.");
        return 0;
    }
}
```

Unrolled Recursion:



```java
private long recursiveUnrolled(long start, int unrollLevel, int size, int counter) {
    if (start > size) {
        return counter;
    }
    int primesFound = 0;
    for (int i = 0; i < unrollLevel && start <= size; i++, start++) {
        if (isPrime(start)) {
            primesFound++;
        }
    }
    counter += primesFound;
    try {
        return recursiveUnrolled(start, unrollLevel, size, counter);
    } catch (StackOverflowError e) {
        System.out.println("Reached nr " + start + "/" + size + " at " + unrollLevel + " levels after " + counter + " calls.");
        return 0;
    }
}
```

Virtual Memory Read/Write

The Virtual Memory Benchmark is designed to measure read and write speeds when interacting with a file that is mapped into virtual memory.

- Write Speed: The rate at which data can be written to the memory-mapped file.
- Read Speed: The rate at which data can be read from the memory-mapped file.

Implementation Details:

Memory Mapper:

- Purpose: Maps files into memory and provides read/write methods.
- Key Methods:
    - put(long offset, byte[] src): Writes to the memory-mapped file.
    - get(long offSet, int size): Reads from the memory-mapped file.

VirtualMemoryBenchmark:

- Purpose: Uses MemoryMapper to measure read/write speeds.
- Key Methods:
    - initialize(Object… params): Sets file and buffer sizes.
    - run(long fileSize, int bufferSize): Performs and times read/write operations.
    - getResult(): Returns formatted speed results.

Write Operation:

```
long startTime = System.nanoTime();
for (long i = 0; i < fileSize; i += bufferSize) {
    memoryMapper.put(i, buffer);
}
long endTime = System.nanoTime();
writeSpeed = (fileSize / 1e6) / ((endTime - startTime) / 1e9); // MB/s
```

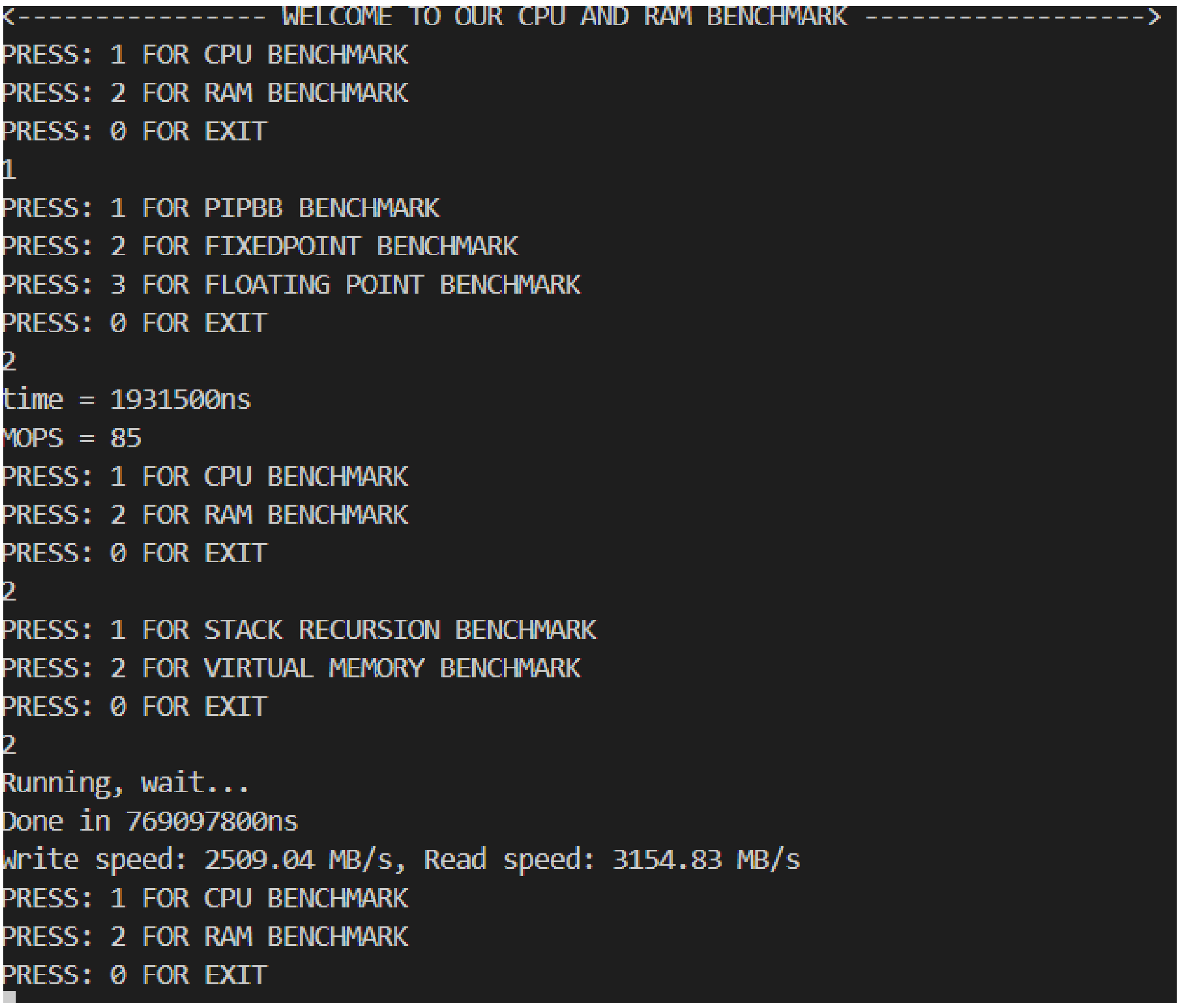
Read Operation:

```
long startTime = System.nanoTime();
for (long i = 0; i < fileSize; i += bufferSize) {
    memoryMapper.get(i, bufferSize);
}
long endTime = System.nanoTime();
readSpeed = (fileSize / 1e6) / ((endTime - startTime) / 1e9); // MB/s
```

Chapter 5

Usage

Regarding the usage of our benchmark, we have a simple in terminal user interface where you have to choose if you want to test the CPU or the memory and then run the benchmark. Some benchmarks may need additional data for example how many digits of pi do you want to calculate.



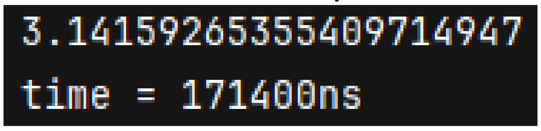
```
<---------------- WELCOME TO OUR CPU AND RAM BENCHMARK ------------------>
PRESS: 1 FOR CPU BENCHMARK
PRESS: 2 FOR RAM BENCHMARK
PRESS: 0 FOR EXIT
1
PRESS: 1 FOR PIPBB BENCHMARK
PRESS: 2 FOR FIXEDPOINT BENCHMARK
PRESS: 3 FOR FLOATING POINT BENCHMARK
PRESS: 0 FOR EXIT
2
time = 1931500ns
MOPS = 85
PRESS: 1 FOR CPU BENCHMARK
PRESS: 2 FOR RAM BENCHMARK
PRESS: 0 FOR EXIT
2
PRESS: 1 FOR STACK RECURSION BENCHMARK
PRESS: 2 FOR VIRTUAL MEMORY BENCHMARK
PRESS: 0 FOR EXIT
2
Running, wait...
Done in 769097800ns
Write speed: 2509.04 MB/s, Read speed: 3154.83 MB/s
PRESS: 1 FOR CPU BENCHMARK
PRESS: 2 FOR RAM BENCHMARK
PRESS: 0 FOR EXIT
```

Step-by-Step Instructions for PIBBP

Clone the Repository Step-by-Step Instructions for PIBBP

1. **Download the Program**
2. **Compile the Program**
3. **Run the benchmark**
4. **Output**
    - the computed value of π to the specified precision.
    - The time taken to perfom the computation, in nanoseconds.

5. **Output Description**

6. The benchmark outputs the calculated value of π and the total computation time. This information is printed directly to the console. For example: 

```
3.14159265355409714947
time = 171400ns
```

Benchmark Results

7. The following results were obtained by running the benchmark on two different machines with varying hardware configurations:

Machine	CPU	Precision(n)	Time(ns)
Machine 1	Ryzen 7 5800x	100	3433700
Machine 2	Ryzen 5 2600	100	5040600
Machine 1	Ryzen 7 5800x	1000	55483900
Machine 2	Ryzen 5 2600	1000	140399700
Machine 1	Ryzen 7 5800x	5000	696676700
Machine 2	Ryzen 5 2600	5000	7245279700

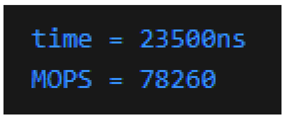
Step-by-Step Instructions for FOPS/MOPS

Steps to Use

1. **Download and Compile**:
   - Clone the repository.
   - Compile with **javac FloatingPointArithmeticBench.java**.
2. **Run the Benchmark**:

java FloatingPointArithmeticBench

Example output:



```
time = 23500ns
MOPS = 78260
```

Chapter 6

Results Summary

Graphical representation is optional. Numerical results:

CPU	Bench	Time(ns)	Mops
Ryzen 5 7600	**Branching**	**23500**	**78260**
Ryzen 5 7600	**Array access**	**64200**	**46875**
Intel i3-6100	**Branching**	**36250**	50000

Step-by-Step Instructions for Virtual Memory Benchmark

Steps to Use
- Download code.
- Clone the repository.
- Compile with javac TestVirtualMemory.java
- Run the Benchmark.

Example output:
Write speed: 350.24 MB/s, Read speed: 420.11 MB/s.

Portability and Execution

- Standalone: No external libraries required.
- No Special Input: Parameters are hardcoded but modifiable in the source.
- Simple Execution: Run with a single command.

Chapter 6:

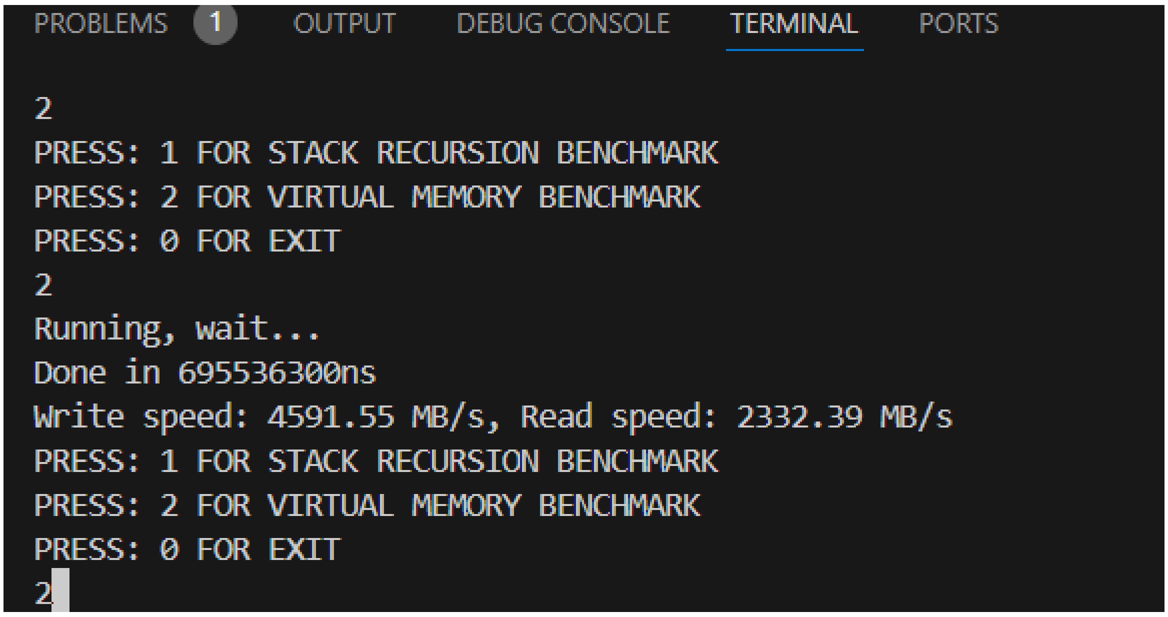
To analyze the performance of different hardware configurations, we executed the Virtual Memory Benchmark on five distinct machines. The benchmark measured read and write speeds with a 1GB file size and a 4KB buffer size. The results are presented graphically below:

Machine	CPU	RAM	STORAGE	OS	Write Speed (MB/s)	Read Speed (MB/s)
1	Ryzen 5 7600x	32GB	SSD	Windows 11	4591.55	2332.39
2	Intel i7-12700	16GB	SSD	Windows 11	2352.57	4034.83
3	Intel i3-6100	4GB	HDD	Windows 10	100.25	150.33

Analysis and Interpretation

Key Observations:

1. Storage Type Impact: Machines with SSD showed significantly higher read and write speeds compared to those with HDDs. This highlights the performance advantage of SSDs over traditional HDDs.
2. RAM and CPU Influence: Higher RAM and newer CPUs resulted in better performance.



```
2
PRESS: 1 FOR STACK RECURSION BENCHMARK
PRESS: 2 FOR VIRTUAL MEMORY BENCHMARK
PRESS: 0 FOR EXIT
2
Running, wait...
Done in 695536300ns
Write speed: 4591.55 MB/s, Read speed: 2332.39 MB/s
PRESS: 1 FOR STACK RECURSION BENCHMARK
PRESS: 2 FOR VIRTUAL MEMORY BENCHMARK
PRESS: 0 FOR EXIT
2
```

Steps to Use for LOOP UNROLLING

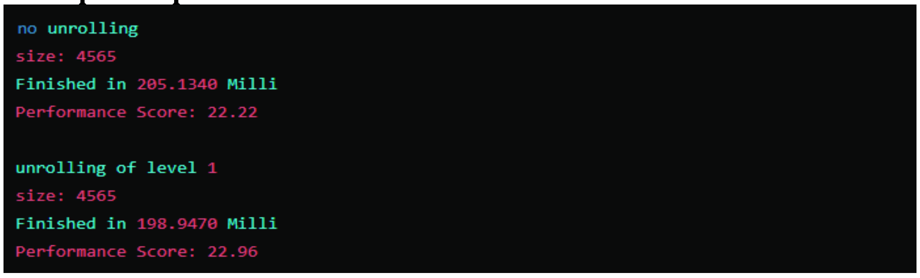
1. **Download and Compile**:

    - Clone the repository.
    - Compile with **javac CPURecursionLoopUnrolling.java**.

2. **Run the Benchmark**:

java CPURecursionLoopUnrolling

Example output:



```
no unrolling
size: 4565
Finished in 205.1340 Milli
Performance Score: 22.22

unrolling of level 1
size: 4565
Finished in 198.9470 Milli
Performance Score: 22.96
```

Results Summary

Graphical representation is optional. Numerical results:

Machine	CPU	Unrolling Level	Execution Time (ms)	Performance Score
M1	Intel i7	0	205.134	22.22
M1	Intel i7	1	198.947	22.96
M2	AMD Ryzen 5	0	210.567	21.66

Analysis

Loop unrolling improves performance but with diminishing returns at higher levels. Stack overflow errors occurred at very high unroll levels.

Chapter 7

Conclusions

Pi BBP

- I leaned about the Bailey-Borwein-Plouffe formula for computing Pi's digits. It's also the first formula for computing Pi's digits I've ever heard of.
- My team was very oganized, each one of was took a part and delivered it on time making the progress easy.

I would give myself a mark of 9. I implemented the benchmark for computing Pi’s digits and I have also helped with the other benchmarks in this project.

MOPS/FOPS

Conclusion: I learned about the differences of time it takes between different machines and different types of opperations.

Self-Evaluations : I would give myself 7/10.

LOOP UNROLLING

Conclusion: By working along in this team project i learned about the benefits and limitations of loop unrolling and effective CPU performance benchmarking.

Self-Evaluations : I would give myself 8/10 because I helped were it was needed every time.

Bibliography

[1] *https://en.wikipedia.org/wiki/Bailey–Borwein–Plouffe_formulaInserting*

[2] *Git repository*. https://github.com/GoldRyan22/DC_BENCH_PROJ

[3] https://observablehq.com/@galopin/the-chudnovsky-algorithm-for-calculating-pi

[4] https://gmplib.org/pi-with-gmp

[5] https://www.geeksforgeeks.org/loop-unrolling/

[6] https://www.mersenne.org/download/