Exploration and Presentation

Assignment 3 - Optimization

Thomas Stevns Nielsen Ebsen

Jonas Valentin Björk Hein

Jonatan Magnus Bakkke

Denmark, April 14, 2021

Contents

Chapter 1

Hand-in

1.1 Introduction

We chose to optimize the program "LetterFrequencies" that were provided to us by the school. The program opens text file, counts all of the occurring letters in the file and then prints the result.

Our assignment were to optimize the programs performance with at least 50%, which we achieved by replacing the programs "reader" with a "bufferedReader".

1.2 Documentation of the current performance

To properly measure the performance of our program, we needed to document the execution time of the program. We created a timer class and implemented it into the program to measure it's performance.

Figure 1.1: Timer.java

```
public class Timer {
    private long start, spent = 0;
    public Timer() { play(); }
    public double check() { return (System.nanoTime()-start+spent)/1e9;}
    public void pause() { spent += System.nanoTime()-start; }
    public void play() { start = System.nanoTime(); }
}
```

Figure 1.2: Main.java

```
public static void main(String[] args) throws FileNotFoundException, IOException {
    String fileName = System.getProperty("user.dir") + "/src/main/resources/FoundationSe
    Timer timer = new Timer();
    timer.play();
    FileReader fileReader = new FileReader(fileName);
    Reader reader = new Reader(fileReader);
    Map<Integer, Long> freq = new HashMap<>();
    tallyChars(reader, freq);
    print_tally(freq);
    System.out.println("Time: " + t.check())
}
```

With our timer implemented, we then ran the program for a total of 10 times and took the average of our timings to present a baseline of performance.

```
Run 1:   152.073 ms
Run 2:   104.019 ms
Run 3:   66.9943 ms
Run 4:   88.3265 ms
Run 5:   64.8921 ms
Run 6:   52.9183 ms
Run 7:   53.6571 ms
Run 8:   53.3412 ms
Run 9:   96.7388 ms
Run 10:  87.2514 ms
```

As we can see by our timings, the time to execute the program varies greatly. The reason our timings are different for each run, is due to the pc running other processes during the test. In hindsight we should have run the test in a virtual environment to prevent any outside interferences, however by running the code 10 times and getting the average runtime we circumvent some of these interferences to get a clearer result.

We can now use our data to do a couple of different calculations to get an even even clearer picture of our programs performance before we improve upon it.

Count:	10
Sum:	820.2117
Mean:	82.02117
Variance:	866.8134585601
Standard Deviation:	29.441695918546

1.3 Bottlenecks and hypothesis of issues

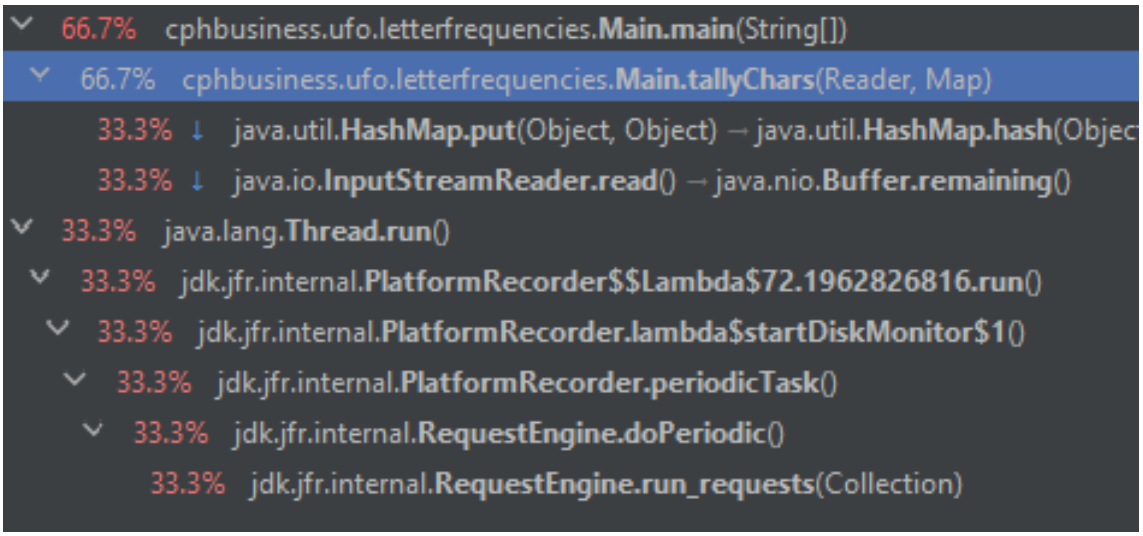
1.3.1 Finding the bottlenecks

To improve the programs performance, first of all we had to figure out what methods would give us the best performance increase once optimized.

We ran the program using intelliJ's built profiler named Flight Recorder to see which methods were the most performance heavy.

Figure 1.3: Profiler Flame Chart

Figure 1.4: Profiler Call Tree



Looking at the charts, we clearly see that the "tallyChars" method is the most costly function.

We ran the timer on the method 10 times and got an average of around 4.5ms, which is fairly high compared to other methods, therefor we decided to look further into it.

1.3.2 Hyphotesis - why is tallyChart so expensive

The method is using a "Reader" to read data from a text file, which in our experience can be a costly method to run. After an hour of extensively searching the internet, we came across a reputable source which explained that:

> The new JDK 1.1 improves I/O performance with the addition of a collection of Reader and Writer classes. The readLine method in BufferedReader is at least 10 to 20 times faster than the one in DataInputStream when a large file is encountered. [1]

We replaced the old "Reader" with the new "BufferedReader" and gave it a max resource of 10.000 to satisfy it.

Figure 1.5: Main.java with changed Reader type

```
public static void main(String[] args) throws FileNotFoundException, IOException {
    String fileName = System.getProperty("user.dir") + "/src/main/resources/FoundationSeries.txt";
    Timer timer = new Timer();
    timer.play();
    //Reader reader = new FileReader(fileName);
    FileReader fileReader = new FileReader(fileName);
    BufferedReader reader = new BufferedReader(fileReader,10000);
    Map<Integer, Long> freq = new HashMap<>();
    tallyChars(reader, freq);
    print_tally(freq);

    System.out.println("Time: " + t.check())
}
```

Figure 1.6: Main.java with updated tallyChars arguments

```
private static void tallyChars(BufferedReader reader, Map<Integer, Long> freq) throws IOException {
    int b;
    while ((b = reader.read()) != -1) {
        try {
            freq.put(b, freq.get(b) + 1);
        } catch (NullPointerException np) {
            freq.put(b, 1L);
        };
    }
}
```

We performed the same test again with a marginal increase in performance.

```
Run 1:  82.226 ms
Run 2:  34.3938 ms
Run 3:  58.0026 ms
Run 4:  40.7641 ms
Run 5:  30.4765 ms
Run 6:  39.6008 ms
Run 7:  41.5937 ms
Run 8:  42.4653 ms
Run 9:  52.4299 ms
Run 10: 39.8449 ms
```

1.4 Documentation of performance after optimization

Lets check if our optimizations are indeed correct based on our research and current results.
We start out by once again, running the program program with a profiler.

Figure 1.7: Profiler Flame Chart

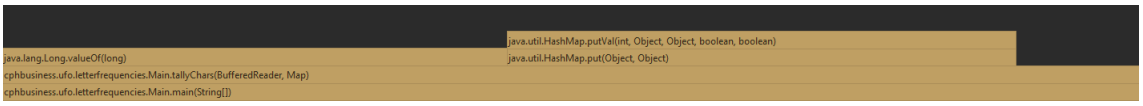


Figure 1.8: Profiler Flame Chart

Just by looking at the profiler, it seems promising that our program will indeed be running less expensive. To backup the profiler, we perform the same calculations as we did before the optimizations:

Optimized data

Count:	10
Sum:	461.7976
Mean:	46.17976
Variance:	201.1248140124
Standard Deviation:	14.181848046443

Now lets compare it to our earlier results.

Unoptimized data

Count:	10
Sum:	820.2117
Mean:	82.02117
Variance:	866.8134585601
Standard Deviation:	29.441695918546

As we can see from our data, small changes to the code can result in HUGE improvements to the execution times, which further reinforces our statement, that it is important to research the methods you're going to use before writing the program.

Our data confirms that we reached an average performance gain of 56.30%, which is greater than our goal of reaching at least 50% optimization.

Bibliography

[1] Nick Zhang. *Java Tip 26: How to improve Java's I/O performance*. 1997. URL: https://www.infoworld.com/article/2077523/java-tip-26--how-to-improve-java-s-i-o-performance.html.