Computer Architecture(CSL3020)

Lab Assignment1 - Report

Soumen Kumar
Roll No-B22ES006

1 Introduction

This report presents the performance analysis of a computationally demanding C++ program that sorts an array of integers having a size of 100000 using Bubble Sort. The analysis was conducted using the 'perf' tool on two different Linux machines. The report also includes an evaluation of the performance of Linux commands 'ls' and 'pwd'.

2 Program Overview

2.1 Purpose

The purpose of this program is to demonstrate the performance of a less efficient sorting algorithm, Bubble Sort, on a large dataset (100,000 integers). The Bubble Sort algorithm has a time complexity of $O(n^2)$, which makes it computationally intensive for large inputs. By measuring the time taken to sort the data, we can evaluate the performance characteristics and efficiency of the algorithm.

2.2 Implementation Details

2.2.1 Bubble Sort Algorithm

The `bub_srt` function implements the Bubble Sort algorithm, which sorts an array of integers in ascending order. The algorithm works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process continues until the entire list is sorted. Given its $O(n^2)$ complexity, Bubble Sort is not suitable for large datasets, which makes it ideal for demonstrating the performance impact in this exercise.

2.2.2 Random Array Generation

A vector of size 100,000 is filled with random integers using the `rand()` function. This simulates a large unsorted dataset.

2.2.3 Time Measurement

The program measures the execution time of the Bubble Sort algorithm using the `chrono` library. The start and end times are recorded before and after the sorting operation, respectively. The duration of the sorting process is calculated in milliseconds and converted to seconds for readability.

2.3 Key Points

- **Algorithm Complexity:** $O(n^2)$ - Demonstrates inefficiency on large datasets.
- **Time Measurement:** Uses `chrono` library to record and display the execution time.
- **Random Data:** Generates 100,000 random integers for sorting.

2.4 Expected Output

Upon executing the program, the output should resemble:

```
Array sorted in X.Y seconds
```

where `X.Y` represents the time taken to sort the 100,000 integers using Bubble Sort.

3 Performance Metrics

The following key metrics were analyzed to evaluate the performance of the program:

3.1 Task Clock

The task clock measures the total time the CPU spends executing the program's code. This metric helps in understanding how much actual CPU time was consumed by the task, excluding time spent waiting or idle.

3.2 Context Switches

A context switch occurs when the CPU switches from executing one process to another. Frequent context switches can indicate that the CPU is spending more time switching between processes than executing them, which can degrade performance.

3.3 CPU Migrations

CPU migrations refer to the movement of a process from one CPU core to another. High CPU migration can lead to performance overhead due to cache invalidation and reloading.

3.4 Page Faults

A page fault happens when a program tries to access data that is not currently in physical memory, causing the operating system to retrieve it from disk. High page fault rates can lead to performance bottlenecks due to increased I/O operations.

3.5 Cycles

CPU cycles represent the total number of clock cycles the CPU spent executing instructions. This metric is crucial for understanding the processing power consumed by the program.

3.6 Instructions

This metric counts the number of instructions executed by the CPU. By comparing instructions to cycles, one can gauge the efficiency of instruction execution, commonly referred to as Instructions Per Cycle (IPC).

3.7 Branches

Branches are points in the program where the flow of execution can take different paths, such as loops or conditionals. The number of branches provides insight into the program's control flow complexity.

3.8 Branch Misses

Branch misses occur when the CPU's branch predictor incorrectly guesses the direction of a branch. High branch miss rates can lead to pipeline stalls, reducing overall performance.

4 Experimental Setup

The program was executed on the following Linux machines:

- **Machine 1:** Intel Core i7, 16GB RAM, Ubuntu 24.04
- **Machine 2:** AMD Ryzen 5, 8GB RAM, Kali Linux

5 Performance Analysis

5.1 Performance Analysis Reports

5.1.1 Bubble Sort Performance Metrics

Machine1:

```
# started on Fri Aug 16 16:51:44 2024
 Performance counter stats for './b22es006':

          33799.86 msec task-clock                       #    1.000 CPUs utilized
               169      context-switches                 #    5.000 /sec
                35      cpu-migrations                   #    1.036 /sec
               227      page-faults                      #    6.716 /sec
      112671381046      cpu_atom/cycles/                 #    3.333 GHz                        (0.13%)
      147556410992      cpu_core/cycles/                 #    4.366 GHz                        (99.83%)
      290532353211      cpu_atom/instructions/           #    2.58  insn per cycle             (0.15%)
      439693462430      cpu_core/instructions/           #    3.90  insn per cycle             (99.83%)
       39723055670      cpu_atom/branches/               #    1.175 G/sec                      (0.15%)
       59956932554      cpu_core/branches/               #    1.774 G/sec                      (99.83%)
         923785306      cpu_atom/branch-misses/          #    2.33% of all branches            (0.15%)
        1289399746      cpu_core/branch-misses/          #    3.25% of all branches            (99.83%)
             TopdownL1 (cpu_core)                 #      1.6 %  tma_backend_bound
                                                  #     23.2 %  tma_bad_speculation
                                                  #     19.0 %  tma_frontend_bound
                                                  #     56.2 %  tma_retiring             (99.83%)
             TopdownL1 (cpu_atom)                 #     18.9 %  tma_bad_speculation
                                                  #     54.9 %  tma_retiring             (0.15%)
                                                  #      4.6 %  tma_backend_bound
                                                  #      4.6 %  tma_backend_bound_aux
                                                  #     21.6 %  tma_frontend_bound       (0.15%)

      33.803903602 seconds time elapsed

      33.802051000 seconds user
       0.000000000 seconds sys
```

Machine2:

```
# started on Fri Aug 16 15:44:26 2024


 Performance counter stats for './b22es006':

          7,141.73 msec task-clock                       #    0.993 CPUs utilized
                58      context-switches                 #    8.121 /sec
                 3      cpu-migrations                   #    0.420 /sec
               225      page-faults                      #   31.505 /sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branch-misses

       7.190449065 seconds time elapsed

       7.128663000 seconds user
       0.012756000 seconds sys
```

5.1.2 `ls` Command Performance Metrics

```
# started on Fri Aug 16 17:00:55 2024
 Performance counter stats for 'ls':

              1.61 msec task-clock                       #    0.641 CPUs utilized
                 1      context-switches                 #  622.662 /sec
                 1      cpu-migrations                   #  622.662 /sec
               103      page-faults                      #   64.134 K/sec
           2200148      cpu_atom/cycles/                 #    1.370 GHz                          (40.58%)
           2195876      cpu_core/cycles/                 #    1.367 GHz                          (59.42%)
           1674983      cpu_atom/instructions/           #    0.76  insn per cycle               (40.58%)
           2468461      cpu_core/instructions/           #    1.12  insn per cycle               (59.42%)
            342687      cpu_atom/branches/               #  213.378 M/sec                        (40.58%)
            507479      cpu_core/branches/               #  315.988 M/sec                        (59.42%)
             17117      cpu_atom/branch-misses/          #    4.99% of all branches              (40.58%)
             15420      cpu_core/branch-misses/          #    4.50% of all branches              (59.42%)

       0.002507122 seconds time elapsed

       0.002655000 seconds user
       0.000000000 seconds sys
```

5.1.3 `pwd` Command Performance Metrics

```
# started on Fri Aug 16 17:01:37 2024
 Performance counter stats for 'pwd':

              1.23 msec task-clock                       #    0.361 CPUs utilized
                 1      context-switches                 #  811.194 /sec
                 0      cpu-migrations                   #    0.000 /sec
                80      page-faults                      #   64.896 K/sec
     <not counted>      cpu_atom/cycles/                                                        (0.00%)
           1400682      cpu_core/cycles/                 #    1.136 GHz
     <not counted>      cpu_atom/instructions/                                                  (0.00%)
           1534160      cpu_core/instructions/
     <not counted>      cpu_atom/branches/                                                      (0.00%)
            319713      cpu_core/branches/               #  259.349 M/sec
     <not counted>      cpu_atom/branch-misses/                                                 (0.00%)
              9472      cpu_core/branch-misses/

       0.003412816 seconds time elapsed

       0.000000000 seconds user
       0.002742000 seconds sys
```

5.2 Performance Analysis of Bubble Sort Program

The performance of the Bubble Sort program was analyzed using the `perf` tool on a Linux machine. The following key performance metrics were captured:

- **Task Clock:** 33799.86 msec
- **Context Switches:** 169
- **CPU Migrations:** 35
- **Page Faults:** 227
- **CPU Cycles:**
  - `cpu_atom/cycles/`: 112671381046 cycles
  - `cpu_core/cycles/`: 147556410992 cycles

- **Instructions:**
  - `cpu_atom/instructions/`: 290532353211 instructions
  - `cpu_core/instructions/`: 439693462430 instructions
- **Branches:**
  - `cpu_atom/branches/`: 39723055670 branches
  - `cpu_core/branches/`: 59956932554 branches
- **Branch Misses:**
  - `cpu_atom/branch-misses/`: 923785306 branch misses (**2.33%**)
  - `cpu_core/branch-misses/`: 1289399746 branch misses (**3.25%**)

5.2.1 Analysis of Performance Metrics

Task Clock: The program utilized the CPU for approximately 33.8 seconds. This indicates the overall time the CPU was actively working on the Bubble Sort process.

Context Switches: There were 169 context switches during the execution. Context switches occur when the CPU switches from one process to another, which can introduce overhead but appears minimal in this case.

CPU Migrations: The 35 CPU migrations reflect instances where the process was moved between CPU cores. Frequent migrations can lead to performance penalties due to cache invalidation.

Page Faults: A total of 227 page faults occurred, which implies that some memory accesses required fetching data from disk, possibly due to insufficient physical memory or suboptimal memory management.

CPU Cycles and Instructions:

- The Bubble Sort program executed 112.67 billion `cpu_atom/cycles/` and 147.56 billion `cpu_core/cycles/`.
- The corresponding instruction counts were 290.53 billion `cpu_atom/instructions/` and 439.69 billion `cpu_core/instructions/`.
- The `cpu_atom` and `cpu_core` instruction-per-cycle (IPC) ratios were 2.58 and 3.90, respectively, suggesting that the `cpu_core` executed more instructions per cycle, potentially due to differences in CPU architecture or core optimization.

Branches and Branch Misses:

- The program executed around 39.72 billion `cpu_atom/branches/` and 59.96 billion `cpu_core/branches/`.
- The branch miss rates were 2.33% for `cpu_atom/branches/` and 3.25% for `cpu_core/branches/`. High branch miss rates can indicate inefficient branching logic, leading to pipeline stalls.

5.2.2 Top-Down Analysis (TMA)

TMA for cpu_core:

- 1.6% TMA backend bound
- 23.2% TMA bad speculation
- 19.0% TMA frontend bound
- 56.2% TMA retiring

TMA for cpu_atom:

- 4.6% TMA backend bound
- 4.6% TMA backend bound aux
- 18.9% TMA bad speculation
- 21.6% TMA frontend bound
- 54.9% TMA retiring

The TMA results indicate that a significant portion of the CPU's effort was spent on retiring instructions (56.2% for `cpu_core`), while a notable percentage was lost due to bad speculation (23.2% for `cpu_core`). The frontend bound and backend bound percentages suggest potential bottlenecks in instruction fetching and execution.

5.2.3 Conclusion

The performance of the Bubble Sort program is significantly impacted by its high number of executed instructions, inefficient branching, and the algorithm's inherent $O(n^2)$ complexity. Optimizations such as using a more efficient sorting algorithm (e.g., Quick Sort or Merge Sort) and improving branch prediction can substantially enhance performance. Additionally, reducing context switches and CPU migrations through process scheduling improvements could minimize overhead.

The analysis reveals that CPU cycles, instructions executed, and branch misses are the primary areas consuming resources in the Bubble Sort program, suggesting opportunities for substantial performance gains through algorithmic and architectural optimizations.

Comparison and analysis of the performance metrics revealed that Machine 1 exhibited lower context switches and CPU migrations, indicating more efficient execution. However, Machine 2 had higher stalled cycles, suggesting potential bottlenecks in floating-point calculations.

5.3 Performance Analysis of Linux Commands

5.3.1 Analysis of `ls` Command

The `ls` command was analyzed using the `perf` tool. Below are the key performance metrics observed:

- **Task Clock:** 1.61 msec, indicating the total CPU time consumed.
- **Context Switches:** 1 switch, which shows the number of times the CPU switched between processes.
- **CPU Migrations:** 1 migration, indicating the movement of processes between CPUs.
- **Page Faults:** 103 faults, showing the number of times the system needed to fetch data from disk due to a missing page in memory.
- **Cycles:** 4.39M total cycles split between CPU cores and atoms.
- **Instructions:** 4.14M total instructions executed, with an instruction per cycle (IPC) of 1.12 for core cycles.
- **Branches:** 850K branches were executed, with a 4.50% miss rate.

5.3.2 Analysis of `pwd` Command

The `pwd` command's performance was also analyzed. The key metrics observed are:

- **Task Clock:** 1.23 msec, indicating lower CPU time compared to `ls`.
- **Context Switches:** 1 switch, similar to the `ls` command.
- **CPU Migrations:** 0 migrations, implying no movement of processes between CPUs.
- **Page Faults:** 80 faults, indicating fewer memory-related issues compared to `ls`.
- **Cycles:** 1.40M core cycles.
- **Instructions:** 1.53M instructions executed, without an IPC rate available.
- **Branches:** 319K branches were executed, with a branch miss rate of around 2.96%.

5.3.3 Comparison and Observations

The `ls` command consumes more CPU cycles and executes more instructions compared to the `pwd` command. This is expected since `ls` lists directory contents, which is a more complex operation than `pwd`. Additionally, the `ls` command exhibits a higher branch miss rate, suggesting more conditional logic is involved.

6 Identifying Potential Bottlenecks and Areas for Optimization

Based on the performance metrics analyzed using `perf`, the following potential bottlenecks and areas for optimization have been identified in the Bubble Sort program:

6.1 High CPU Cycles and Instructions Count

Issue: The Bubble Sort algorithm has a time complexity of $O(n^2)$, resulting in a high number of CPU cycles and instructions executed, especially for large datasets like the 100,000 elements used in this program. The algorithm performs repeated comparisons and swaps, leading to inefficient use of CPU resources.

Optimization:

- *Algorithmic Improvement:* Replace Bubble Sort with a more efficient sorting algorithm, such as Quick Sort or Merge Sort, which have a time complexity of $O(n \log n)$. This would significantly reduce the number of instructions executed and CPU cycles required, improving overall performance.

6.2 Frequent Context Switches

Issue: If the program shows a high number of context switches, it could indicate that the CPU is frequently switching between processes, leading to reduced efficiency as it spends more time managing context switches rather than executing the program.

Optimization:

- *Process Prioritization:* Ensure that the program is given a higher priority for CPU time, especially if it's running on a multi-tasking system. This can be done by adjusting the scheduling policy or using `nice` and `renice` commands to reduce the impact of context switches.

6.3 Branch Misses

Issue: The Bubble Sort algorithm involves many conditional checks (if statements), leading to branch instructions. A high rate of branch misses indicates that the CPU's branch predictor is frequently guessing wrong, causing pipeline stalls and wasted cycles.

Optimization:

- *Algorithmic Optimization:* Optimize the inner loop by introducing an early exit condition when the list is already sorted, reducing the number of unnecessary comparisons and branch instructions.

- Alternatively, use a different sorting algorithm that involves fewer branches or is more predictable in terms of branch behavior, which could reduce branch misses.

6.4 Page Faults

Issue: High page fault rates can occur if the dataset size exceeds the available physical memory, causing frequent data retrieval from disk. This is especially a concern with very large datasets.

Optimization:

- *Memory Management:* Ensure that the system has enough physical memory to handle the dataset without causing excessive page faults. Consider optimizing the data structures used or breaking down the dataset into smaller chunks that fit comfortably in memory.

- Additionally, using memory-mapped files or adjusting the system's swappiness can better manage how data is paged.

6.5 CPU Migrations

Issue: If CPU migrations are frequent, they can lead to performance overheads due to cache invalidation and the need to reload data into the new CPU's cache.

Optimization:

- *Pinning Threads to CPUs:* Consider pinning the process or specific threads to a particular CPU core using `taskset` or similar tools to reduce the number of CPU migrations. This can help in maintaining cache locality and reducing the overhead associated with migrations.

6.6 Resource-Intensive Computational Aspect

The most resource-consuming aspect of the function is the nested loop structure of the Bubble Sort algorithm, which results in a quadratic number of comparisons and swaps. Each iteration of the outer loop involves a full pass over the unsorted portion of the array, leading to $O(n^2)$ operations. This is inefficient for large datasets, where the number of operations grows rapidly, consuming excessive CPU cycles and memory bandwidth.

6.7 Conclusion and Recommendations

- **Algorithm Replacement:** The primary optimization needed is to replace Bubble Sort with a more efficient sorting algorithm like Quick Sort or Merge Sort. This will dramatically reduce the number of CPU cycles, instructions executed, and overall execution time.
- **Process Optimization:** Reduce context switches and CPU migrations by managing the process scheduling and affinity, ensuring the program gets sufficient uninterrupted CPU time.
- **Memory Optimization:** Minimize page faults by ensuring adequate physical memory or optimizing data structures and memory usage.

These optimizations should significantly enhance the program's performance, making it more efficient in terms of CPU usage, execution time, and overall resource consumption.

7 Conclusion

The 'perf' tool effectively highlighted performance differences between the two machines, providing insights into CPU utilization and potential optimization areas. Future improvements could include optimizing the bubble sort algorithm by exploring more efficient sorting algorithms like quicksort or mergesort, and fine-tuning the implementation to reduce unnecessary comparisons and swaps for better instruction efficiency..

8 Resources Used:-

- **Perf Wiki**: The official documentation and community-maintained wiki for `perf`, which covers installation, usage, and advanced topics.
- **Brendan Gregg's Perf Examples**: A comprehensive guide by Brendan Gregg that provides practical examples and explanations on using `perf`.
- **Perf Man Page**: The manual page for `perf`, offering a detailed breakdown of available commands and options.