Assignment 1: Perf Installation, Types, Common Events, and Execution Options

Sahil Malawde
MIS: 612303156

Aim:

This guide focuses on enhancing application performance using performance analysis tools, with a particular emphasis on perf. By analyzing execution patterns and profiling applications, it aims to identify inefficiencies and computational bottlenecks. The ultimate objective is to optimize resource allocation and boost execution speed.

Theory:

Achieving optimal application performance requires a deep understanding of execution characteristics, resource allocation, and inefficiencies. Several factors influence a program's overall performance, which can be categorized as follows:

Application-Level Factors:

- Algorithm efficiency
- Input/output operations
- Load distribution
- Memory access patterns
- Dataset scalability

Hardware-Level Factors:

- CPU architecture
- Memory structure
- I/O configuration
- Network protocol efficiency

Software-Level Factors:

- Operating system capabilities
- Compiler optimizations
- Preprocessing techniques
- Utilized libraries

Given the complexity of performance optimization, analytical tools play a crucial role in identifying inefficiencies within applications. These tools provide valuable insights into runtime behavior, enabling targeted performance improvements. While the primary objective of optimization is to minimize execution time, secondary goals may include reducing memory or storage usage.

By utilizing performance analysis tools, developers can systematically enhance application behaviour, eliminate inefficiencies, and improve execution speed. The use of perf enables precise performance monitoring and supports iterative optimizations.

System Specifications:

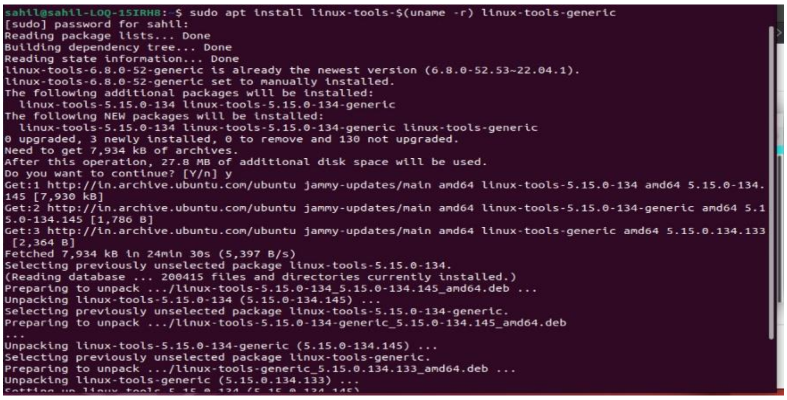
This tutorial will be conducted on a system equipped with an **Intel Core i5 12450H processor**, **16GB RAM**, and running **Ubuntu Linux 24.02 LTS** to evaluate various performance parameters.

Installation of perf in Linux

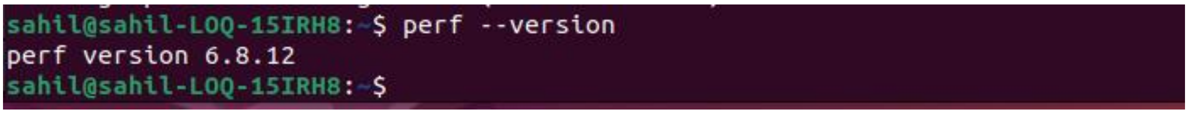
perf is a powerful performance analysis tool in Linux, primarily used for profiling CPU and system performance metrics. Below are the steps followed for installing and setting up perf on a Linux system.

Steps for Installation

1. **Install linux-tools Package**
   The perf tool is included in the linux-tools package, which should match the kernel version in use. Install it using:



2. **Check Version**
   Once installed, verify that perf is available by checking its version:



3. **Grant Necessary Permissions**
   If perf requires elevated permissions for certain profiling tasks, set the appropriate kernel parameters:

   sudo sysctl -w kernel.perf_event_paranoid=1

4. List the available events
   Shows the list of hardware and software events of perf.

This will display the available events that can be monitored.

```
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]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]
  duration_time                                      [Tool event]
  user_time                                          [Tool event]
  system_time                                        [Tool event]

cpu_atom:
  L1-dcache-loads OR cpu_atom/L1-dcache-loads/
  L1-dcache-stores OR cpu_atom/L1-dcache-stores/
  L1-icache-loads OR cpu_atom/L1-icache-loads/
  L1-icache-load-misses OR cpu_atom/L1-icache-load-misses/
  LLC-loads OR cpu_atom/LLC-loads/
  LLC-load-misses OR cpu_atom/LLC-load-misses/
  LLC-stores OR cpu_atom/LLC-stores/
  LLC-store-misses OR cpu_atom/LLC-store-misses/
  dTLB-loads OR cpu_atom/dTLB-loads/
  dTLB-load-misses OR cpu_atom/dTLB-load-misses/
  dTLB-stores OR cpu_atom/dTLB-stores/
  dTLB-store-misses OR cpu_atom/dTLB-store-misses/
  iTLB-load-misses OR cpu_atom/iTLB-load-misses/
  branch-loads OR cpu_atom/branch-loads/
  branch-load-misses OR cpu_atom/branch-load-misses/

cpu_core:
  L1-dcache-loads OR cpu_core/L1-dcache-loads/
  L1-dcache-load-misses OR cpu_core/L1-dcache-load-misses/
  L1-dcache-stores OR cpu_core/L1-dcache-stores/
  L1-icache-load-misses OR cpu_core/L1-icache-load-misses/
  LLC-loads OR cpu_core/LLC-loads/
:
```

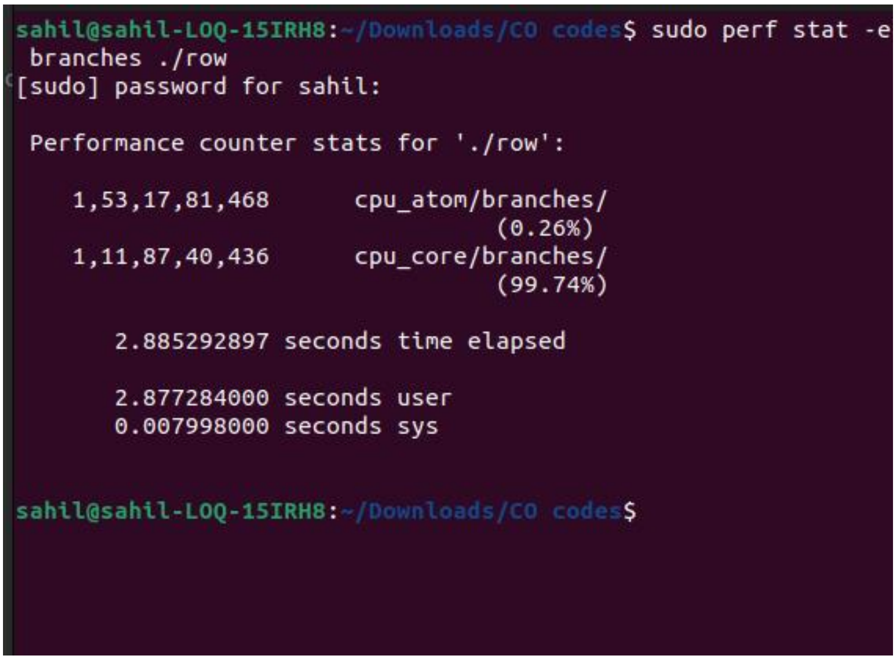
Hardware and Software Events

Hardware Events	Software Events	Tool Events
branch-instructions	alignment-faults	duration_time
branch-misses	bpf-output	user_time
bus-cycles	cgroup-switches	system_time
cache-misses	context-switches OR cs	
cache-references	cpu-clock	
cpu-cycles OR cycles	cpu-migrations OR migrations	
instructions	dummy	
ref-cycles	emulation-faults	
	major-faults	
	minor-faults	
	page-faults OR faults	
	task-clock	

Hardware Events and Their Explanations

1. branch-instructions OR branches

- Measures the number of branch instructions executed, including conditional/unconditional jumps, calls, and returns.
- Example:
  - If a program contains frequent loops and function calls, the branch instruction count will be high.
  - Optimizing loop conditions and minimizing unnecessary function calls can improve performance.



```
sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$ sudo perf stat -e
 branches ./row
[sudo] password for sahil:

 Performance counter stats for './row':

    1,53,17,81,468      cpu_atom/branches/
                                  (0.26%)
    1,11,87,40,436      cpu_core/branches/
                                  (99.74%)

       2.885292897 seconds time elapsed

       2.877284000 seconds user
       0.007998000 seconds sys


sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$
```

2. branch-misses

- Counts the number of times a branch instruction was mis predicted by the processor.
- Example:
    - A program with frequent if-else conditions that do not follow a predictable pattern may lead to high branch-miss rates.
    - Using branch-friendly algorithms like loop unrolling can help reduce mispredictions.

```
sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$ sudo perf stat -e
 branch-misses ./row

 Performance counter stats for './row':

     <not counted>      cpu_atom/branch-misses/
                                       (0.00%)
         11,35,611      cpu_core/branch-misses/


       2.925311258 seconds time elapsed

       2.917274000 seconds user
       0.008000000 seconds sys
```

3. bus-cycles

- Measures the number of cycles the processor spends waiting on the system bus.
- Example:
    - If a program frequently accesses memory or interacts with I/O devices, the bus cycles will increase.
    - Optimizing memory access patterns and reducing unnecessary I/O operations can lower this metric.

```
sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$ sudo perf stat -e
 bus-cycles ./row

 Performance counter stats for './row':

     <not counted>      cpu_atom/bus-cycles/
                                       (0.00%)
    7,16,37,47,175      cpu_core/bus-cycles/


       2.871081934 seconds time elapsed

       2.862924000 seconds user
       0.007999000 seconds sys
```

4. cache-misses

- Counts the number of times the CPU fails to retrieve data from the cache and must fetch it from main memory.
- Example:
  - A program with random memory access patterns (e.g., linked lists with scattered memory locations) can have a high cache-miss rate.
  - Using contiguous memory allocations (like arrays instead of linked lists) can improve cache efficiency.

```
sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$ sudo perf stat -e
 cache-misses ./row

 Performance counter stats for './row':

      11,22,41,040      cpu_atom/cache-misses/
                                          (0.24%)
          6,19,123      cpu_core/cache-misses/
                                          (99.76%)

       2.938253563 seconds time elapsed

       2.932914000 seconds user
       0.007004000 seconds sys
```

5. cache-references

- Counts the number of times the CPU tries to access the cache.
- Example:
  - Programs that frequently read/write data to arrays will have a high cache reference count.
  - Analysing cache references along with cache misses helps determine whether data is being accessed efficiently.

```
sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$ sudo perf stat -e
 cache-references ./row

 Performance counter stats for './row':

     <not counted>      cpu_atom/cache-references/
                                          (0.00%)
       7,05,27,156      cpu_core/cache-references/


       2.917782864 seconds time elapsed

       2.912832000 seconds user
       0.006003000 seconds sys
```

6. CPU-cycles OR cycles

- Measures the total number of CPU cycles executed.
- Example:
    - A computationally intensive program like matrix multiplication will have a high CPU cycle count.
    - Comparing cycles with instructions executed helps calculate IPC (Instructions Per Cycle), a key performance indicator.

```
sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$ sudo perf stat -e
 cpu-cycles ./row

 Performance counter stats for './row':

    4,23,66,32,890      cpu_atom/cpu-cycles/
                                  (0.14%)
   12,87,54,79,671      cpu_core/cpu-cycles/
                                  (99.86%)

       2.942526157 seconds time elapsed

       2.934450000 seconds user
       0.009004000 seconds sys
```

7. instructions

- Counts the total number of instructions executed by the processor.
- Example:
    - A simple loop with a high number of iterations will increase the instruction count.
    - Reducing redundant calculations and optimizing loops can help reduce instruction execution time.

```
sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$ sudo perf stat -e
 instructions ./row

 Performance counter stats for './row':

    3,88,06,59,090      cpu_atom/instructions/
                                  (0.26%)
   52,99,49,48,421      cpu_core/instructions/
                                  (99.74%)

       2.893666539 seconds time elapsed

       2.882548000 seconds user
       0.012002000 seconds sys
```

8. ref-cycles

- Measures reference CPU cycles that are independent of CPU frequency scaling.
- Example:
  - Used when measuring performance across CPUs with dynamic frequency scaling (e.g., laptop CPUs adjusting power for battery efficiency).
  - Helps normalize performance comparisons across different processor clock speeds.

```
sahil@sahil-LOQ-15IRH8:~/Downloads/CO codes$ sudo perf stat -e
 ref-cycles ./row

 Performance counter stats for './row':

    7,32,42,99,134      cpu_atom/ref-cycles/
                                  (0.55%)
    7,32,89,51,357      cpu_core/ref-cycles/
                                  (99.45%)

       2.937102816 seconds time elapsed

       2.931952000 seconds user
       0.005999000 seconds sys
```

Conclusion

Hardware performance monitoring is essential for optimizing system efficiency and diagnosing performance bottlenecks. Using the perf tool, developers can collect valuable insights into CPU behaviour, cache efficiency, and execution patterns, allowing them to make informed optimizations.