

# Assignment 1 (T1)

Name : Pratyay Prasad Dhond

MIS : 142203004

Subject : Computer Organisation

## ▼ 1. Study and experimentation using **perf** tool to observe different statistics of a program

### Install **perf** tool in Ubuntu Linux

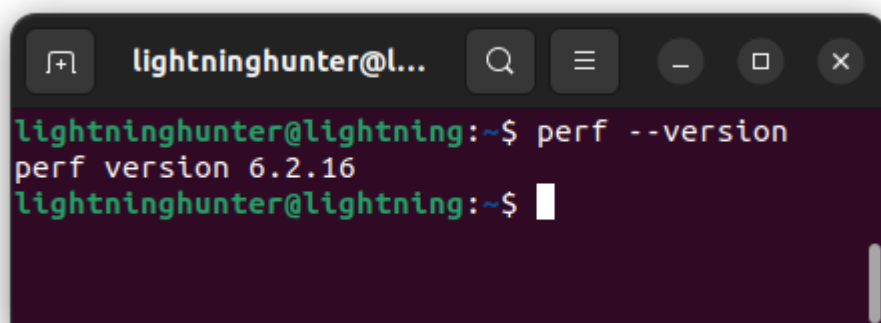
#### 1. Installing Perf and its dependencies

```
$> sudo aptget install linux-tools-common linux-tools-generic linux-tools-`uname -r`
```

- Enter and execute the above command in the terminal to install all the required dependencies and the perf tool itself.

#### 1. Verifying the version of the installed perf tool

```
lightninghunter@lightning:~$ perf --version
```



```
lightninghunter@lightning:~$ perf --version
perf version 6.2.16
lightninghunter@lightning:~$
```

- Note that the `perf` command requires `sudo` privileges by default. To allow regular users to use `perf`, you have to execute the following command in your terminal:

```
sudo sysctl -w kernel.perf_event_paranoid=1
```

## What is `perf` tool?

- `perf` - Performance Analysis Tool for Linux
- Performance counters for Linux are a new kernel-based subsystem that provide a framework for all things performance analysis.
- It covers hardware level (CPU/PMU, Performance Monitoring Unit) features and software features (software counters, tracepoints) as well.
- It is capable of statistical profiling of the entire system (both kernel and user code).

## Events in `perf`

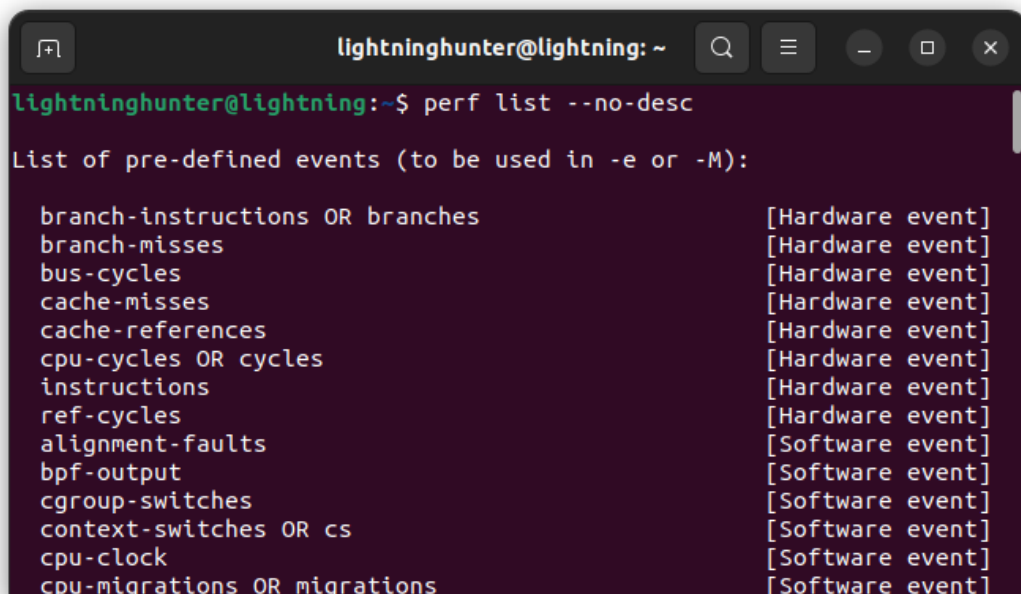
- To get the list of all available events (approx 3500+ lines of specification) use the command  

```
$> perf list
```
- One of the source of events is the processor itself and its Performance Monitoring Unit (PMU).
- It provides a list of events to measure micro-architectural events such as the number of cycles, instructions retired, L1 cache misses and so on.
- Those events are called **PMU hardware events** or **hardware events** in short. They vary with each processor type and model.
- The events in `perf` can be categorised into two parts
  1. Software Events
  2. Hardware Events

## Hardware Events and Their Explanation:

- The number of hardware events that can be monitored using the `perf` command in Linux depends on several factors, including your CPU architecture, the specific CPU model, and the Linux kernel version you are using.
- Different CPU architectures provide different sets of performance monitoring counters (PMCs), and the kernel must support these counters.
- For example, x86 CPUs have more hardware events than the other architectures.
- Additionally, specific CPU models within the same architecture may offer different sets of events.
- To get the list of all available event we can use the following command
  - The command below will list all the possible events according to the build model and specifications of the computer's processor.

```
$> perf list --no-desc
```



```
lightninghunter@lightning:~$ perf list --no-desc
List of pre-defined events (to be used in -e or -M):

branch-instructions OR branches      [Hardware event]
branch-misses                        [Hardware event]
bus-cycles                           [Hardware event]
cache-misses                         [Hardware event]
cache-references                     [Hardware event]
cpu-cycles OR cycles                 [Hardware event]
instructions                         [Hardware event]
ref-cycles                           [Hardware event]
alignment-faults                    [Software event]
bpf-output                           [Software event]
cgroup-switches                     [Software event]
context-switches OR cs              [Software event]
cpu-clock                           [Software event]
cpu-migrations OR migrations        [Software event]
```

- These events can be mainly categorised into the following types:
  1. Hardware Event
    - A low level performance metric. Monitored using hardware performance counters.
  2. Software Event
    - performance metric related to software execution such as number of context switches, page faults, etc.
    - Used to analyse behaviour and performance of software.
  3. Tool Event
    - Not a standard term, refers to events generated or tracked by profiling and monitoring tools like `perf` and other tools.
  4. Hardware Cache Event
    - Performance metrics related to CPU cache behaviour
    - This includes cache hits, misses, cache-related events used to optimise memory.
  5. Kernel PMU event
    - Performance metrics collected by the Linux Kernel using CPU performance counters
    - This is done for monitoring and profiling system and application performance.

## Hardware Events and Their Explanation

- All the above explanations are with respect to `perf` tool and the statistics it provides.

### 1. branch-instructions or branches

- The "branch-instructions" event counts the *total number of branch* instructions executed by a program.
- branch-instructions are instructions that control the flow of the program using either *conditional branches* [ `if-else` statements] or *unconditional branches* [ `functions` and `return statements` ].

- Monitoring these branch instructions helps in the understanding of program's branching/jumping mechanisms and how often branches are encountered in your code.
- This metric provides insights into how frequently the program's control flow diverges, or in simpler terms breaks the continuous linear path and performs a jump to some other address.

Example:

- ( N-Queen's problem)

```
$> perf stat -e branch-instructions ./NQueens
```

The screenshot shows a terminal window with the following output:

```
lightninghunter@lightning: ~/Coding/DSA/recursion
lightninghunter@lightning:~/Coding/DSA/recursion$ perf stat -e branch-instructions ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':
      7,97,199      branch-instructions
    2.855801365 seconds time elapsed
    0.000000000 seconds user
    0.002375000 seconds sys

lightninghunter@lightning:~/Coding/DSA/recursion$
```

- Branches also vary based on input

```
lightninghunter@lightning: ~/Coding/DSA/recursion
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0

Performance counter stats for './NQueens':

    14,40,51,553      branches

    0.740765521 seconds time elapsed

    0.146495000 seconds user
    0.000000000 seconds sys

lightninghunter@lightning:~/Coding/DSA/recursion$
```

## 2. branch-misses

- A performance metric that counts the number of branch instructions that were incorrectly predicted by the CPU's branch prediction mechanism.
- When a branch instruction is incorrectly predicted, the CPU needs to discard the incorrect speculative work and re-execute the instructions based on the correct branch outcome.
- The above prediction might lead to a performance penalty.
- Monitoring the “branch-misses” with `perf` will help you in assessing the efficiency of the branch prediction mechanism for a given program.
- A high count of branch misses may indicate that your code has sub-optimal branch behavior, potentially leading to performance bottlenecks.
- *Number of branch-misses* can be valuable information when optimising code for better performance.

What is a branch Prediction Mechanism?

- A branch predictor is a (very important) component of modern computer processors that helps improve the execution speed of programs by making informed predictions about the outcome of conditional branch instructions in a program's code.
- Branch predictor only comes into the play during the case of conditional branching.

- Branch predictors are essential for improving CPU performance because they allow the CPU to continue executing instructions without waiting for the branch condition to be evaluated.
- If the predictor makes accurate predictions most of the time, the CPU can maintain a high instruction throughput, resulting in faster program execution.
- Predictors come in a lot of types
  1. two-level predictors (simpler one)
  2. tournament predictors
  3. Neural predictors

### Example

- (N Queen's Problem)

```
$ perf stat -e branch-misses ./NQueens
```

The screenshot shows a terminal window with the following output:

```
lightninghunter@lightning: ~/Coding/DSA/recursion
lightninghunter@lightning:~/Coding/DSA/recursion$ perf stat -e branch-misses ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

      19,161      branch-misses

      1.675304463 seconds time elapsed

      0.002688000 seconds user
      0.000000000 seconds sys

lightninghunter@lightning:~/Coding/DSA/recursion$
```

### 3. Bus-cycles

- It refers to a hardware event that counts the number of bus cycles used during the execution of a program.

- The bus cycle here represents the number of clock cycles during which the CPU's data is actively transferring data between the CPU and other components of the system, such as memory or peripheral devices
- Monitoring the `Bus-cycles` can provide insights into memory and I/O subsystem efficiency.
- High bus cycle count may indicate that the CPU is spending a significant amount of time waiting for data to be transferred between the CPU and memory or other devices.

### Example

- (N Queen's Problem)

```
$ perf stat -e bus-cycles ./NQueens
```

```
lightninghunter@lightning: ~/Coding/DSA/recursion
lightninghunter@lightning:~/Coding/DSA/recursion$ perf stat -e bus-cycles ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

      47,941      bus-cycles

    0.598336323 seconds time elapsed

    0.002420000 seconds user
    0.000000000 seconds sys

lightninghunter@lightning:~/Coding/DSA/recursion$
```

## 4. cache-misses

- It refers to a performance metric that counts the number of times data or instructions were not found in the CPU's cache memory and had to be fetched from a higher-level cache or main memory
- Cache misses can occur in different levels of the CPU's memory hierarchy, such as L1 (Level 1) cache, L2 (Level 2) cache, or even in main memory (RAM).



- Monitoring these cache-misses can help in better understanding of how efficiently the program is utilising the CPU's cache hierarchy.
- High cache miss rates can indicate not-so-optimal memory access patterns, which can lead to reduced program performance.

### Example

- (N Queen's Problem)

```
$ perf stat -e cache-misses ./NQueens
```

```
lightninghunter@lightning: ~/Coding/DSA/recursion
lightninghunter@lightning:~/Coding/DSA/recursion$ perf stat -e cache-misses ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

      51,746      cache-misses

    0.738886239 seconds time elapsed

    0.000000000 seconds user
    0.001717000 seconds sys

lightninghunter@lightning:~/Coding/DSA/recursion$
```

- It is possible to **monitor specific cache misses** in `perf` as well:
- Cache misses types are:
  1. **L1 data cache Misses**
    - For monitoring L1 data cache misses on both data loads, and data stores.
  1. L1-dcache-load-misses
  2. L1-dcache-store-misses

```
perf stat -e L1-dcache-load-misses,L1-dcache-store-misses ./NQueens
```

```
lightninghunter@lightning: ~/Coding/DNA/recursion
lightninghunter@lightning:~/Coding/DNA/recursion$ perf stat -e L1-dcache-load-misses,L1-dcache-store-misses ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

      42,950      L1-dcache-load-misses
<not supported>  L1-dcache-store-misses

      1.188301936 seconds time elapsed

      0.002677000 seconds user
      0.000000000 seconds sys
```

- As in the above execution we are not storing any data just reading from cache, there are no L1-data cache- store-misses

2. **L1 Instruction Cache Misses:** To monitor L1 instruction cache misses, you can use the event "L1-icache-load-misses."

```
perf stat -e L1-icache-load-misses ./NQueens
```

```
lightninghunter@lightning: ~/Coding/DNA/recursion
lightninghunter@lightning:~/Coding/DNA/recursion$ perf stat -e L1-icache-load-misses ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

      91,170      L1-icache-load-misses

      1.491187900 seconds time elapsed

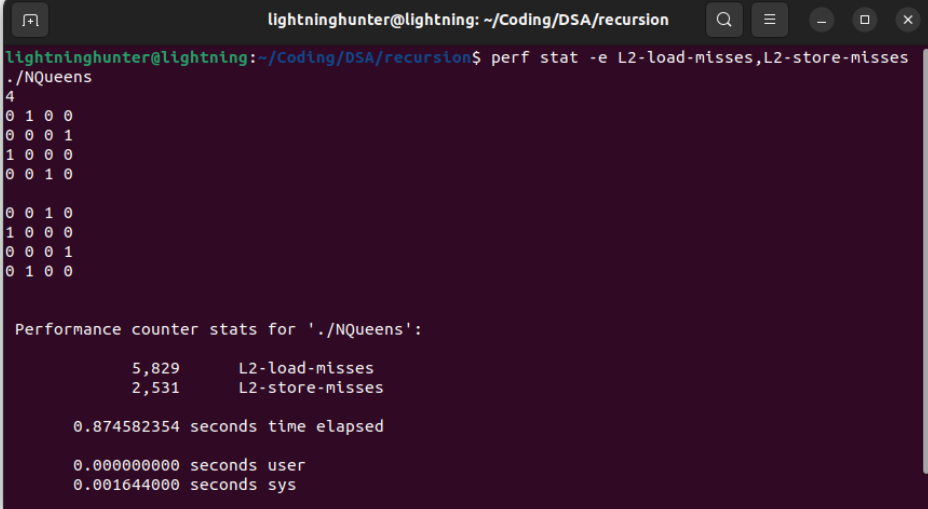
      0.000000000 seconds user
      0.002596000 seconds sys
```

3. **L2 Cache Misses:** To monitor L2 cache misses, you can use the events:

- **L2-load-misses** for cache misses on data loads

- **L2-store-misses** for cache misses on data stores.

```
perf stat -e L2-load-misses,L2-store-misses ./NQueens
```



```
lightninghunter@lightning: ~/Coding/DSA/recursion
lightninghunter@lightning:~/Coding/DSA/recursion$ perf stat -e L2-load-misses,L2-store-misses ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

          5,829      L2-load-misses
          2,531      L2-store-misses

    0.874582354 seconds time elapsed

    0.000000000 seconds user
    0.001644000 seconds sys
```

There are also other levels of cache-misses that can be explicitly mentioned for the specific level's number of cache misses, such as:

1. Last Level Cache (LLC) misses
2. Total Cache Misses
3. Data Prefetcher Cache Misses
4. Instruction Prefetcher Cache Misses

## 5. Cache References

- This refers to a performance metric that counts the total number of cache memory references made during the execution of a given program.
- This includes both **cache hits** and **cache misses**.
- Cache references provides insight into how often the CPU accesses its cache memory during program execution.
- High count of cache references suggests that the CPU is frequently accessing cache memory, which can be an indicator of efficient memory access patterns and good cache utilisation

- It should be noted that, the efficiency of the cache usage depends not only on the number of references but also on the cache hit rate.

#### Example

- (N Queen's Problem)

```
perf stat -e cache-references ./NQueens
```

```
lightninghunter@lightning: ~/Coding/DSA/recursion
lightninghunter@lightning:~/Coding/DSA/recursion$ perf stat -e cache-references ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

      1,50,892      cache-references

    0.680644160 seconds time elapsed

    0.002530000 seconds user
    0.000000000 seconds sys

lightninghunter@lightning:~/Coding/DSA/recursion$
```

## 6. cpu-cycles or cycles

- refers to a performance metric that counts the ***number of processor cycles*** used during the execution of a program.
- It provides an estimate of the **total time spent executing instructions** on the CPU.
- It can help you assess the efficiency of your program's execution in terms of how much time it consumes on the CPU.
- It's a useful metric for understanding the overall performance of your code and identifying potential bottlenecks
- By analysing the `cpu cycles` metric along with other performance metrics, you can gain insights into your program's efficiency and identify areas where optimisation may be needed to improve performance.

#### Example

- (N Queen's Problem)

```
perf stat -e cycles ./NQueens
```

```
lightninghunter@lightning: ~/Coding/DSA/recursion
lightninghunter@lightning:~/Coding/DSA/recursion$ perf stat -e cycles ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

    51,54,565      cycles

    0.661114365 seconds time elapsed

    0.001698000 seconds user
    0.000000000 seconds sys

lightninghunter@lightning:~/Coding/DSA/recursion$
```

## 7. Instructions

- It refers to a performance metric that counts the total number of instructions retired or executed by the CPU during the execution of a program.
- This metric provides insight into how many instructions your program has executed.
- Monitoring the `instructions` helps you understand the computational complexity of your program and can be used to assess its efficiency in terms of instruction execution.

### Example

- (N Queen's Problem)

```
$ perf stat -e instructions ./NQueens
```

```
lightninghunter@lightning: ~/Coding/D5A/recursion
lightninghunter@lightning:~/Coding/D5A/recursion$ perf stat -e instructions ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

      47,42,952      instructions
    0.465629669 seconds time elapsed

    0.001590000 seconds user
    0.000000000 seconds sys

lightninghunter@lightning:~/Coding/D5A/recursion$
```

## 8. ref-cycles

- A reference cycle is a situation that occurs when two or more objects have strong references to each other, creating a cycle of references that cannot be broken
- This can lead to memory leaks, as the objects involved in the cycle cannot be de-allocated by the garbage collector.
- `ref-cycles` is not a standard or very common event name.
- It is used to measure the number of cycles based on a reference clock signal in the CPU.
- concept of *reference cycles* may only apply to certain CPU architectures and availability of this event may depend on your CPU model and kernel version.

### Example

- (N Queen's Problem)

```
$ perf stat -e ref-cycles ./NQueens
```

```
lightninghunter@lightning: ~/Coding/DSA/recursion
lightninghunter@lightning:~/Coding/DSA/recursion$ perf stat -e ref-cycles ./NQueens
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Performance counter stats for './NQueens':

        63,58,284          ref-cycles

    0.768629603 seconds time elapsed

    0.003101000 seconds user
    0.000000000 seconds sys

lightninghunter@lightning:~/Coding/DSA/recursion$
```

## ▼ Program to multiply two different matrices of size 1024x1024

### Row Wise Access

- `result[i][j] += matrix1[i][k] * matrix2[k][j];`

```
#include<iostream>
#include<vector>
using namespace std;

#define vvll vector<vector<long long>>

void setRandoms(){
    srand(time(0));
}
// 0 cannot be allowed in the matrix as it will cause 0 in multiplication
void randomiseMatrix(vvll &matrix){
    for(int i = 0; i < 1024; i++){
        for(int j = 0; j < 1024; j++){
            matrix[i][j] = rand() % 20 + 1; // to get values from 1 to 9
        }
    }
}

void multiplyMatrix(vvll &matrix1, vvll &matrix2, vvll& result){
    int n = matrix1.size();
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
```

```

        for(int k = 0; k < n; k++){
            result[i][j] += matrix1[i][k] * matrix2[k][j];
        }
    }
}

void displayMatrix(vvll matrix){
    for(auto row : matrix){
        for(auto val : row)
            cout << val << " ";
        cout << endl;
    }
}

int main(){
    vvll matrix1(1024,vector<long long>(1024));
    vvll matrix2(1024,vector<long long>(1024));
    vvll result(1024,vector<long long>(1024,0));

    setRandoms();
    randomiseMatrix(matrix1);
    randomiseMatrix(matrix2);

    multiplyMatrix(matrix1, matrix2, result);

    // displayMatrix(result);
}

```

## Column Wise Access

- `result[i][j] += matrix1[j][k] * matrix2[k][i];`

```

#include<iostream>
#include<vector>
using namespace std;

#define vvll vector<vector<long long>>

void setRandoms(){
    srand(time(0));
}
// 0 cannot be allowed in the matrix as it will cause 0 in multiplication
void randomiseMatrix(vvll &matrix){
    for(int i = 0; i < 1024; i++){
        for(int j = 0; j < 1024; j++){
            matrix[i][j] = rand() % 20 + 1; // to get values from 1 to 9
        }
    }
}

```



```

void multiplyMatrix(vvll &matrix1, vvll &matrix2, vvll& result){
    int n = matrix1.size();
    for(int j = 0; j < n; j++){
        for(int i = 0; i < n; i++){
            for(int k = 0; k < n; k++){
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}

void displayMatrix(vvll matrix){

    for(auto row : matrix){
        for(auto val : row)
            cout << val << " ";
        cout << endl;
    }
}

int main(){
    vvll matrix1(1024,vector<long long>(1024));
    vvll matrix2(1024,vector<long long>(1024));
    vvll result(1024,vector<long long>(1024,0));

    setRandoms();
    randomiseMatrix(matrix1);
    randomiseMatrix(matrix2);

    multiplyMatrix(matrix1, matrix2, result);

    // displayMatrix(result);
}

```

## Observations:

### First Command

```
$> sudo stat -e cpu-clock (./MatMulRowWise + ./MatMulColWise)
```

```

lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cpu-clock ./MatMulRowWise
Performance counter stats for './MatMulRowWise':
   14,334.87 msec cpu-clock             #    1.000 CPUs utilized

   14.335813925 seconds time elapsed
   14.327315000 seconds user
   0.008001000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cpu-clock ./MatMulRowWise
Performance counter stats for './MatMulRowWise':
   14,357.23 msec cpu-clock             #    1.000 CPUs utilized

   14.359712455 seconds time elapsed
   14.338034000 seconds user
   0.020003000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cpu-clock ./MatMulRowWise
Performance counter stats for './MatMulRowWise':
   13,644.35 msec cpu-clock             #    1.000 CPUs utilized

   13.645875314 seconds time elapsed
   13.640443000 seconds user
   0.004000000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$

lightninghunter@lightning:~/Coding/SEM-5/CO$ cd Coding/SEM-5/CO/
lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cpu-clock ./MatMulColWise
[sudo] password for lightninghunter:
Performance counter stats for './MatMulColWise':
   12,729.60 msec cpu-clock             #    0.998 CPUs utilized

   12.749735046 seconds time elapsed
   12.709962000 seconds user
   0.019996000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cpu-clock ./MatMulColWise
Performance counter stats for './MatMulColWise':
   12,729.07 msec cpu-clock             #    1.000 CPUs utilized

   12.730579633 seconds time elapsed
   12.722064000 seconds user
   0.008001000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cpu-clock ./MatMulColWise
Performance counter stats for './MatMulColWise':
   12,337.71 msec cpu-clock             #    1.000 CPUs utilized

   12.338779789 seconds time elapsed
   12.334193000 seconds user
   0.004000000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$

```

## Observations:

- On Average the row-wise multiplication seem to be running a bit slower than the column wise multiplication, even though only by a slight measure.
- This could be caused by a lot of reasons such as
  1. Particular Input ( This result could be occurring for only a certain sub-group of inputs)
  2. Decoding of the instruction at CPU level
  3. Cache memory state at the present and how the cache is being maintained internally

The output of the above can be described as follows:

### 1. Performance Counter Stats:

**14,334.87 msec cpu-clock :**

- This line shows the total CPU clock cycles (in milliseconds) consumed by the program during its execution.
- It represents the amount of time the CPU spent executing the program.
- In this case, the program consumed approximately 14,334.87 milliseconds of CPU time.

```
# 1.000 CPUs utilized :
```

- This indicates that the program utilized one CPU core during its execution.
- It means that the program was single-threaded and ran on a single CPU core.

## 2. Time Elapsed:

- `14.335813925 seconds time elapsed` : This line displays the total time elapsed (in seconds) from the start of the program's execution until it completed. In this case, the program took approximately 14.336 seconds to complete.

## 3. User and System Time:

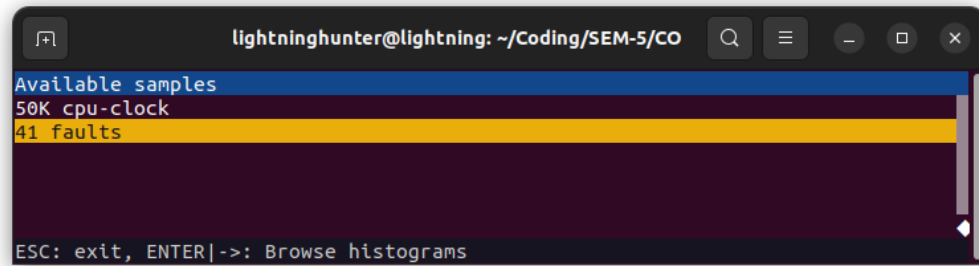
- `14.327315000 seconds user` : This represents the amount of time (in seconds) the program spent in user mode, executing its own code.
- `0.008001000 seconds sys` : This represents the amount of time (in seconds) the program spent in system mode, executing kernel (operating system) code on behalf of the program. System time is usually minimal for user-level programs, as it primarily involves interaction with the operating system.

## Second Command

```
$> sudo perf record -e cpu-clock,faults ./MatMulRowWise
```

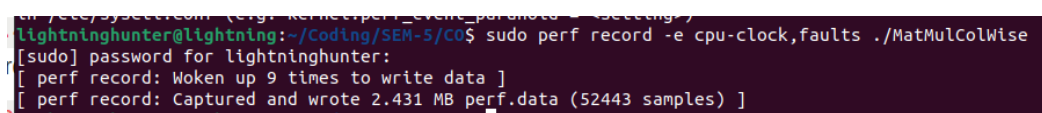
- the `perf` commands `record` attribute is used to record the performance metrics of the data related to the specified events
- The output of the file will be stored in a newly created file `perf.data` which can be only accessed using sudo access
- These reports can be further analysed by a variety of commands such as `perf report` , `perf script` , `perf diff` , etc.

```
$> sudo perf report -i perf.data
```



```
lightninghunter@lightning: ~/Coding/SEM-5/CO
Available samples
50K cpu-clock
41 faults
ESC: exit, ENTER|->: Browse histograms
```

output for `perf report -i perf.data`

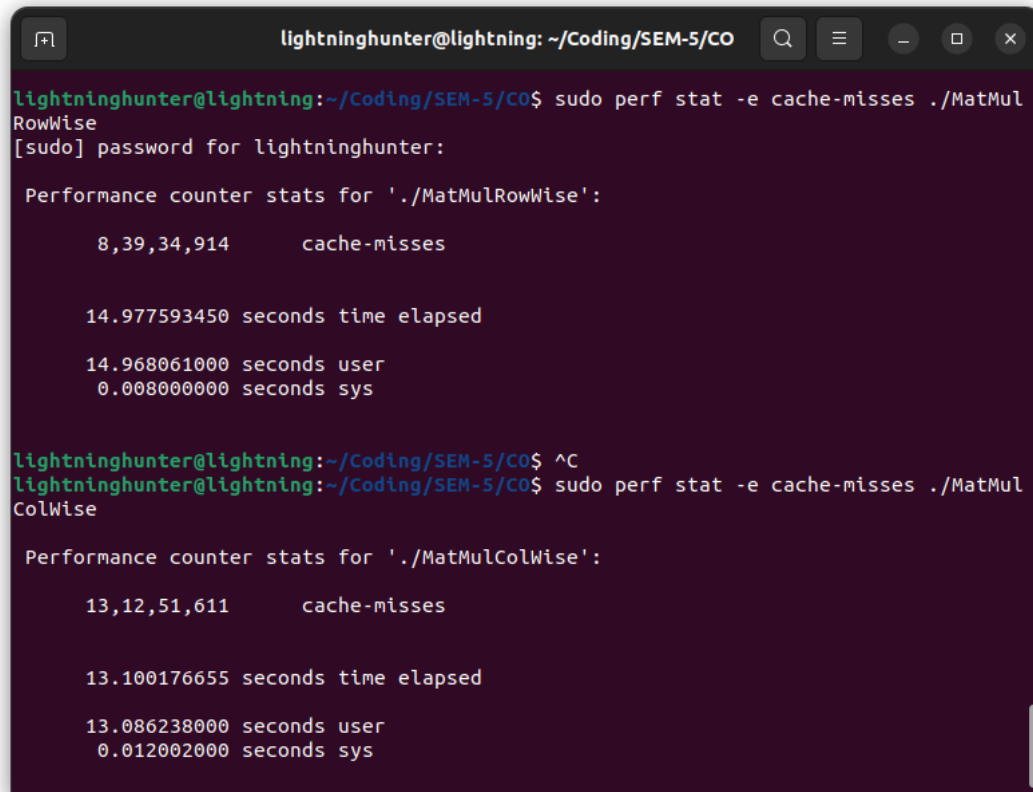


```
lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf record -e cpu-clock,faults ./MatMulColWise
[sudo] password for lightninghunter:
[ perf record: Woken up 9 times to write data ]
[ perf record: Captured and wrote 2.431 MB perf.data (52443 samples) ]
```

1. `[ perf record: Woken up 9 times to write data ]`:
  - This message indicates that the `perf record` process was periodically woken up 9 times during its execution to write collected performance data to the `perf.data` file.
  - This periodic writing helps ensure that the data is recorded and saved efficiently.
2. `[ perf record: Captured and wrote 2.431 MB perf.data (52443 samples) ]`:
  - This message provides a summary of the recording process:
  - `Captured` indicates that `perf record` successfully captured the performance data.
  - `wrote 2.431 MB perf.data`: Indicates that the recorded data was written to a file named `perf.data`, and the file size is 2.431 megabytes.
  - `(52443 samples)`:
    - Number of samples collected during the recording process.
    - Samples are individual measurements taken to profile the program's performance.

## Command 3

```
$> sudo perf stat -e cache-misses ./MatMul(Col+Row)Wise
```



```
lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cache-misses ./MatMul
RowWise
[sudo] password for lightninghunter:

Performance counter stats for './MatMulRowWise':

      8,39,34,914      cache-misses

      14.977593450 seconds time elapsed

      14.968061000 seconds user
       0.008000000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$ ^C
lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cache-misses ./MatMul
ColWise

Performance counter stats for './MatMulColWise':

     13,12,51,611      cache-misses

     13.100176655 seconds time elapsed

     13.086238000 seconds user
       0.012002000 seconds sys
```

- The above output shows the `cache-misses` in both Row-wise multiplication and column wise multiplication
- Row wise multiplication has **less cache-misses** as compared to column wise multiplication.
- This is because of the way caching works internally, the number of instructions executed or retired throughout the execution of the program may vary as it depends on how the compiler would be unrolling the for loops.
- It is also compiler and OS dependent how many instructions are executed per cycle, there are less number of instructions in row Wise multiplication but the Instructions per cycle is more as compared to column wise execution which is ultimately making the column wise multiplication execute faster irrespective of having less instruction to execute in the first place.
- As for the cache misses,

```
lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cache-misses,cache-references
./MatMulRowWise

Performance counter stats for './MatMulRowWise':

      4,32,76,605      cache-misses              #    9.047 % of all cache refs
      47,83,41,760      cache-references

      13.544287297 seconds time elapsed

      13.531985000 seconds user
       0.011999000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$ sudo perf stat -e cache-misses,cache-references
./MatMulColWise

Performance counter stats for './MatMulColWise':

      23,03,40,048      cache-misses              #   28.646 % of all cache refs
      80,40,77,725      cache-references

      25.523916795 seconds time elapsed

      25.439861000 seconds user
       0.035999000 seconds sys
```

- The percentage of `cache-misses` in row wise multiplications is drastically less as compared to column wise multiplication, this is for the following reasons:
  1. When multiplying two matrices, let's say A and B, in row wise manner, we will be calculating the values for the resultant matrix sequentially, that is row after row. Whereas in Column wise multiplication we would be doing exactly opposite, we would calculate the resultant matrix, column by column, meaning from cell 0 to 1023 in the first column, then cell 0 of second column to 1023 of second column till 1023rd column.
  2. Due to this, as in matrix multiplication, the cached data is mostly from the previous accesses which would be from the same row, it would be a better-fetch scenario for caching as in C++, the language used for the code here, the matrices are stored in a row-major order, which makes row access more efficient and memory friendly.
  3. Along with that, as multiplication requires the complete current row of Matrix A, and complete current column of matrix B, the data from the current row is cached in the L1 cache, whereas the remaining data is kept into the L2 cache and L3 cache as follows.

4. The row wise multiplication would require the same row of data 1024 times for the entire row, whereas in column wise multiplication the rows would be different every single time thus needing different data on the cache.
  5. As the size of L1 cache is less all this data required in column wise multiplication cannot be stored in L1 cache, so instead it is stored in the L2 cache and the L3 cache at the end.
  6. When the processor searches for data in the L1 cache, there is a high chance of it not being available in the L1 cache in the column wise multiplication. This causes the processor to look for the data in the L2 cache, and then swap blocks from desired data in L2 cache with a less priority data in L1 cache.
  7. This process of swapping is also known as eviction, where a data block is evicted and moved to the L1 cache where previously a data block resided which is moved to L2 cache.
  8. Irrespective of this process, which in itself consumes time. When the processor looks for a particular data in L1 Cache and doesn't find it, this is known as a `cache-miss` and the number of `cache-misses` in the Column wise multiplication is more as compared to row wise multiplication due to the reason explained in point no 4.
  9. Thus in the above example row-wise multiplication has 9% cache-misses as compared to 28.6% cache misses of the column-wise multiplication.
- So in conclusion, we can say row-wise multiplication is better than column wise multiplication due to the higher `cache-hit` rate, and thus lower `cache-miss` rates.

### Conclusion:

- In Conclusion, `Col-Wise Multiplication` might run faster on some computers due to underling hardware and row-major / column-major based on language preference but in general case `row-Wise Multiplication` will always be preferred as it has high `cache-hit to cache-miss` ratio, meaning it is more likely to find data on the cache itself and save time in the longer run, thus being more efficient.

Additional Stats:

## Row Wise Access Stats

```
lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ perf stat -e branch-instructions,branch-misses,bus-cycles,cache-misses,cache-references,cpu-cycles,instructions,ref-cycles ./MatMulRowWise

Performance counter stats for './MatMulRowWise':

   14,02,01,28,383      branch-instructions      (74.95%)
      18,35,330         branch-misses           #    0.01% of all branches (74.96%)
   43,60,53,236         bus-cycles               (75.00%)
   20,75,42,002         cache-misses             #   30.297 % of all cache refs (75.01%)
   68,50,18,761         cache-references         (50.05%)
   74,30,09,92,288      cpu-cycles               (50.03%)
  1,36,85,72,05,489     instructions             #    1.84  insn per cycle   (62.51%)
   47,09,73,85,030     ref-cycles               (74.99%)

   18.457430715 seconds time elapsed

   18.380845000 seconds user
    0.051968000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$
```

Experiment 1

```
lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ perf stat -e branch-instructions,branch-misses,bus-cycles,cache-misses,cache-references,cpu-cycles,instructions,ref-cycles ./MatMulRowWise

Performance counter stats for './MatMulRowWise':

   14,04,84,26,148      branch-instructions      (75.00%)
      20,23,928         branch-misses           #    0.01% of all branches (75.00%)
   51,38,47,448         bus-cycles               (75.00%)
   21,95,79,312         cache-misses             #   27.382 % of all cache refs (75.00%)
   80,19,19,776         cache-references         (49.99%)
   84,59,59,69,320      cpu-cycles               (49.99%)
  1,36,77,32,06,622     instructions             #    1.62  insn per cycle   (62.49%)
   55,50,31,29,539     ref-cycles               (74.99%)

   21.827320641 seconds time elapsed

   21.796546000 seconds user
    0.027995000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$
```

Experiment 2



```
lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ perf stat -e branch-instructions,branch-misses,bus-cycles,cache-misses,cache-references,cpu-cycles,instructions,ref-cycles ./MatMulRowWise

Performance counter stats for './MatMulRowWise':

   14,00,08,48,784      branch-instructions          (74.98%)
      18,96,690      branch-misses                #    0.01% of all branches    (74.97%)
   49,32,41,939      bus-cycles                    (74.98%)
   20,88,86,149      cache-misses                #   26.625 % of all cache refs (75.02%)
   78,45,37,191      cache-references            (50.04%)
   81,07,91,01,272      cpu-cycles                  (50.01%)
  1,36,70,08,76,650      instructions                #    1.69 insn per cycle     (62.50%)
   53,26,93,71,787      ref-cycles                  (74.99%)

   20.848067956 seconds time elapsed

   20.771070000 seconds user
    0.071982000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$
```

Experiment 3

## Column Wise Access Stats

```
lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ perf stat -e branch-instructions,branch-misses,bus-cycles,cache-misses,cache-references,cpu-cycles,instructions,ref-cycles ./MatMulColWise

Performance counter stats for './MatMulColWise':

   14,04,83,51,008      branch-instructions          (74.97%)
      17,04,435      branch-misses                #    0.01% of all branches    (75.00%)
   33,61,44,864      bus-cycles                    (75.00%)
   23,11,50,267      cache-misses                #   36.553 % of all cache refs (75.02%)
   63,23,64,804      cache-references            (50.03%)
   56,92,67,48,327      cpu-cycles                  (49.98%)
  1,37,19,09,42,378      instructions                #    2.41 insn per cycle     (62.46%)
   36,35,06,04,450      ref-cycles                  (74.95%)

   14.255470914 seconds time elapsed

   14.236252000 seconds user
    0.015995000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$
```

Experiment 1 - Col Wise

```
lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ perf stat -e branch-instructions,branch-misses,bus-cycles,cache-misses,cache-references,cpu-cycles,instructions,ref-cycles ./MatMulColWise

Performance counter stats for './MatMulColWise':

   14,01,63,69,494    branch-instructions          (75.02%)
      19,97,163      branch-misses                #   0.01% of all branches (74.99%)
   32,62,96,772      bus-cycles                    (74.99%)
   16,91,57,891      cache-misses                 #  22.86% of all cache refs (75.02%)
   73,99,22,790      cache-references             (49.99%)
   55,63,18,12,151   cpu-cycles                   (49.99%)
  1,36,73,96,10,576 instructions                 #   2.46 insn per cycle (62.51%)
   35,22,13,54,228  ref-cycles                   (75.01%)

   13.834018524 seconds time elapsed

   13.812643000 seconds user
    0.020006000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$
```

Experiment 2 Col Wise

```
lightninghunter@lightning: ~/Coding/SEM-5/CO
lightninghunter@lightning:~/Coding/SEM-5/CO$ perf stat -e branch-instructions,branch-misses,bus-cycles,cache-misses,cache-references,cpu-cycles,instructions,ref-cycles ./MatMulColWise

Performance counter stats for './MatMulColWise':

   14,02,73,25,903    branch-instructions          (75.00%)
      12,63,235      branch-misses                #   0.01% of all branches (75.00%)
   29,58,48,879      bus-cycles                    (75.00%)
    7,43,38,174      cache-misses                 #  13.09% of all cache refs (74.99%)
   56,77,27,617      cache-references             (50.01%)
   53,27,90,78,361   cpu-cycles                   (50.00%)
  1,36,60,52,11,774 instructions                 #   2.56 insn per cycle (62.50%)
   31,95,47,71,096  ref-cycles                   (75.01%)

   12.574829275 seconds time elapsed

   12.562230000 seconds user
    0.011998000 seconds sys

lightninghunter@lightning:~/Coding/SEM-5/CO$
```

Experiment 3 - Col Wise