Fundamentals of Computing Systems Homework Assignment #1 - EVAL

1. EVAL Problem 1

   (a) **CPU clock Speed using SlEEP Method**

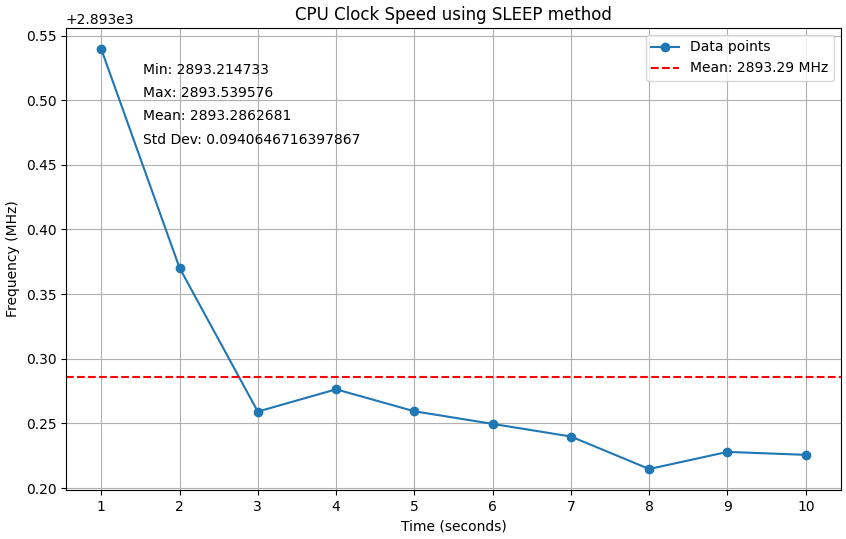

Figure 1: CPU Clock speed using Sleep Method

According to the 10 CPU clock speed measurements I have obtained from Sleep Method, the maximum is 2893.53958MHz, the minimum is 2893.21473MHz, the mean is 2893.28627MHz, and the standard deviation is 0.09406467.

September 19, 2024

Method	MAX	MIN	MEAN	STD
SLEEP	2893.53958MHz	2893.21473MHz	2893.28627MHz	0.09406467

Table 1: Summary of Clock Frequencies for SLEEP Method

(b) **CPU Clock Speed using BUSYWAIT Method**

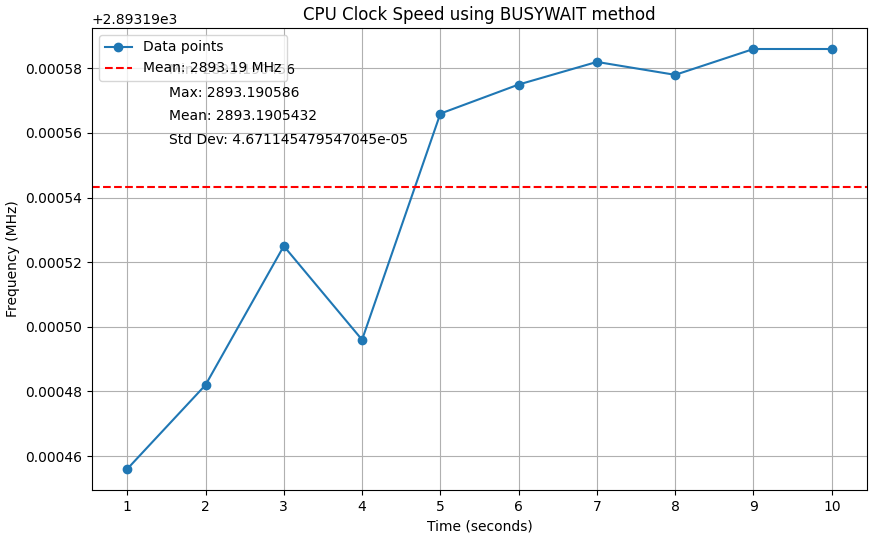

Figure 2: CPU Clock speed using BUSYWAIT Method

According to the 10 CPU clock speed measurements I have obtained from Busy-wait Method, the maximum is 2893.190586MHz, the minimum is 2893.190456MHz, the mean is 2893.1905432MHz, and the standard deviation is 4.671145479547045e-05.

Method	MAX	MIN	MEAN	STD
BUSYWAIT	2893.190586MHz	2893.190456MHz	2893.1905432MHz	0.00004671

Table 2: Summary of Clock Frequencies for BUSYWAIT Method

(c) **Does the SCC always allocate you a physical machine with the same type of CPUs?**

In order to determine whether the SCC allocates me a physical machine with the same CPU type, I conducted 10 clock frequency measurements on Sunday through Thursday between 10am and 6pm. All measurements are in MHz, megahertz. Clock frequency, measured in MHz, indicates the number of cycles a CPU completes per second. A higher clock frequency means the CPU can execute more cycles per second, generally leading to faster performance and the ability to handle more tasks simultaneously. Conversely, a lower clock frequency results in fewer cycles per second, which can make the CPU slower in executing tasks. Clock frequency is a key factor in determining CPU speed and efficiency.

Here are the results

10 Clock Frequency Measurements of 10 seconds each

Date/Time	SLEEP Clock Frequency	BUSYWAIT Clock Frequency
Sunday Morning (10am)	2893.210690MHz	2893.190558MHz
Sunday Afternoon (6pm)	2893.226345MHz	2893.190553MHz
Monday Morning (10am)	2793.471372MHz	2793.434384MHz
Monday Afternoon (6pm)	2599.981557MHz	2599.963650MHz
Tuesday Morning (10am)	2793.459690MHz	2793.421919MHz
Tuesday Afternoon (6pm)	2793.459890MHz	2793.418744MHz
Wednesday Morning (10am)	2793.455472MHz	2793.418386MHz
Wednesday Afternoon (6pm)	2793.457106MHz	2793.419676MHz
Thursday Morning (10am)	2793.434154MHz	2793.415095MHz
Thursday Afternoon (6pm)	2599.978443MHz	2599.959175MHz

Table 3: Clock Frequency Measurements

I visualize the trend of the results using these data from the SLEEP and BUSYWAIT methods.

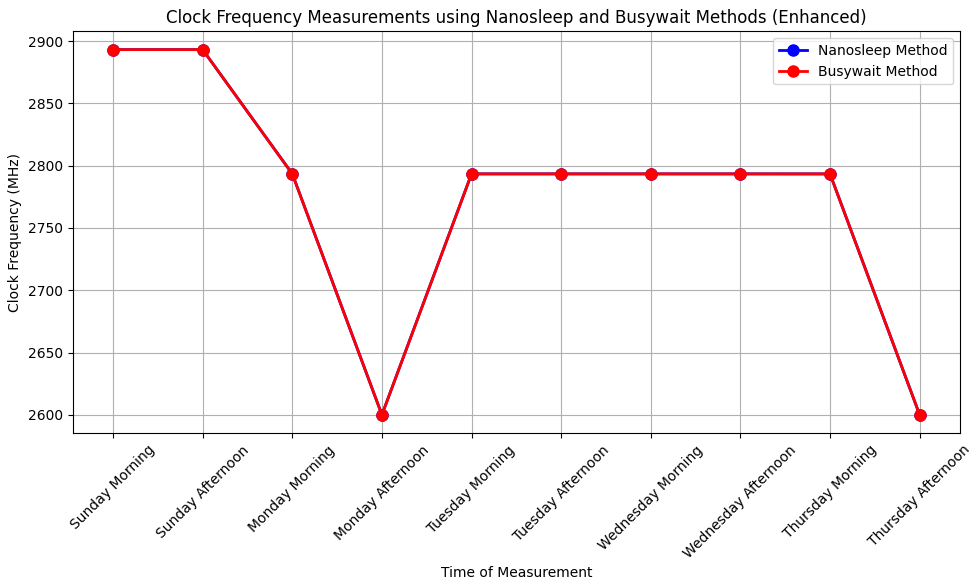

Figure 3: CPU Clock speed Measurements

When I examine the graph, although I plotted the SLEEP and BUSYWAIT methods together, only the red lines representing the BUSYWAIT method are visible. This is because the SLEEP method data is nearly identical and hidden beneath the BUSYWAIT method. To confirm this, I plotted the correlation between the two methods to verify their similar trends and how closely aligned the data is.

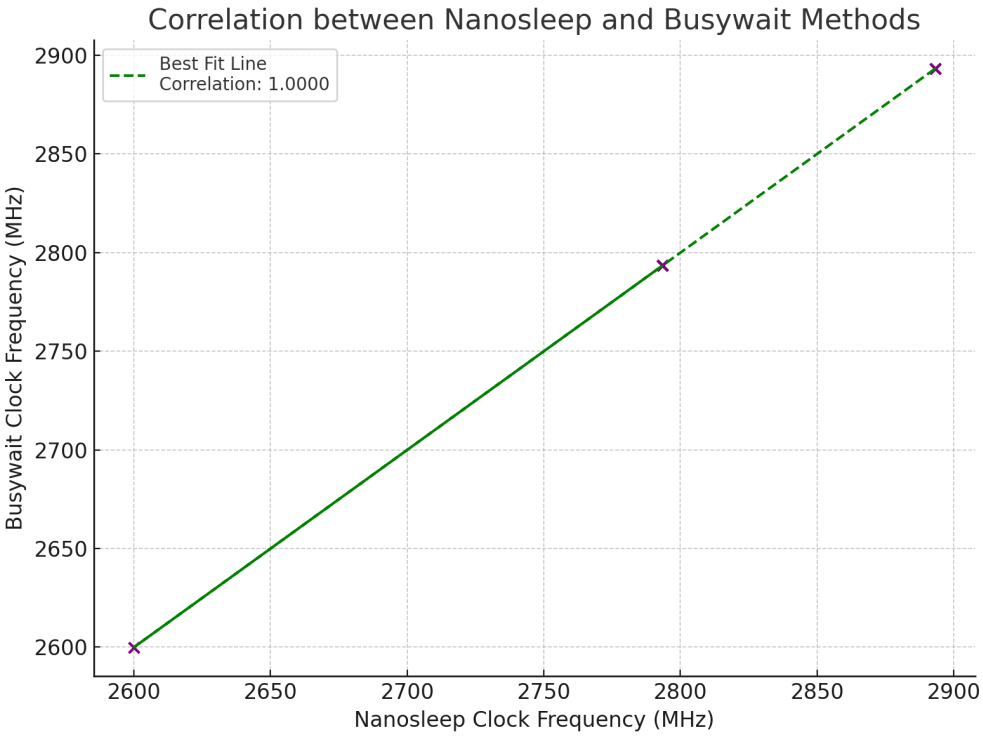

Figure 4: CPU Clock speed Measurements

When analyzing the data, I found the correlation to be 0.9999999962471202, which is almost 1. This indicates that the for each Date/Time clock frequencies for the BUSYWAIT and SLEEP methods are almost identical. Thus, I can conclude that each process or execution of the program uses the same CPU in my assigned SCC session. Likewise, when I was first assigned a CPU to a SCC session, it does not change during my session time. Given this, it would have been sufficient to use only one method and compare it over time and days. Let's now compare CPU frequencies of different SCC sessions by date and time.

Back to figure 3, I can see the result trend by each days like this.

Sunday Observations: On Sunday morning, both the Nanosleep and Busywait methods reach clock frequencies well over 2900 MHz. The reason for this is likely due to the fact that SCC usage is lower on Sundays, resulting in each user having more CPU power, which leads to higher clock speeds.

Monday Observations: On Monday, both methods showed a significant drop in clock frequency, especially in the afternoon (2599 MHz for nanosleep, 2599 MHz for busywait). This suggests increased SCC usage, likely due to students and researchers, which reduces CPU resources per session.

Tuesday Observations: Clock frequencies rebound slightly on Tuesday, but remain below the levels seen on Sunday. This moderate increase suggests that while the SCC may still be busy, the usage load is lighter compared to Monday, allowing for a partial recovery in clock frequency allocation.

Wednesday Observations: On Wednesday, the clock frequency remained stable at around 2800 MHz. SCC usage was consistent across the morning and afternoon periods, with fewer fluctuations than on Monday and Tuesday. Steady levels may reflect a balanced workload.

Thursday Observations: Similar to Monday, Thursday morning and afternoon show a significant decline in clock frequency, with values dropping again toward 2600 MHz in the afternoon. This may suggest that towards the end of the week, SCC experiences a peak in usage, likely as deadlines for research or assignments approach.

The overall conclusion is that **SCC does not allocate physical machines with identical CPUs**. Observations show that Sundays have the highest CPU clock frequencies, indicating that with fewer users on Sundays, the system can provide more CPU power per user. In contrast, Mondays have the lowest CPU frequencies. From my undergraduate research experience, my PhD advisor mentioned that he often started his SCC sessions on Mondays for training machine learning models that required several days. While this is just one instance, it suggests that many users, including CS350 students with assignments, also start their work on Mondays and finishing up on Thursday. This high user demand results in Mondays and Thursday having the most users competing for CPU resources.

2. EVAL Problem 2

(a) **From the server output, compute the average throughput of the server**

In order to evaluate the question, I ran the below code to save my server output.

Listing 1: bash version

```
./build/server 2222 > server_result.txt 2>&1
```

I used port number 2222 to communicate with client program. This is what server_result.txt looks like.

```
INFO: setting server port as: 2222
INFO: Waiting for incoming connection...
R0:2910317.279398,0.041792,2910317.279455,2910317.321275
R1:2910317.462815,0.133472,2910317.462854,2910317.596357
.
.
.
R497:2910367.885143,0.014006,2910368.465292,2910368.479308
R498:2910368.067176,0.032612,2910368.479312,2910368.511936
R499:2910368.179173,0.063366,2910368.511942,2910368.575323
```

From request 0 to request 499, there are total 500 response made by the server. The response format is like this :
R<request ID>: <sent timestamp>, <request length>, <receipt timestamp>, <completion timestamp>

<request ID>: the id # of request
<sent timestamp>: timestamp at which the request was sent by the client
<request length>: length of the request as sent by the client
<receipt timestamp>: timestamp at which the server received the request
<completion timestamp>: timestamp at which the server completed processing of the request and sent a response back to the client

Throughput is calculated by dividing the total number of requests by the total time. The formula is

$$\text{Throughput} = \frac{N}{T}$$

N is the total number of requests, T is the total time in seconds.

The total number of requests are 500. I need to calculate the total time. The total time is when the server got the first request from the client to when the server response to the last request of the client. So I have to look at <receipt timestamp>of R0 and <completion timestamp>of R499. Which is 2910317.279398 seconds and 2910368.575323 seconds.

$$\text{Total Time} = 2910368.575323sec - 2910317.279455sec = 51.295868sec$$

So, the Throughput is

$$\frac{500\ requests}{51.295868\ sec} = 9.74737380407\ req/sec$$

(b) **Compute the utilization of the server as a percentage**

Based on the result output of server from part a), let's calculate the utilization. **Utilization** is a fraction of time over a given time window during which the response is busy (not idle). The formula is

$$\text{Utilization} = \frac{\text{Time Resource is busy}}{\text{Total Available Time}}$$

In essence, utilization refers to the total time minus idle time. Here, idle time is the period during which the server wait for the client to send the new request after it sent the response—the time between sending a request and receiving a response.

From the lecture, Professor mentioned that in this assignment case, we can calculate the run time of CPU as the length of the request since the CPU's execution is waiting for length amount of BUSYWAIT. So let's slit this question into two different method.

First, take the summation of all 500 request's length as a run time. Second, for all 500 requests, subrtract receipt timestamp from completion timestamp to calculate each request's run time and add all.

For the first method, in order to add all 500 request's length, I implemented the below python code to add all 2nd column which is the length request time from client.

Listing 2: Summing second timestamps from a file

```
# Open the server result file and read the content
with open('server_result.txt', 'r') as file:
    lines = file.readlines()

# Initialize a total seconds to hold the sum
total_seconds = 0.0

# Loop through each line
for line in lines:
    # Split the line by ',' to extract the second timestamp (
       second value)
    parts = line.split(',')

    # Extract the second timestamp and convert to float
    second_timestamp = float(parts[1])

    # Add to the total
    total_seconds += second_timestamp

# Print the total sum
print(f"Total sum of second timestamps: {total_seconds}")
```

From this code I got the total time of length requests are **40.66409700000005** sec. From previous part a), we know the total time is 51.295925 sec, so the utilization is

$$\frac{40.66409700000005\ sec}{51.295868\ sec} = 0.79273630772\ percent$$

Now calculate the second method and compare. In order to subrtract receipt timestamp from completion timestamp and add all I change the previous code to as follow:

Listing 3: Summing second timestamps from a file

```
# Open the file and read the content
with open('server_result.txt', 'r') as file:  # Replace '
   timestamps.txt' with your file name
    lines = file.readlines()

# Initialize a variable to hold the sum
total_seconds = 0.0

# Loop through each line
```

```
for line in lines:
    # Split the line by commas to extract the second timestamp (
       second value)
    parts = line.split(',')

    # Extract the completion and recipt timestamp and subtract
    second_timestamp = float(parts[3]) - float(parts[2])

    # Add to the total
    total_seconds += second_timestamp

# Print the total sum
print(f"Total sum of second timestamps: {total_seconds}")
```

The output is **40.67286800639704 sec**. Compare to the first method, there is a little differece which is 40.67286800639704 sec - 40.66409700000005 = 0.00877100639 sec. I think this difference is the amount of time the server had to handle received the request message to find out the length time, prepare response message struct, and measure clock time etc. It is amazaing that only 0.00877100639 sec used to handle 500 request other than busywait.

So for the second method, the utilization is

$$\frac{40.67286800639704\ sec}{51.295868\ sec} = 0.79290729628\ percent$$

(c) "Percent of CPU this job got:"

After I ran the server and client at the same time, my **time** utility print as following:

Listing 4: bash version

```
  Command being timed: "./server 2222"
        User time (seconds): 40.58
        System time (seconds): 0.01
        Percent of CPU this job got: 79%
        Elapsed (wall clock) time (h:mm:ss[jhonglee@scc-yf3 build
           ]$  or m:ss): 0:51.31
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 1964
        Average resident set size (kbytes): 0
```

```
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 77
        Voluntary context switches: 126
        Involuntary context switches: 17
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

My Percent of CPU this job got is 79% which actually match my Utilization! Also interestingly, the user time is 40.58 sec which is really close to my run time from part b).

(d) Change -a parameter and repeat 12 times

After I ran the the server-client changing -a parameter 1 through 12 and calculating utilization using above python codes are as below:

Utilization of changing -a parameter

-a parameter	Runtime	Total Time	Utilization
1	40.679353 sec	508.971805 sec	7.992457068 %
2	40.67832701 sec	254.524561 sec	15.98208316 %
3	40.66885401 sec	169.697587 sec	23.96548751 %
4	40.668798 sec	127.289754 sec	31.94978129 %
5	40.668411 sec	101.84515 sec	39.93161284 %
6	40.66830499 sec	84.881975 sec	47.91159135 %
7	40.667744 sec	72.765408 sec	55.88884212 %
8	40.66772199 sec	63.749471 sec	63.79303445 %
9	40.667197 sec	56.806531 sec	71.58894635 %
10	40.666622 sec	51.288346 sec	79.29018026 %
11	40.666131 sec	46.773054 sec	86.9435017 %
12	40.665921 sec	43.010437 sec	94.54896029 %

Table 4: Clock Frequency Measurements

I can plot the utilization and -a parameter and get the correlation between two values.

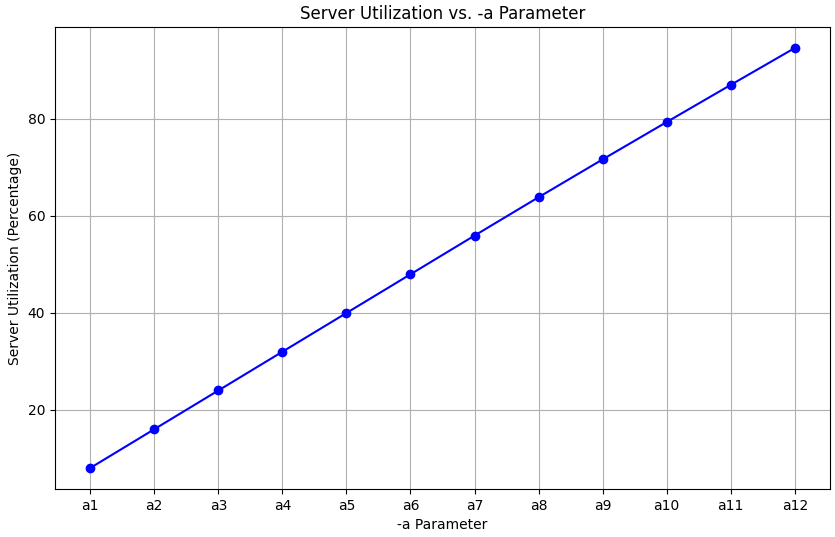

Figure 5: Utilization and -a parameter plot

The correlation between utilization and -a paramter is **0.999962084106445**. The correlation of two variables are close to 1, which is a extremely strong positive linear relationship between the -a parameter and server utilization.

Because the value is very close to 1, it shows that as the -a parameter increases, server utilization also increases. The correlation is almost identical, implying that an increase in -a increases utilization directly and predictably. Almost all utilization variability up to -a 12 can be explained by changes in -a.

(e) Max, Min, Mean, STD of Response Time with -a 10 server

The response time of each request is how long does the server take to process clients' requests and send back the response. For each server response time, I need to calculate (Completion timestamp - send timestamp). Professor metioned in the class that for this question, the response time is between when client sends the messages and until client receives the response. This is each request's response time, so I need to calculate MAX, MIN, MEAN, STD DEVIATION for 500 requests.

The Stats are as follows:

-a	MAX	MIN	MEAN	STD
10	1.446328 sec	0.000377 sec	0.3287245159978047 sec	0.314179

Table 5: Statistics of -a : 10

(f) Repeat the average response time calculation

Repeat the Part e method for all -a 1 $\tilde{1}2$, and I got the result as follows:

-a	MAX	MIN	MEAN	STD
1	0.516773 sec	0.0004277 sec	0.08694804399926215 sec	0.082452
2	0.533892 sec	0.000379 sec	0.09574253800325096 sec	0.093711
3	0.568277 sec	0.000376 sec	0.10481401402223856 sec	0.103962
4	0.598291 sec	0.000391 sec	0.11513752001337707 sec	0.114635
5	0.638486 sec	0.000324 sec	0.12865784599632024 sec	0.126950
6	0.684174 sec	0.000378 sec	0.1452020660014823 sec	0.142459
7	0.741363 sec	0.000377 sec	0.16844792199973016 sec	0.163084
8	0.784295 sec	0.000390 sec	0.1977727539902553 sec	0.187589
9	0.914748 sec	0.000380 sec	0.25022777199279517 sec	0.229068
10	1.446328 sec	0.000377 sec	0.3287245159978047 sec	0.314179
11	1.920860 sec	0.000321 sec	0.44226453801058235 sec	0.400245
12	2.336336 sec	0.024530 sec	1.118266559992917 sec	0.624193

Table 6: Statistics of -a : 10

With the average response time of each arrival time (-a) parameter, I can make the plot between Utilization and the average response time.

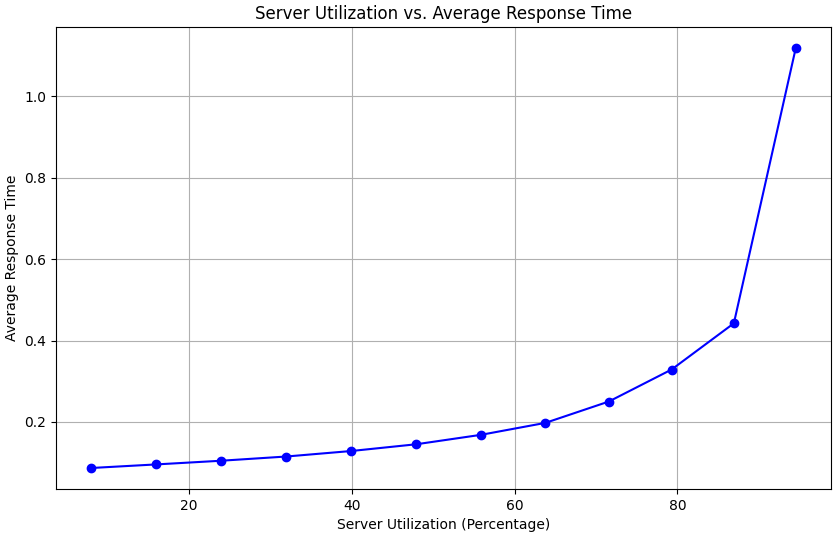

Figure 6: Utilization and -a parameter plot

This graph shows the relationship between average response time and utilization percentage, where the plot demonstrates an exponential correlation between the two. From the plot, I learned that a lower arrival rate means clients send requests at a wider intervals. This gives the CPU more idle time while waiting for new requests, resulting in a shorter queue and faster response times. On the other hand, a high utilization percentage indicates that the CPU is consistently busy, as requests arrive at shorter intervals, which also means higher arrival rate (-a values is higher). This leads to a longer average response time for each request. As seen in Table 6, when the arrival rate is faster (-a 12), the average response time increases, confirming the exponential relationship between utilization and response time.