**CPU Bench Mark**

**David Tran**

**CSC 641**

**Introduction**

The purpose of this assignment is to create a bench mark program which uses a combination of numerical operations and combinatorial operations to test the speed of a CPU (Processor) in a computer system. We try to simulate a realistic workload that CPUs we use today are put under during regular use. Since CPUs today currently perform using multiple cores, I will test the CPU under a full workload testing the CPU using 1 core to double the CPU’s number of cores (Ex. Cores=2 means there will be 4 tests at the same time). Each type of workload (Combinatorial and Numerical) will run for 10 seconds each totaling a whole 20 seconds for the entire program. By doing this it allows us to get a more accurate reading of the CPU’s speed. I will be testing the bench mark on two different systems with two different CPUs.

**Combinatorial Workload**

A combinatorial workload focuses on using integer and control operations on the CPU. It deals with integers operations. A combinatorial workload can be simulating using operations such as: Sort, Search, Integer array processing, and more. For my bench mark I decided to use **Quick Sort,** which used a quicksort algorithm to sort a giant array of integers. The array was populated using **srand()** with a seed allowing me to create the same sequence of numbers for more precise testing. I made sure the quick sort function ran for at least 1 second before using a loop to run it for 10 seconds. To do this I had to adjust the size of the array that was to be sorted and use a loop to run the function, by doing this it allowed the function to run for 1 second.

**Numerical Workload**

A numerical workload focuses mainly on numerical operations on the CPU. This is primarily floating point operations. The CPU is put under floating point operations load where you can use operations such as: Matrix Inversion, Matrix Multiplication, Numerical Integration, and much more. For my bench mark I decided to use **Matrix Inversion** operation, which takes a matrix populated with real numbers and inverts it. The numbers populated in the matrix was chosen specifically, so we don’t have a randomization within the matrix. The matrix is the same before each time it is inversed; this allows us to get more accurate results without randomness. I made sure the function ran for at least 1 second before allowing it to run for a total of 10 seconds using a loop.

**Experiments**

The benchmark program was written in C++ and compiled using Cygwin Terminal G++ compiler. The program was compiled using the highest Optimization level 3.

**1st system:** Custom PC Tower

**CPU:** Intel i7-960 @3.20GHz

**Cores:** 4

**Operating System:** Windows 10

**Data System 1:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **N (Run #)** | **v1** | **v2** | **v3** | **v4** | **v5** | **v6** | **v7** | **v8** | **Total Speed** |
| 1 | 65.45 |  |  |  |  |  |  |  | 65.45 |
| 2 | 61.71 | 61.71 |  |  |  |  |  |  | 123.42 |
| 3 | 55.58 | 53.05 | 55.58 |  |  |  |  |  | 164.21 |
| 4 | 43.75 | 43.75 | 43.75 | 43.75 |  |  |  |  | 175 |
| 5 | 34.29 | 31.08 | 33.23 | 32 | 31.08 |  |  |  | 161.68 |
| 6 | 25.49 | 19.2 | 26.18 | 25.12 | 27 | 26.18 |  |  | 149.17 |
| 7 | 16.15 | 17.45 | 18.82 | 16.15 | 16.15 | 16.55 | 16.88 |  | 118.15 |
| 8 | 9.655 | 9.655 | 9.655 | 9.655 | 9.655 | 9.655 | 9.655 | 9.655 | 77.24 |

**Figure 1.1**

The table here in Figure 1.1 shows the experimental results for the 1st system. The CPU has a total of 4 cores, so I ran tests up to 8 executions of the bench mark at the same time (1-8) forcing the CPU’s workload. V1-V8 represents the speed per execution. Total speed is the sum of all speeds during that run.

**Figure 1.2**

Figure 1.2 is a graph which represents the total speed of the CPU as the number of runs (meaning open executions of the program at the same time) is increased.

**Figure 1.3**

Figure 1.3 is a graph which represents the total speed as the number of cores used is increased.

**2nd system:** Gaming Laptop

**CPU:** Intel i7-4510U @2.60GHz

**Cores:** 2

**Operating System:** Windows 10

**Data System 2:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **N(Run#)** | **v1** | **v2** | **v3** | **v4** | **Total Speed** |
| 1 | 96 |  |  |  | 96 |
| 2 | 74.88 | 74.88 |  |  | 149.76 |
| 3 | 45 | 53.33 | 51.33 |  | 149.66 |
| 4 | 16 | 16 | 16 | 16 | 64 |

**Figure 2.1**

The table here in Figure 2.1 shows the experimental results for the 2nd system. The CPU has a total of 2 cores, so I ran tests up to 4 executions of the bench mark at the same time (1-4) forcing the CPU’s workload. V1-V4 represents the speed per execution. Total speed is the sum of all speeds during that run.

**Figure 2.2**

Figure 2.2 is a graph which represents the total speed of the CPU as the number of runs (meaning open executions of the program at the same time) is increased.

**References**

**1.** AlgoList

I used this website as a reference for finding an efficient quicksort program.

**URL:** <http://www.algolist.net/Algorithms/Sorting/Quicksort>

**2.** CSC 641 Reader Dr. Jozo Dujmovic

I used the CSC641 Reader to get the function for Matrix Inversion.

**Page:** 277

**Appendix**







