Dynamic Profiling

By: Ethan J. Nephew

Program Overview

The program that I will be profiling is my JavaFX client server currency converter program that I developed last summer. The program is not complete, but it is entirely functional as it is. The user of the program is presented with 2 Java programs but has a total of 3 unique windows.

Server Interface

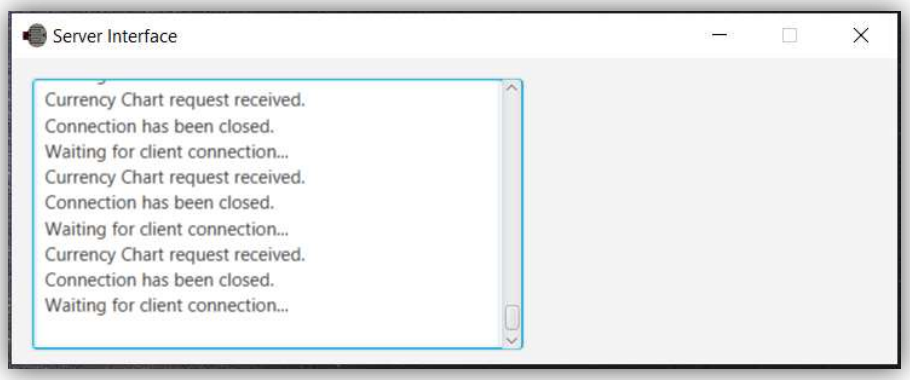

Figure 1 Server Interface. Displays useful information related to client-server interactivity. The white space on the right is used for setting start-up configerations.

The first window is the Sever Interface. This window represents the server side of the application, and it communicates with the client side of the application.

Currency Converter Interface

The other window is the Currency Converter window. This window is used for requesting and displaying useful information related to currency conversions. The user can compare any two of fifty-four currencies. The user can also up the historical Chart Data window that will display currency conversion rates over the past couple years.

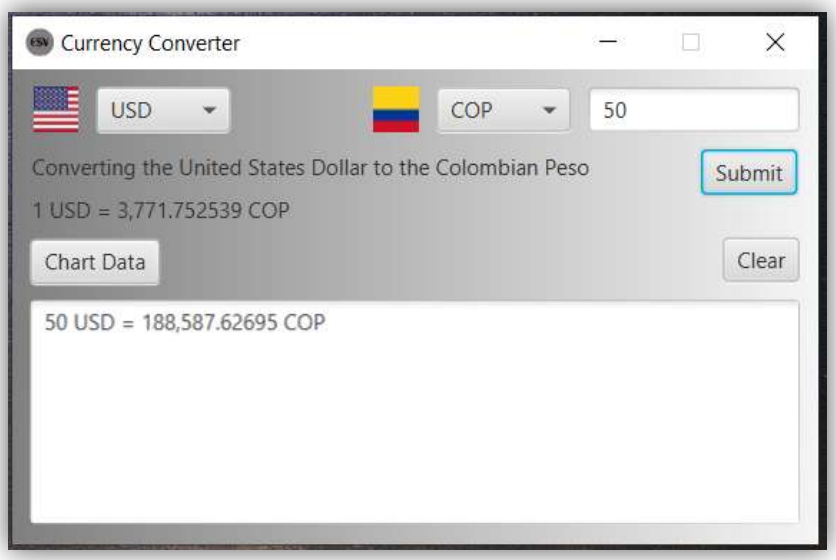

Figure 2 Currency Conversion interface that is used to receive user data and display useful information to the user.

Historical Data Chart Interface

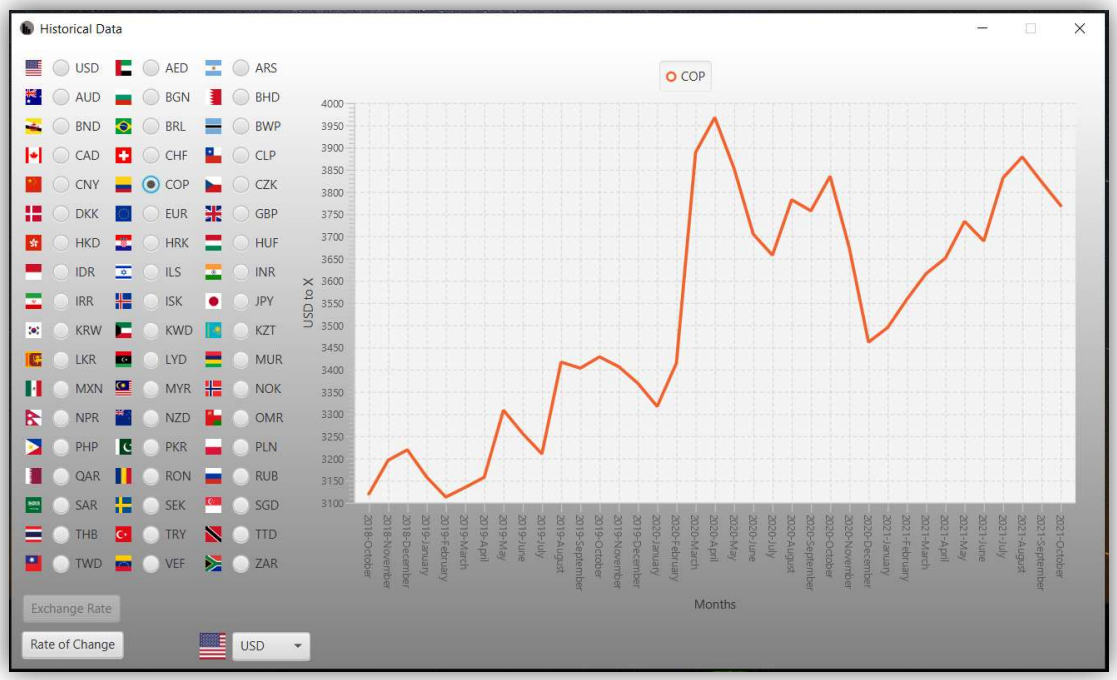

Figure 3 Chart that displays the historical conversion rate. USD is selected as the base currency for comparison.

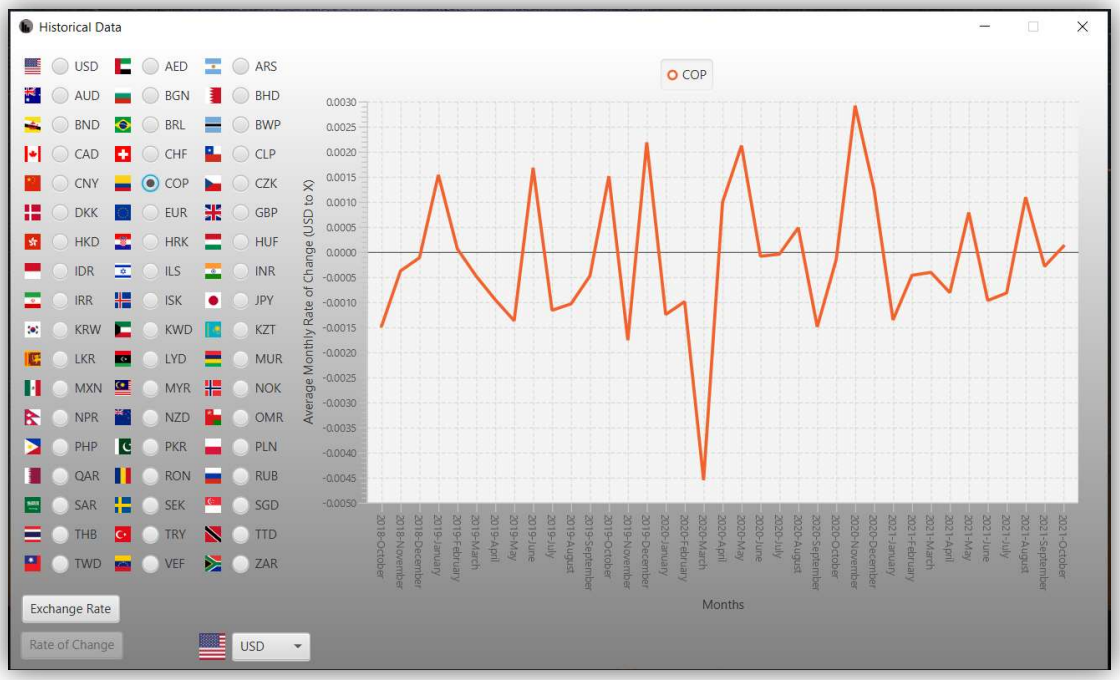

Figure 4 Chart that displays the historical rate of change. USD is selected as the base currency for comparison.

The Chart Data button on the Currency Converter interface from Figure 2 opens a new window that is used to graph historical data from October of 2018 to October of 2021. On the lower left-

hand corner of the Historical Data window, there is an option to use either Exchange Rate or Rate of Change. The exchange rate, in figure 3, is the raw conversion rate between currencies. The rate of change, in figure 4, is useful for determining the volatility of a given currency.

Currency Conversion Visual VM

The heap performance for the client side of the application has a relatively low profile. It remains within the 20-45 MB range, which is respectable. It can also be determined here that a memory leak probably does not exist in this component of the application.

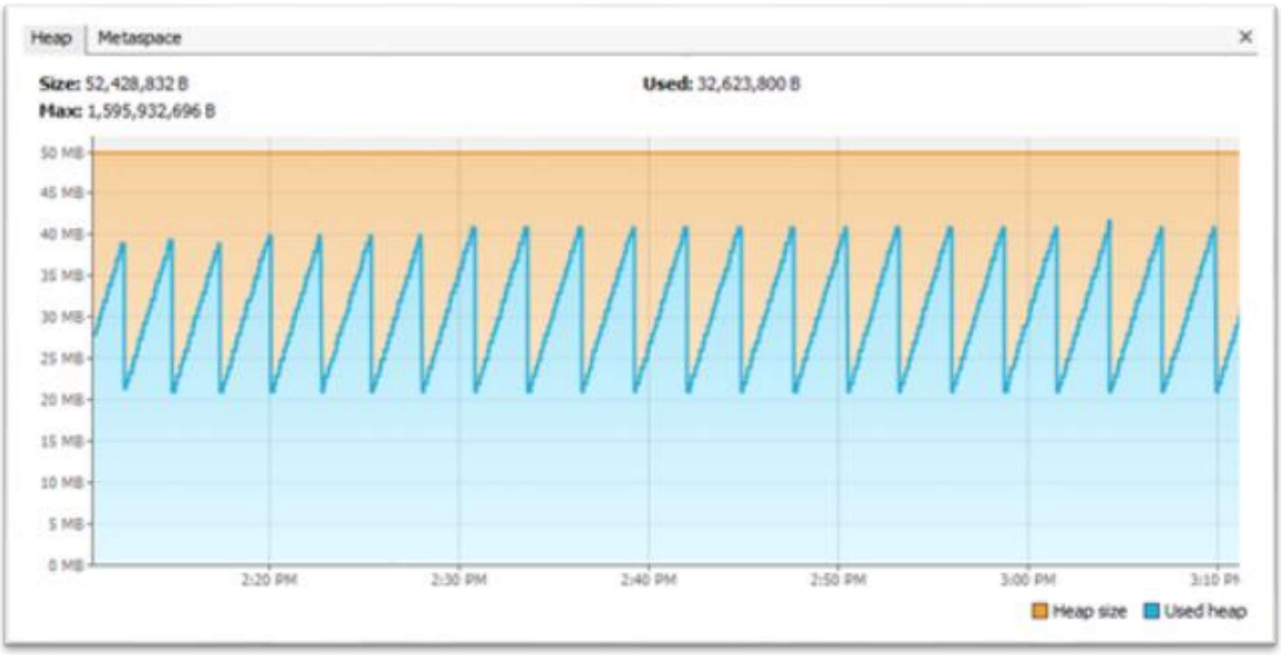

Figure 5 When the Currency Conversion interface is running it appears that there is a sort of memory leak taking place, but I believe this is just responsible garbage collection taking place. This is the profile that is produced when the application is running, but little to no interaction is taking place.

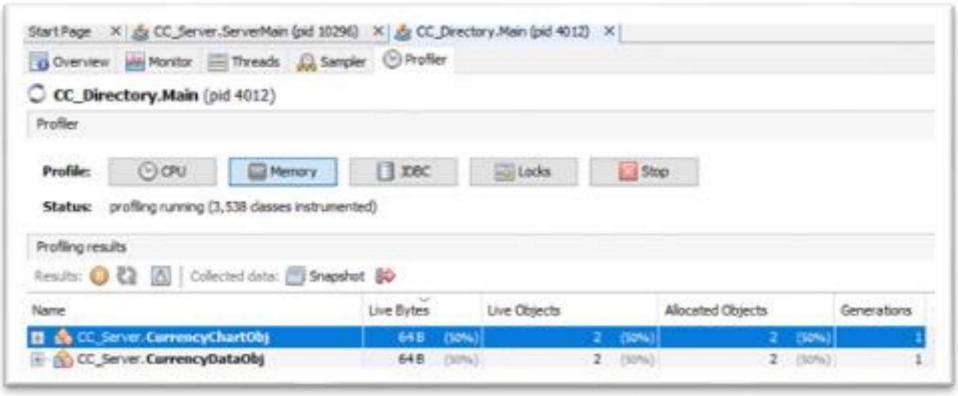

Figure 6 Something interesting is that on the client side, when requesting and receiving data objects, it instantiated two of each type. The overall size of the objects remains consistent throughout the program.

Server Interface Visual VM

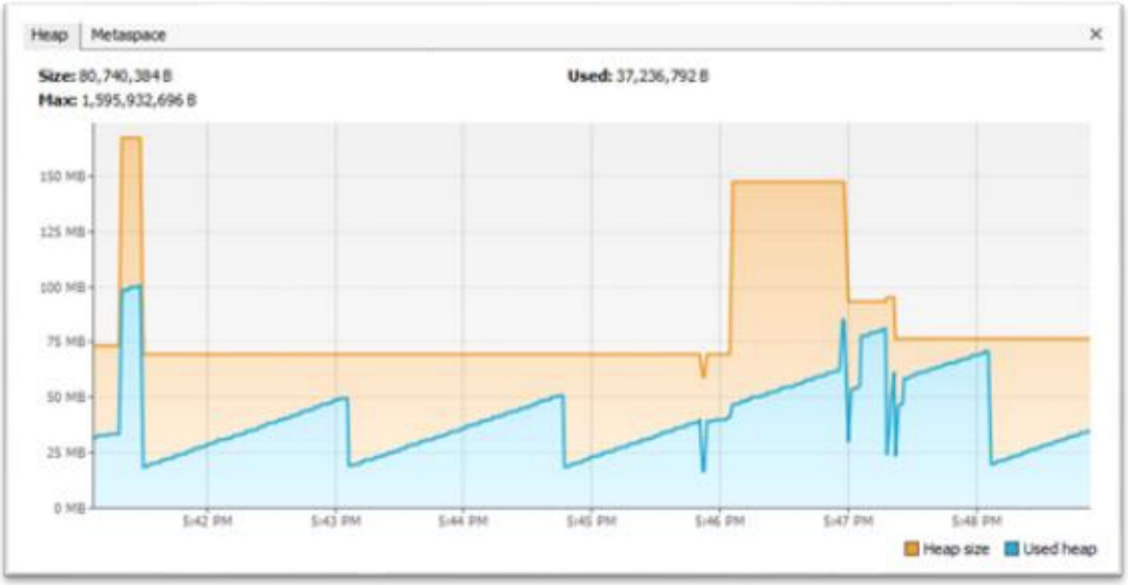

Figure 7 Heap size on the server side can be more turbulent. I think this can be mostly attributed to my use of profiling. The peaks in the graph occurred when I engaged memory profiling. From 5:42 – 5:45 is what could be described as normal operating behavior. Server requests can result in fluctuations in the heap size.

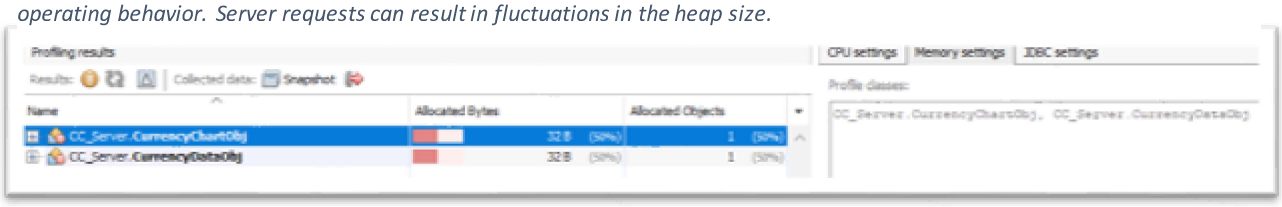

Figure 8 Profiling the memory for the data objects on the server side demonstrates that both currency conversion objects and currency data objects use 32B of space. To complete a request for a specific type only requires a single object to be instantiated. This is a good use of data objects.

From viewing the heap of the server side of the application it can be observed that there is a similar garbage collection trend occuring on this side too. It might be the case that this is a normal JavaFX behavior or that it is the result of VisualVM monitoring a JavaFX application. The screenshot of the heap was taken after engaging in CPU and memory profiling. It can be observed that profiling does significantly increase the required heap size. In terms of user performance, profiling does noticiably reduce response time. The application's response time is noticably slower compared to when profiling is not being implemented. When I was creating this program, I prioritized the quick transfer and depiction of data. It was my goal that the data being transferred from the sever to the client interface would be close to an instant process. As a result, when it takes .15 more seconds, due to VisualVM, it is noticiable.

The Use of Dynamic Profiling

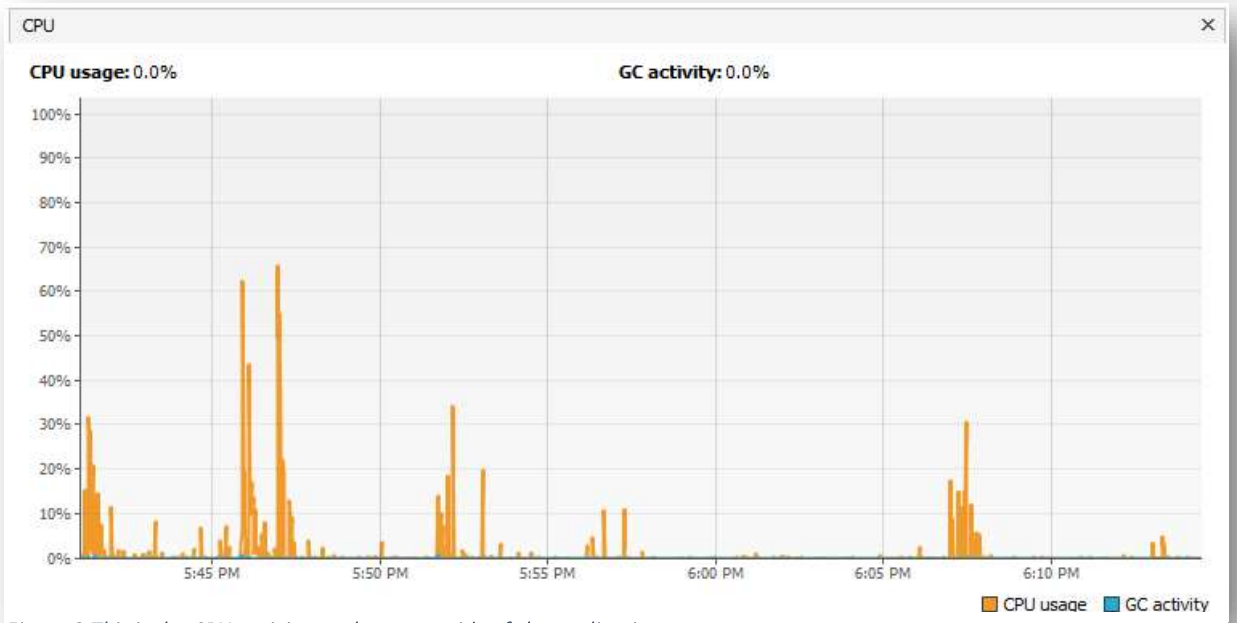

Figure 9 This is the CPU activity on the server side of the application.

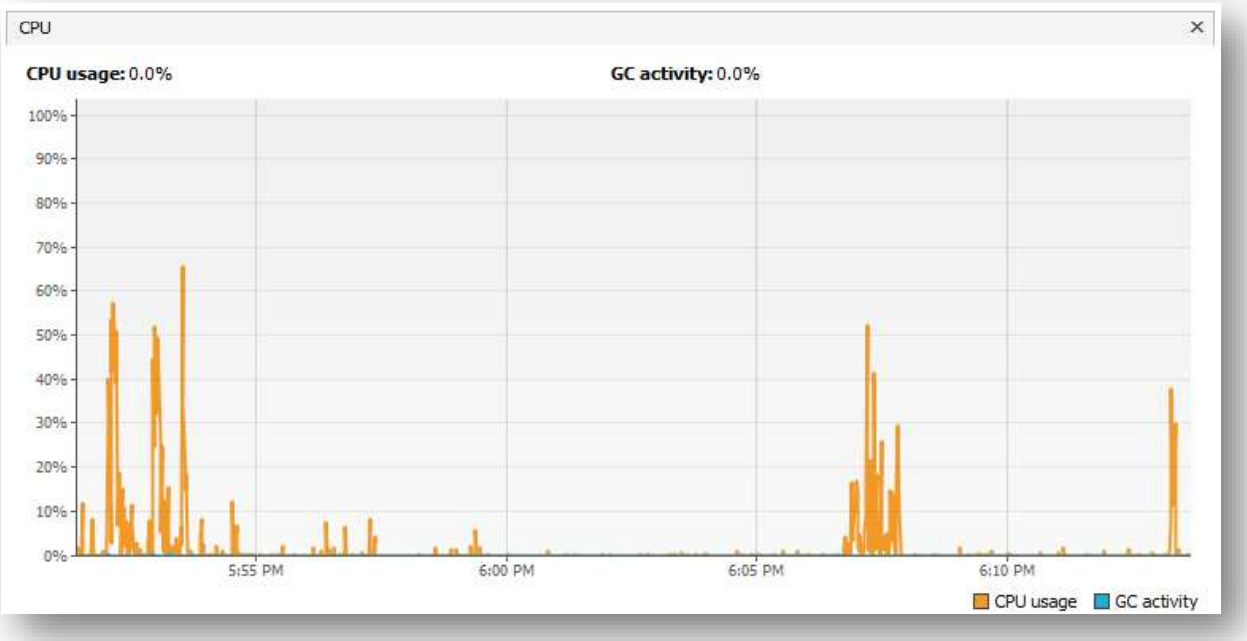

Figure 10 This is the CPU activity on the client side of the application.

CPU activity will vary depending on the system. My computer has 4 CPUs at 2.6 GHz. The last measurements of CPU activity that occurred after 6:10 are the results of serving up chart data. What can be observed is that retrieving the historical charting data isn't particularly CPU intensive compared to what it takes to display that information into a JavaFX chart. I found this to be

particularly surprising. Some of the calculations will be handled database side, because my queries for retrieving this information are rather complex. It could be the case that Visual VM is not measuring the cost of conducting database transactions, because the complex query transactions could be occurring outside of the Java virtual machine.

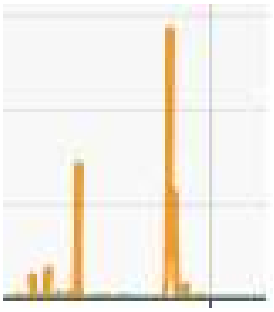

Figure 11 JavaFX Chart can require up to 40% of the CPU to chart a chart data object.

How a data chart object is converted into a chart is through a simple loop. The client side loops through the data chart object and displays each point chronologically. In my opinion, this shouldn't require 40% of my processor's capacity in order to achieve this.

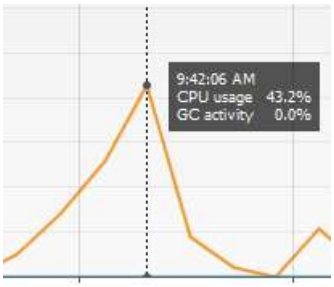

Figure 12 Changing the way elements are added to the chart produced no positive performance on CPU usage.

To see if I could reduce CPU usage, I tried reverse the order by which elements were added to the chart. This produced a reverse ordered chart but required slightly more CPU capacity. I spent some more time experimenting with how elements are added to the chart, but nothing seemed to improve CPU performance. I spent some time investigating performance related mistakes against JavaFX charts. I found a comment where someone was asking why the chart gridlines were inefficient, so I decided to try disabling them.

```
lineChart.setHorizontalGridLinesVisible(false);
lineChart.setVerticalGridLinesVisible(false);
```

Figure 13 Disabled the gridlines in the chart.

To my surprise, disabling the chart lines reduced the CPU load by nearly 15%. This is a significant decrease in my JavaFX charting resource usage. Without gridlines, it is slightly more difficult to read the chart. If this was a real project, there would probably be a debate about whether this is efficiency gain is worth decreasing the charts overall readability. However,

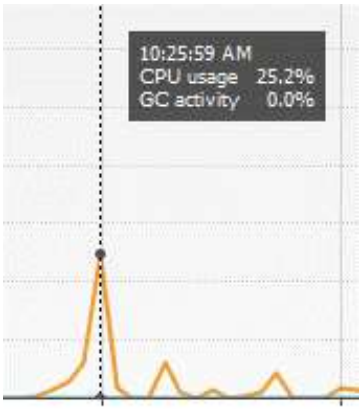

Figure 14 CPU usage has almost a 15% decrease after disabling gridlines in the chart.

this highlights how Visual VM can be used to improve the performance of an application.

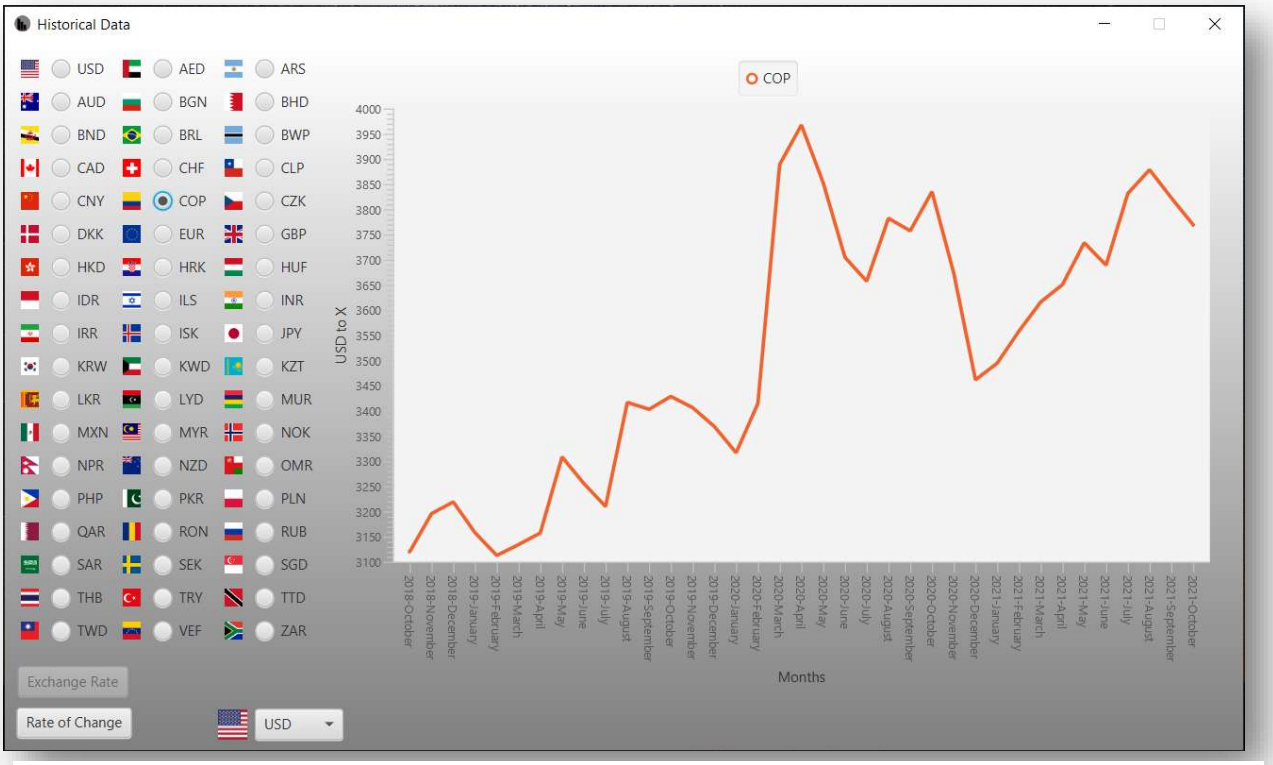

Figure 15 This is how the charting is depicted after removing the lines from the chart. Is the performance gain worth it? Is the readability of the chart significantly decreased? That is something that could be debated.

Closing Statement

Dynamic profiling is used for analyzing programs with precision. It is used to measure memory usage, CPU requirements, method frequency and duration, and object usage. Often programs perform many tasks that produce observable results, but the innerworkings are more difficult to observes. Dynamic profiling is useful for demystifying how a program performs on a system. In my implementation I discovered that populating the JavaFX chart was causing a spike of 40% of the processor capacity on my laptop. It would only do this for a moment, but if my system was already using a significant portion of my processors capabilities, then this could result in the application having problems populating the chart. Absent VisualVM, I probably would have never placed this aspect of my program under such scrutiny. Over hours of tinkering with it I discovered that disabling the gridlines significantly decreased the processor intensity of this component.