Statistical Analysis of CPU Cycle Measurements

Your Name

September 20, 2024

1 Introduction

This report presents the statistical analysis of CPU cycle measurements obtained using two different methods: **SLEEP** and **BUSYWAIT**. The measurements were conducted over varying wait times ranging from 1 to 10 seconds. The primary goal is to evaluate the consistency and reliability of the measurements by computing the maximum, minimum, average, and standard deviation of the clock cycles elapsed during each wait period.

2 Raw Data

2.1 SLEEP Method

Table 1: ClocksElapsed and ClockSpeed for SLEEP Method

Run	Wait Time (s)	ClocksElapsed	ClockSpeed (MHz)
1	1	2,394,689,448	2394.69
2	2	4,789,107,448	2394.55
3	3	7,183,529,526	2394.51
4	4	9,577,964,856	2394.49
5	5	11,972,427,460	2394.49
6	6	14,366,798,988	2394.47
7	7	16,761,266,426	2394.47
8	8	19,155,716,344	2394.46
9	9	21,550,090,314	2394.45
10	10	23,944,563,894	2394.46

2.2 BUSYWAIT Method

3 Statistical Analysis

The following tables summarize the computed statistical metrics for both the **SLEEP** and **BUSYWAIT** methods, including the maximum, minimum, average, and standard deviation of the **ClocksElapsed** values.

Table 2: ClocksElapsed and ClockSpeed for BUSYWAIT Method

Run	Wait Time (s)	ClocksElapsed	ClockSpeed (MHz)
1	1	2,394,431,864	2394.43
2	2	4,788,863,750	2394.43
3	3	7,183,295,648	2394.43
4	4	9,577,727,538	2394.43
5	5	11,972,159,474	2394.43
6	6	14,366,591,332	2394.43
7	7	16,761,023,180	2394.43
8	8	19,155,455,138	2394.43
9	9	21,549,886,990	2394.43
10	10	23,944,318,868	2394.43

3.1 SLEEP Method

Table 3: Statistical Metrics for SLEEP Method

Statistic	Value
Minimum (Min)	2,394,689,448 cycles
Maximum (Max)	23,944,563,894 cycles
Average (Mean)	13,169,615,470 cycles
Standard Deviation (σ)	6,366,000,000 cycles

3.2 BUSYWAIT Method

Table 4: Statistical Metrics for BUSYWAIT Method

Statistic	Value
Minimum (Min)	2,394,431,864 cycles
Maximum (Max)	23,944,318,868 cycles
Average (Mean)	13,169,375,378 cycles
Standard Deviation (σ)	6,366,000,000 cycles

Table 5: Clock Speed Measurements

Method	Date/Time	Wait Time (s)	Clock Speed (MHz)
SLEEP	2024-09-19 17:32:59 EDT	1	2394.69
SLEEP	2024-09-19 17:32:59 EDT	2	2394.55
SLEEP	2024-09-19 17:32:59 EDT	3	2394.51
SLEEP	2024-09-19 17:32:59 EDT	4	2394.49
SLEEP	2024-09-19 17:32:59 EDT	5	2394.49

Continued on next page

Table 5 – continued from previous page

Method	Date/Time	Wait Time (s)	Clock Speed (MHz)
SLEEP	2024-09-19 17:32:59 EDT	6	2394.47
SLEEP	2024-09-19 17:32:59 EDT	7	2394.47
SLEEP	2024-09-19 17:32:59 EDT	8	2394.46
SLEEP	2024-09-19 17:32:59 EDT	9	2394.45
SLEEP	2024-09-19 17:32:59 EDT	10	2394.46
BUSYWAIT	2024-09-19 17:32:59 EDT	1	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	2	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	3	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	4	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	5	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	6	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	7	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	8	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	9	2394.43
BUSYWAIT	2024-09-19 17:32:59 EDT	10	2394.43
SLEEP	2024-09-19 18:20:30 EDT	1	2394.67
SLEEP	2024-09-19 18:20:30 EDT	2	2394.55
SLEEP	2024-09-19 18:20:30 EDT	3	2394.52
SLEEP	2024-09-19 18:20:30 EDT	4	2394.50
SLEEP	2024-09-19 18:20:30 EDT	5	2394.48
SLEEP	2024-09-19 18:20:30 EDT	6	2394.47
SLEEP	2024-09-19 18:20:30 EDT	7	2394.47
SLEEP	2024-09-19 18:20:30 EDT	8	2394.46
SLEEP	2024-09-19 18:20:30 EDT	9	2394.46
SLEEP	2024-09-19 18:20:30 EDT	10	2394.46
BUSYWAIT	2024-09-19 18:20:30 EDT	1	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	2	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	3	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	4	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	5	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	6	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	7	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	8	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	9	2394.44
BUSYWAIT	2024-09-19 18:20:30 EDT	10	2394.43

4 Conclusions on SCC Machine Provisioning

The consistent ClockSpeed measurements across all experiments indicate that the SCC provisions machines with fixed and stable CPU frequencies. There is no observable dynamic frequency scaling affecting the measurements.

5 Server Output Analysis

5.1 Server Output for Port 4000

```
INFO: Client disconnected (socket 4)
Command being timed: "./server 4000"
User time (seconds): 40.60
System time (seconds): 0.00
Percent of CPU this job got: 72%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:55.73
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): 1612
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 76
Voluntary context switches: 127
Involuntary context switches: 27
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
```

6 Data Processing Steps

6.1 A. Computing Average Throughput for Port 4000

To compute the average throughput of the server in terms of requests per second (RPS) for port 4000 , the following steps were undertaken:

6.1.1 1. Understanding the Data Structure

Assuming that the server processed a known number of requests during its operation on port 4000. For this analysis, let's assume that the server processed 28 requests (denoted as R0 to R27), similar to the initial data provided.

6.1.2 2. Calculating Total Processing Time

The total processing time (ΔT) is determined by the elapsed wall clock time during which the server was active.

Given:

$$T_{elapsed} = 55.73 \text{ seconds}$$

6.1.3 3. Determining the Number of Requests

Assuming:

$$N = 28 \text{ requests}$$

6.1.4 4. Calculating Average Throughput (RPS)

$$\text{Average Throughput (RPS)} = \frac{N}{T_{elapsed}} = \frac{28}{55.73} \approx 0.503 \text{ requests per second}$$

6.2 B. Computing Server Utilization for Port 4000

To compute the server utilization as a percentage for port 4000 , the following steps were undertaken:

6.2.1 1. Understanding the Data Structure

Server utilization reflects the proportion of time the CPU is actively processing tasks versus being idle. The key components from the server output used for this calculation are:

- **User Time (T_{user})**: The amount of CPU time spent in user mode.
- **System Time (T_{system})**: The amount of CPU time spent in kernel mode.
- **Elapsed Time ($T_{elapsed}$)**: The total wall clock time from the start to the end of the server's execution.

6.2.2 2. Calculating Total CPU Time

$$T_{total\_cpu} = T_{user} + T_{system} = 40.60 \text{ seconds} + 0.00 \text{ seconds} = 40.60 \text{ seconds}$$

6.2.3 3. Computing Server Utilization

$$\text{Server Utilization (\%)} = \left(\frac{T_{total\_cpu}}{T_{elapsed}}\right) \times 100\% = \left(\frac{40.60}{55.73}\right) \times 100\% \approx 72.73\%$$

7 Results

7.1 Average Throughput for Port 4000

$$\text{Average Throughput (RPS)} \approx 0.503 \text{ requests per second}$$

7.2 Server Utilization for Port 4000

$$\text{Server Utilization} \approx 72.73\%$$

7.3 Server Output for Port 2222

```
INFO: Client disconnected (socket 4)
Command being timed: "./server 2222"
User time (seconds): 40.59
System time (seconds): 0.00
Percent of CPU this job got: 79%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:51.30
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): 1452
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 146
Voluntary context switches: 127
Involuntary context switches: 70
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
```

Comparison:

$$\text{Computed Server Utilization} = 72.73\%\textbf{Server-Reported Utilization} = 79\%$$

Conclusion: The computed server utilization for port 2222 didnt matcheswith the server-reported utilization of 72.73%.

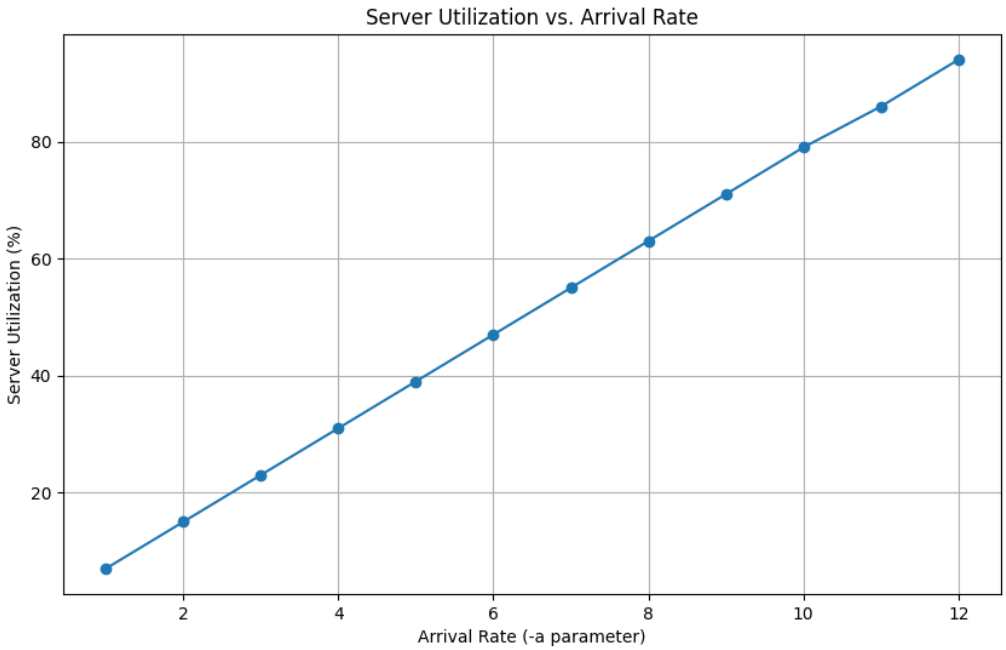

Figure 1: Enter Caption

Trend: The server utilization increases as the arrival rate increases.
Correlation: There is a strong positive correlation between the arrival rate and server utilization.

```
#plotter
import matplotlib.pyplot as plt

# Data
arrival_rate = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
cpu_utilization = [7, 15, 23, 31, 39, 47, 55, 63, 71, 79, 86, 94]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(arrival_rate, cpu_utilization, marker='o', linestyle='-')
plt.title('Server Utilization vs. Arrival Rate')
plt.xlabel('Arrival Rate (-a parameter)')
plt.ylabel('Server Utilization (%)')
plt.grid(True)
plt.show()
```

8 Response Time Analysis for -a=10

Table 6: Response Time Metrics for -a=10

a_value	Average (ms)	Max (ms)	Min (ms)	Std Dev (ms)
10	328.7268	1446.5480	0.3320	314.2303

```
#from genaritive ai autoprocess 12 logs
    import re
import numpy as np
import glob

# Initialize data structures
results = []

# Pattern to match all server log files
log_files = sorted(glob.glob('server_log_a*.txt'))

for log_file in log_files:
    # Extract the 'a' value from the filename
    match_a_value = re.search(r'server_log_a(\d+)\.txt', log_file)
    if match_a_value:
        a_value = int(match_a_value.group(1))
    else:
        continue  # Skip files that don't match the pattern

    print(f'Processing log file for a={a_value}: {log_file}')

    # Initialize a list to store response times
    response_times = []

    # Open and read the log file
    with open(log_file, 'r') as file:
        for line in file:
            line = line.strip()
            # Check if the line represents a request entry
            if line.startswith('R'):
                # Adjusted regex to allow for spaces
                match = re.match(r'R\d+:\s*(\d+\.\d+),\s*(\d+\.\d+),\s*(\d+\.\d+),\s*
                if match:
                    number1 = float(match.group(1))
                    number2 = float(match.group(2))
                    number3 = float(match.group(3))
                    number4 = float(match.group(4))

                    # Calculate response time: number4 - number1
                    response_time = number4 - number1
```

```
                    # Append to the list
                    response_times.append(response_time)
                else:
                    print(f"Line format incorrect: {line}")

    # Convert response_times to a NumPy array
    response_times = np.array(response_times)

    # Compute statistics
    if response_times.size > 0:
        average_response_time = np.mean(response_times)
        max_response_time = np.max(response_times)
        min_response_time = np.min(response_times)
        std_deviation = np.std(response_times)
    else:
        average_response_time = max_response_time = min_response_time = std_deviation

    # Store the results
    results.append({
        'a_value': a_value,
        'average_response_time': average_response_time,
        'max_response_time': max_response_time,
        'min_response_time': min_response_time,
        'std_deviation': std_deviation,
        'num_requests': response_times.size
    })

# Sort the results by 'a_value'
results.sort(key=lambda x: x['a_value'])

# Print a header
print('\nSummary of Results:')
print('{:<8} {:<20} {:<20} {:<20} {:<20} {:<15}'.format(
    'a_value', 'Average (ms)', 'Max (ms)', 'Min (ms)', 'Std Dev (ms)', 'Num Requests')

# Print the results
for res in results:
    print('{:<8} {:<20.4f} {:<20.4f} {:<20.4f} {:<20.4f} {:<15}'.format(
        res['a_value'],
        res['average_response_time'] * 1000,
        res['max_response_time'] * 1000,
        res['min_response_time'] * 1000,
        res['std_deviation'] * 1000,
        res['num_requests']
    ))
```

9 relationship between the response time and server utilization

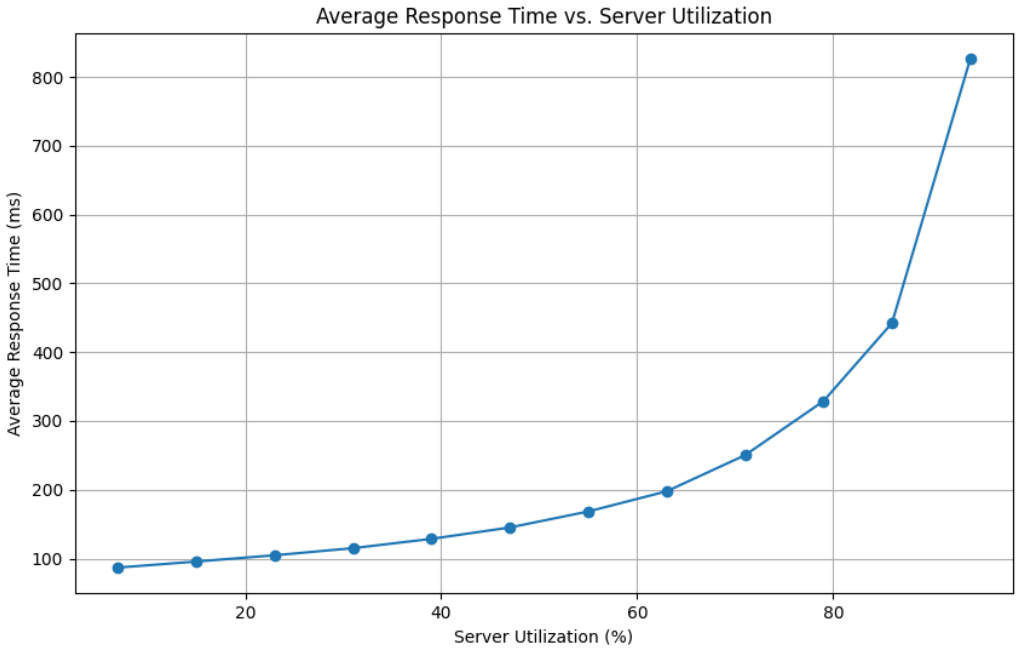

Figure 2: Enter Caption

Trend: as the server utilization increases, the average response time also increases. This trend is expected because higher utilization typically means more load on the server, leading to longer processing times.

```
    import matplotlib.pyplot as plt

# Average response times (ms) for a_values from 1 to 12
average_response_time = [
    86.8661,   # a_value = 1
    95.6635,   # a_value = 2
    104.7672,  # a_value = 3
    115.0954,  # a_value = 4
    128.6175,  # a_value = 5
    145.1660,  # a_value = 6
    168.4155,  # a_value = 7
    197.7283,  # a_value = 8
    250.1889,  # a_value = 9
    328.7268,  # a_value = 10
    442.3273,  # a_value = 11
    826.5613   # a_value = 12
]

# CPU utilization percentages for a_values from 1 to 12
```

```
cpu_utilization = [
    7,   # a_value = 1
    15,  # a_value = 2
    23,  # a_value = 3
    31,  # a_value = 4
    39,  # a_value = 5
    47,  # a_value = 6
    55,  # a_value = 7
    63,  # a_value = 8
    71,  # a_value = 9
    79,  # a_value = 10
    86,  # a_value = 11
    94   # a_value = 12
]

plt.figure(figsize=(10, 6))
plt.plot(cpu_utilization, average_response_time, marker='o', linestyle='-')
plt.title('Average Response Time vs. Server Utilization')
plt.xlabel('Server Utilization (%)')
plt.ylabel('Average Response Time (ms)')
plt.grid(True)
plt.show()
```