**Name- Dhyey Findoriya** **Comp Arch Lab-1**
**Roll No.- B22EE024**

---

## Machine-1 Specifications:
## OS-Fedora Linux 39

| | |
|---|---|
| Architecture: | x86_64 |
| CPU op-mode(s): | 32-bit, 64-bit |
| Address sizes: | 39 bits physical, 48 bits virtual |
| Byte Order: | Little Endian |
| CPU(s): | 8 |
| On-line CPU(s) list: | 0-7 |
| Vendor ID: | GenuineIntel |
| Model name: | 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz |
| CPU family: | 6 |
| Model: | 140 |
| Thread(s) per core: | 2 |
| Core(s) per socket: | 4 |
| Socket(s): | 1 |
| Stepping: | 1 |
| CPU(s) scaling MHz: | 24% |
| CPU max MHz: | 4200.0000 |
| CPU min MHz: | 400.0000 |
| BogoMIPS: | 4838.40 |

### About the code:

**MandelBrot Problem(From GFG)**

- The Mandelbrot set is a collection of complex numbers where the iterative sequence `z_{n+1} = z_n^2 + c` remains bounded.
- Each point represents a complex number `c = x + yi`, with the set visualized as a fractal image.
- The function is iteratively applied to determine if the sequence diverges for each point.
- Points are colored based on the number of iterations before divergence, creating a detailed fractal.
- Uses complex numbers, iterative functions, and the concept of fractals to define and visualize the set.

## Analyzing the metrics:

| Metric | Value | Details |
|---|---|---|
| **Task Clock** | 657.10ms | |
| **CPUs Utilized** | 0.930 | |
| **Context Switches** | 50 | 76.092 /sec |
| **CPU Migrations** | 6 | 9.131 /sec |
| **Page Faults** | 76 | 115.660 /sec |
| **Cycles** | 2,236,805,514 | 3.404 GHz |
| **Instructions** | 4,406,708,556 | 1.97 instructions per cycle |
| **Branches** | 617,164,238 | 939.226 M/sec |
| **Branch Misses** | 25,901,643 | 4.20% of all branches |
| **TopdownL1** | | |
| **- Backend Bound** | 13.1% | |
| **- Bad Speculation** | 24.8% | |
| **- Frontend Bound** | 22.2% | |
| **- Retiring** | 40.0% | |
| **Time Elapsed** | 1.006904578 seconds | |
| **User Time** | 0.646168000 seconds | |
| **System Time** | 0.006917000 seconds | |

## Analyzing using linux command:

| Metric | `cd` | `pwd` | `ls` |
|---|---|---|---|
| **Task Clock** | 2.03 msec | 1.22 msec | 0.67 msec |
| **CPUs Utilized** | 0.768 | 0.203 | 0.327 |
| **Context Switches** | 0 | 0 | 2 |
| **CPU Migrations** | 0 | 0 | 0 |
| **Page Faults** | 138 | 69 | 97 |
| **Cycles** | 5,905,836 | 2,674,957 | 2,488,525 |
| **Instructions** | 2,885,653 | 1,211,022 | 2,110,343 |
| **Branches** | 591,225 | 253,999 | 429,366 |
| **Branch Misses** | 15,941 | 7,953 | 14,165 |
| **TopdownL1** | | | |
| **- Backend Bound** | 39.6% | 48.6% | 32.5% |
| **- Bad Speculation** | 5.6% | 5.5% | 10.7% |
| **- Frontend Bound** | 44.7% | 35.6% | 37.5% |
| **- Retiring** | 10.2% | 10.2% | 19.2% |
| **Time Elapsed** | 0.002646429 seconds | 0.006021428 seconds | 0.002059844 seconds |
| **User Time** | 0.001372000 seconds | 0.000000000 seconds | 0.000000000 seconds |
| **System Time** | 0.001379000 seconds | 0.003443000 seconds | 0.001275000 seconds |

# Analysis

## 1) Resources Utilization:

- **`cd`**: Shows the highest task clock time (2.03 msec) and CPU cycles (5,905,836), indicating more resource usage compared to `pwd` and `ls`. It has the lowest user and system time, but higher page faults.
- **`pwd`**: Consumes less task clock time (1.22 msec) and CPU cycles (2,674,957) than `cd`. It also has a lower number of page faults and branch misses.
- **`ls`**: Has the lowest task clock time (0.67 msec) and a relatively high number of CPU cycles (2,488,525) and instructions (2,110,343). It has a moderate number of context switches and page faults compared to `pwd` and `cd`.

## 2) Efficinecy

- **`ls`** appears to be the most efficient in terms of task clock and system time. Despite having a moderate number of context switches and page faults, it uses fewer resources compared to `cd`.
- **`pwd`** has low task clock time and system time, but the number of page faults and branch misses are relatively higher than `ls`.
- **`cd`** has the highest task clock time and CPU cycles, indicating it might be doing more work or involving more complex operations.

## 3) Performance Characteristics

- **`ls`**: Likely involves scanning directory contents, which explains the higher cycles and instructions. Despite this, it is efficient in terms of time elapsed.
- **`pwd`**: Primarily involves retrieving and printing the current directory path, resulting in lower resource usage and faster execution.
- **`cd`**: Although it appears simple, it might involve filesystem operations or other underlying operations that contribute to its higher resource usage.

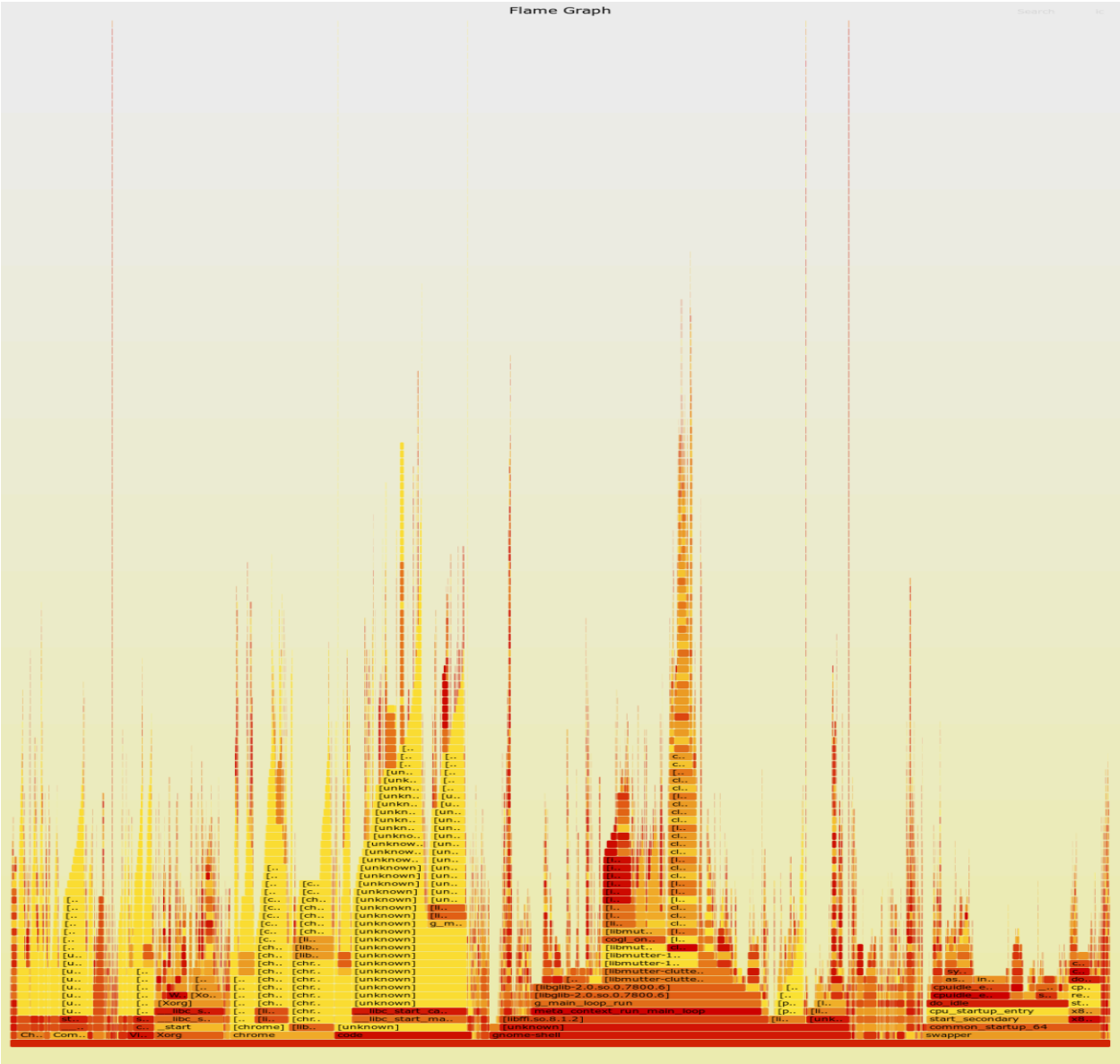
## 4) Potential Bottlenecks

- **`cd`**: The high number of CPU cycles and page faults suggest potential inefficiencies or additional overhead associated with changing directories.
- **`pwd`**: Higher branch misses and page faults could be an area for further investigation to improve efficiency.

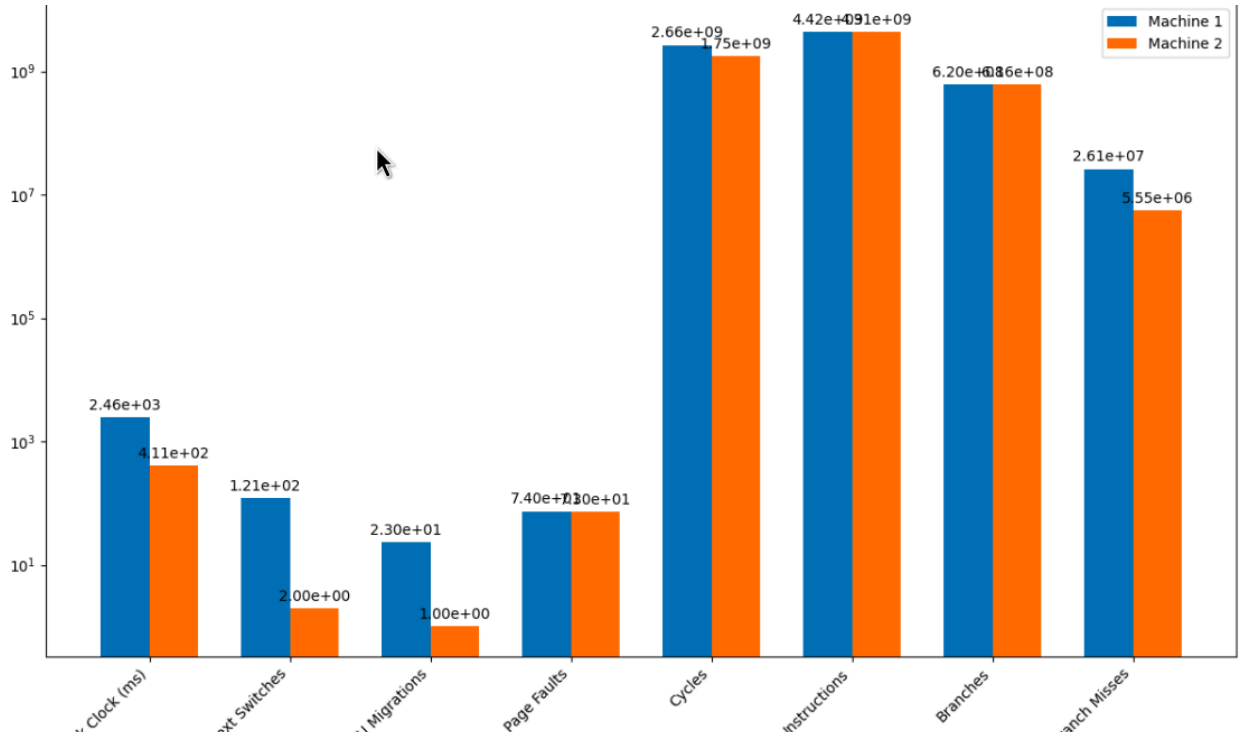
## My Observations

- **For `cd`**: Investigate the filesystem operations or additional tasks performed during directory changes. Optimization might involve reducing overhead or improving efficiency in handling filesystem operations.
- **For `pwd`**: While efficient, reducing branch misses and page faults can further optimize performance, especially for frequent use.

### Flamegraph



## Visualizing the data:




## Perf Bench command:

`Perf bench syscall all`

| Benchmark | Calls Executed | Total Time | Time per Operation | Operations per Second |
|---|---|---|---|---|
| `syscall/basic` | 10,000,000 | 1.374 sec | 0.137433 µs/op | 7,276,299 ops/sec |
| `syscall/getpgid` | 10,000,000 | 1.453 sec | 0.145394 µs/op | 6,877,853 ops/sec |
| `syscall/fork` | 10,000 | 12.253 sec | 1225.313300 µs/op | 816 ops/sec |
| `syscall/execve` | 10,000 | 11.613 sec | 1161.368400 µs/op | 861 ops/sec |

- **Calls Executed**: Number of system calls performed in the benchmark.
- **Total Time**: Total time taken to execute the benchmark.
- **Time per Operation**: Average time taken for each individual system call.
- **Operations per Second**: Number of system calls handled per second.

# Machine-2 Specifications:
## OS-Kali Linux

| | |
|---|---|
| Architecture: | x86_64 |
| CPU op-mode(s): | 32-bit, 64-bit |
| Address sizes: | 48 bits physical, 48 bits virtual |
| Byte Order: | Little Endian |
| CPU(s): | 12 |
| On-line CPU(s) list: | 0-11 |
| Vendor ID: | AuthenticAMD |
| Model name: | AMD Ryzen 5 5625U with Radeon Graphics |
| CPU family: | 25 |
| Model: | 80 |
| Thread(s) per core: | 2 |
| Core(s) per socket: | 6 |
| Socket(s): | 1 |
| Stepping: | 0 |
| CPU(s) scaling MHz: | 31% |
| CPU max MHz: | 4388.0000 |
| CPU min MHz: | 400.0000 |
| BogoMIPS: | 4591.50 |

## Analyzing the metrics:

| Metrics | Value | Details |
|---|---|---|
| **Task Clock** | 411.07 msec | 0.997 CPUs utilized |
| **Context Switches** | 2 | 4.865 /sec |
| **CPU Migrations** | 1 | 2.433 /sec |
| **Page Faults** | 73 | 177.585 /sec |
| **Cycles** | 1,746,945,076 | 4.250 GHz |
| **Stalled Cycles (Frontend)** | 35,530,577 | 2.03% frontend cycles idle |
| **Instructions** | 4,312,074,188 | 2.47 insn per cycle |
| **Stalled Cycles per Instruction** | 0.01 | |
| **Branches** | 616,022,963 | 1.499 G/sec |
| **Branch Misses** | 5,550,725 | 0.90% of all branches |
| **Elapsed Time** | 0.412235896 sec | |
| **User Time** | 0.408745000 sec | |
| **System Time** | 0.004007000 sec | |

## Observation

| | Machine-1(Fedora) | Machine-2(Kali) |
|---|---|---|
| **Run time** | 1.0069sec | 0.4122sec |

## 1) Hardware Differences:

- **Kali Linux Machine:** 12 CPU cores (6 cores, 12 threads) with a maximum clock speed of 4.39 GHz.
- **Fedora Linux Machine:** 8 CPU cores (4 cores, 8 threads) with a maximum clock speed of 4.20 GHz.
- The Kali machine has more cores and threads, which can lead to better parallel processing capabilities, especially in multi-threaded applications, reducing execution time.

## 2) CPU Architecture:

- **Kali:** AMD Ryzen 5 5625U (Zen 3 architecture).
- **Fedora:** Intel Core i5-1135G7 (Tiger Lake architecture).
- The architectures of the CPUs can impact performance based on how well each is optimized for certain tasks. Ryzen processors are known for better multi-threaded performance, which might explain the faster execution on the Kali machine.

## 3) Kernel Optimization:

- **Kali Linux:** Often uses a custom kernel optimized for low latency and quick response times, which benefits short, intensive tasks. This can result in faster execution times for certain workloads, especially those that are CPU-bound.
- **Fedora Linux:** Uses a more general-purpose kernel optimized for a balance between performance, stability, and security. This might not be as finely tuned for performance in high-demand scenarios, leading to slightly slower execution times.

---

---