CSC 641 Computer Performance Analysis

BenchMark Program

Andrey Barsukov

Table of content Page Number

I) About the BenchMark ………………………………………………………………………………………………. 2

II) How BenchMark number is calculated ………………………………………………………………….…. 4

III) Test runs …………………………………………………………………………………………………………………. 5

1) Main system SP3 i5 ……………………………………………………………………………………… 7

-False data ………………………………………………………………………………………….. 8

-release vs debug ………………………………………………………………………………… 9

2) Intel i5 – Linux …………………………………………………………………………………………..… 10

3) Intel i7 …………………………………………………………………………………………………………. 11

4) Intel i5 desktop ……………………………………………………………………………………………. 12

5) AMD Athelon II ……………………………………………………………………………………...……. 13

IV) Combined Data ………………………………………………………………………………………………...……. 14

V) Conclusion …………………………………………………………………………………………………………….…. 15

VI) C++ Code ………………………………………………………………………………………………………………… 16

Andrey Barsukov

CSC 641 CPE HW1

09/23/2015

**I)** About:

This is a BenchMark program written in C++ using Dev C++ IDE on Window 8.1 OS. To minimize overhead there I did not use separate compilation. This program is designed to run for 20+ seconds performing various operations.

10 seconds to run integer operations:

Optimized to run about 3 times per second on a Surface Pro 3, this operation sorts an 11k element array using an inefficient sort.

10 seconds to run double precision operations:

Optimized to run about 3 times per second on a Surface Pro 3, this operation does matrix multiplication on a 200x200 2D array initiated with **double**s

+ ~0.3 seconds for overhead

Why Bubble Sort:

Total time complexity: n^2 + 2n for n = 11000

Bubble Sort is a simple and well known sorting algorithm. It is generally used only to introduce the concept of sorting and almost never in practice. This sort belongs to the complexity of O(n^2) making it very ineffective for large n.

I chose this sort because it is stable. It is a good representation of integer operations. Since I am populating the array myself I can ensure that the worst case of O(n^2) operations is always performed. In this BenchMark I initiate an array of size 11,000 in a specific unsorted order. Using a **forloop** I fill this array with a repeating sequence of 10, 9, 8, 7,.…, 1, 0 until the array is full.

There are 3 operations performed in this function.

1) n operations

Fill the array with decreasing integers 10-0

2) n^2 operations

Bubble sort the full array increasing order

3) n operations

Go through the entire array check that is it fully sorted

Why Matrix Multiplication:

Total time complexity: 5(n^3 + n^2) for n = 200

In the matrix multiplication function there are two main operations performed.

1) 5n^2 operations

Initializing the matrix using a double nested **forloop.**

2) 5n^3 operations

Performing the multiplication 5 times using a triple nested **forloop.**

Matrix multiplication is a good representation of **double** operations for several reasons.

-First, there are 3 [200]x[200] array is 320,000x3 = 960,000 bytes or .9155 MB. This ensures that at least some of the data will be stored in a slower L3 cache.

-Another reason is that I can populate the matrix with predetermined non-random **doubles** and ensure similar performance on all platforms

After each matrix multiplication iteration the last element is added to a variable **sumOfLast** to ensure that the complier is not simplifying the code and throwing away the loop.

**II)** How BenchMark number is calculated

**#include <time.h>**  //this library allows me to retrieve the time that an operation runs for.

//set a specified run time of operations

Calculating clock resolution:

T1 = T2 = sec( ); //set T1 and T2 to equal each other by calling a function which returns the current time

// Wait for clock change

while(T2 == T1) T2 = sec( );

r = T2-T1 ; //r is the clock resolution in milliseconds

Running INTEGER or DOUBLE for 10 seconds:

START = sec(); //start time count

while(sec() < START + 10){ //will run the loop for 10 seconds

INTEGER(); or DOUBLE();

Nops ++;

}

After both INTEGER and DOUBLE operations have ran for 10 seconds each I calculate their corresponding run times

Vint = 60\*Nint/ (sec() - START);

Vfloat = 60\*Nfloat / (sec() - START);

Nint/Nfloat //number of operations performed in 10 seconds

Vint/Vfloat //number of operations per min

Then using number of operations per min I calculate the harmonic mean of integer and double.

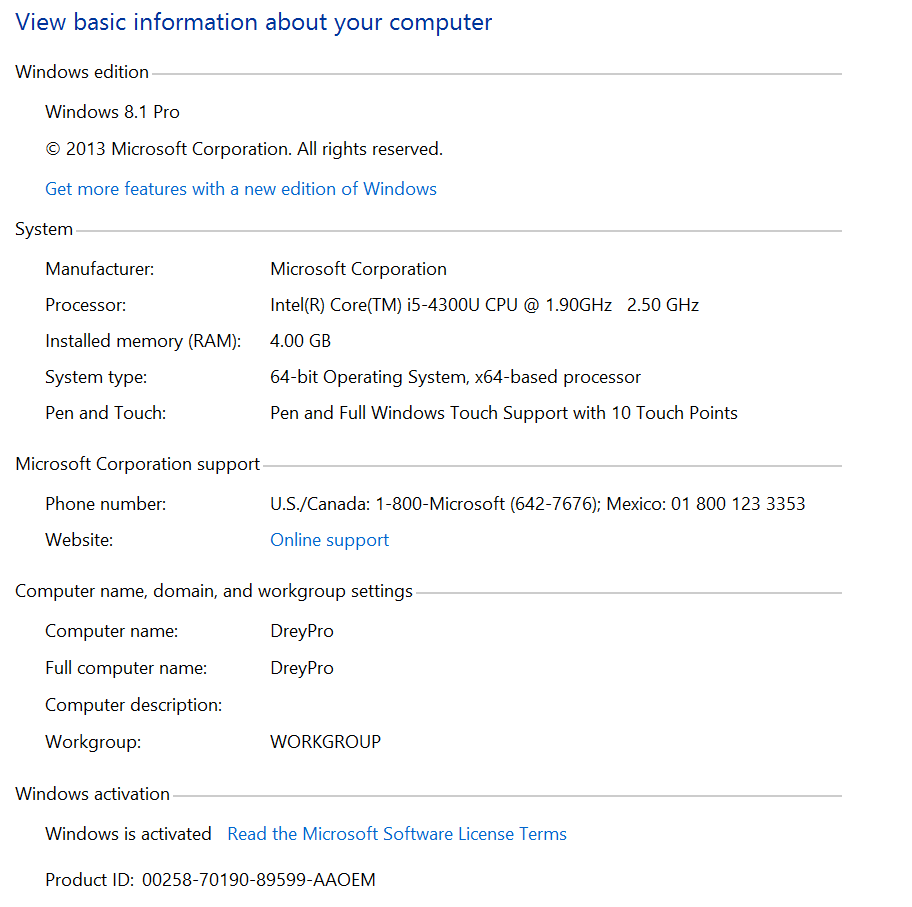
SM = (2 \* Vint \* Vfloat) / (Vint + Vfloat);

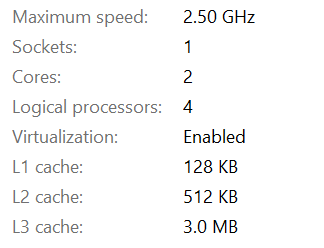
SM //this is the final BenchMark score of a machine

**III) Test run 1**

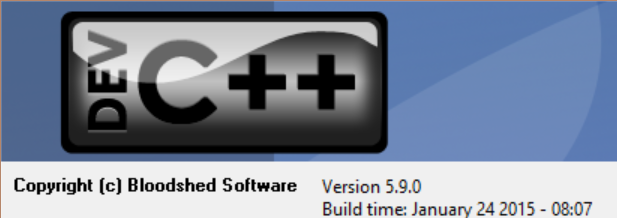
The first test is performed on my own machine. Surface Pro 3.

System info:





Single BenchMark.exe release version run 1:



Compilation results...

--------

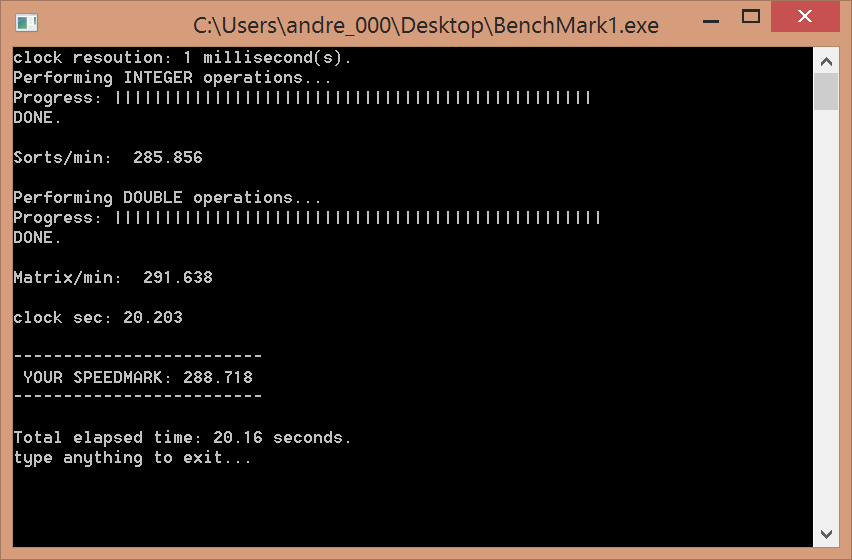
- Errors: 0

- Warnings: 0

- Output Filename: C:\Users\andre\_000\Documents\Github\SchoolWork\BenchMark.exe

- Output Size: 1.82814311981201 MiB

- Compilation Time: 0.16s



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| System 1 - Surface Pro 3 | | |  |  |  |  |  |  |  |
| BMs ran | BM1 | BM2 | BM3 | BM4 | BM5 | BM6 | BM7 | BM8 | Total |
| 1 | 287 |  |  |  |  |  |  |  | 287 |
| 2 | 247 | 247 |  |  |  |  |  |  | 494 |
| 3 | 184 | 181 | 150 |  |  |  |  |  | 515 |
| 4 | 127 | 127 | 126 | 126 |  |  |  |  | 506 |
| 5 | 120 | 97 | 97 | 97 | 96 |  |  |  | 507 |
| 6 | 114 | 86 | 82 | 80 | 79 | 79 |  |  | 520 |
| 7 | 108 | 72 | 70 | 69 | 67 | 67 | 64 |  | 517 |
| 8 | 102 | 62 | 61 | 57 | 57 | 56 | 56 | 56 | 507 |

Data Analysis:

This is the first BenchMark test.

The BenchMark was compiled on Windows 8.1 OS using DevC++ release build .exe

The performance we get just one BenchMark is 287. This is the score that the program was designed around.

2 physical cores

With two iterations the performance is drastically increased

494/287 = 1.72 or a **72% increase in performance**

Before I ran this test I made a prediction that performance would be close to double when both core are working, but sadly that is not the case.

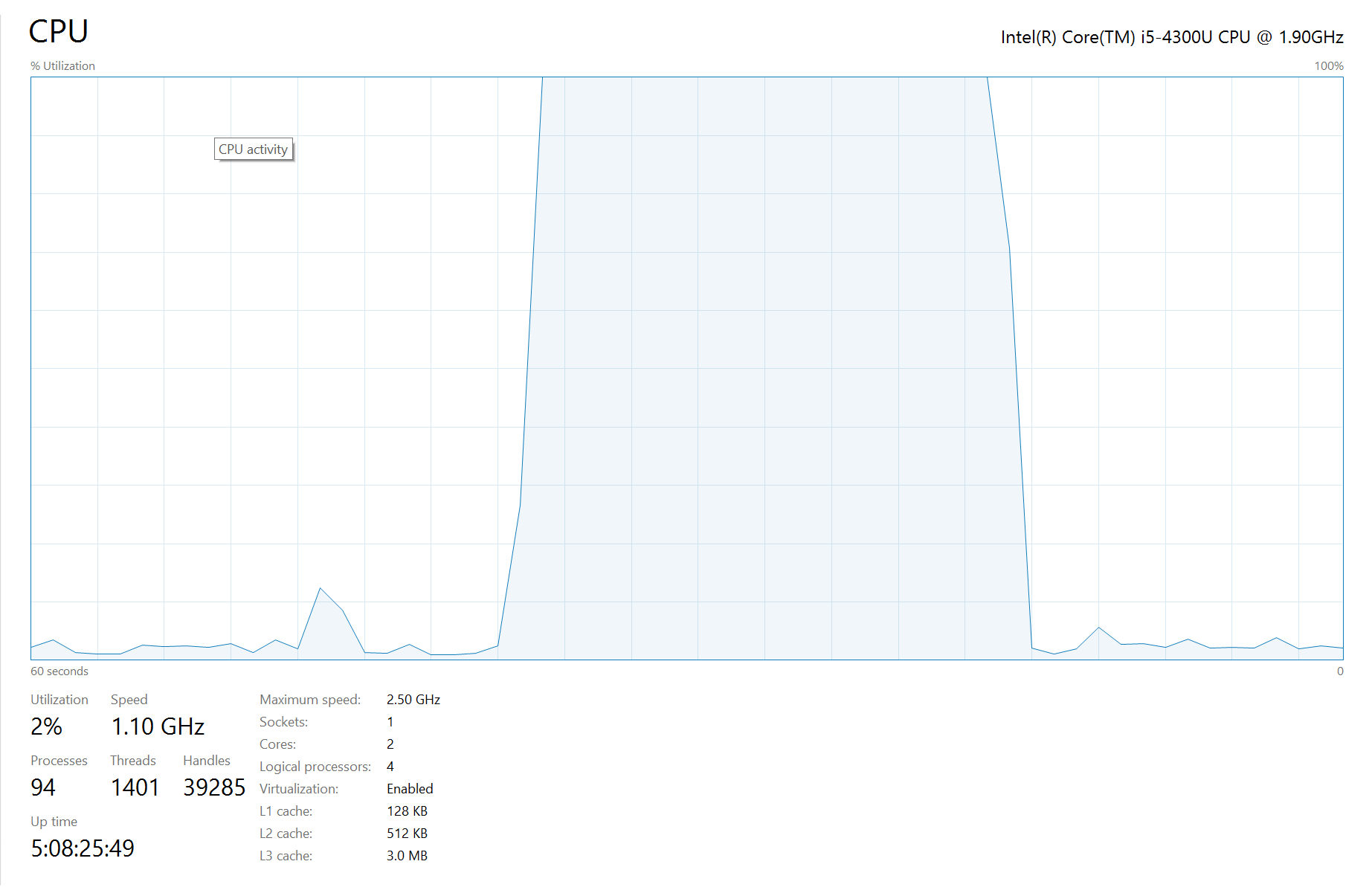
**False data for 3+ BenchMark runs:**

As we run more BenchMarks we can see that the performance plateaus at around 500. Even though we see a slight bump in performance from 2 to 3 BenchMarks, this is false data. At 3+ BenchMark tests the computer cannot start the programs exactly at the same time. This means that the last one or two programs will start later than the rest, giving them time to run alone at for a few seconds at the end.

We can clearly see this false data in the first column (BM1) of the data table. The BenchMark score of one BenchMark is almost double of the rest. Taking this into account we can safely assume that there is no increased performance after two BenchMarks.

Some more proof of false data for 3+ BenchMarks can be seen by looking at the CPU performance during execution of 3 or more BenchMarks.

Here is a snapshot of CPU performance when 8 BenchMarks are ran:



Each square on the graph is 2 seconds

We can see that it takes the CPU about two seconds to become fully utilized and then another two seconds after the first program finished to go back to its original state. This shows that not all programs start or finished together.

Release vs Debug versions of BenchMark:

While compiling the BenchMark program there is an option to create a release .exe version with only my code and no extra error handing. This release version is shown in the test runs above.

There is also an option to create a debug .exe version of the same program. This contains compiler generated check and error handling. As assumed the extra code does take away from the performance of the program.

Data for Debug version:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Surface Pro 3 Debug .exe | | |  |  |  |  |  |  |  |
| BMs ran | BM1 | BM2 | BM3 | BM4 | BM5 | BM6 | BM7 | BM8 | Total score |
| 1 | 263 |  |  |  |  |  |  |  | 263 |
| 2 | 209 | 209 |  |  |  |  |  |  | 418 |
| 3 | 160 | 164 | 169 |  |  |  |  |  | 493 |
| 4 | 120 | 120 | 126 | 120 |  |  |  |  | 486 |
| 5 | 91 | 91 | 91 | 93 | 120 |  |  |  | 486 |
| 6 | 78 | 78 | 77 | 76 | 73 | 111 |  |  | 493 |
| 7 | 67 | 64 | 67 | 65 | 63 | 67 | 107 |  | 500 |
| 8 | 53 | 55 | 58 | 59 | 60 | 53 | 55 | 95 | 488 |

Not a significant difference but the difference is there nevertheless.

Testing competitive computers:

**Test 2: Lenovo Laptop running Linux**

This test was ran from an executable BenchMark program compiled using g++ in Linux.

System specs:

Clock Resolution: 1 mSec

CPU: i5 @ 1.70 GHz

RAM: 8.00 GB

64-bit OS, x64-based processor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BMs ran** | **BM1** | **BM2** | **BM3** | **BM4** | **Total** |
| **1** | **269** |  |  |  | **269** |
| **2** | **203** | **204** |  |  | **407** |
| **3** | **163** | **164** | **165** |  | **492** |
| **4** | **121** | **126** | **119** | **120** | **486** |

This was a friend’s computer, so I was not able to run more than 4 tests. But from the data I have is it evident that his system maxes out at score of ~500.

**Test 3: Lenovo ThinkPad Laptop**

Compiler: DevC++ (release .exe)

System specs:

Clock resolution: 15 mSec

OS: Windows 7 Pro SP1

CPU: i7 @2.50 GHz

RAM: 16.0 GB

System type: 64-bit OS

**Test data:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BMs ran | BM1 | BM2 | BM3 | BM4 | BM5 | BM6 | BM7 | BM8 | Total |
| 1 | 334 |  |  |  |  |  |  |  | 334 |
| 2 | 304 | 304 |  |  |  |  |  |  | 608 |
| 3 | 264 | 269 | 263 |  |  |  |  |  | 796 |
| 4 | 233 | 247 | 239 | 228 |  |  |  |  | 947 |
| 5 | 214 | 220 | 211 | 218 | 213 |  |  |  | 1076 |
| 6 | 190 | 189 | 193 | 188 | 189 | 189 |  |  | 1138 |
| 7 | 165 | 164 | 164 | 162 | 168 | 181 | 164 |  | 1168 |
| 8 | 154 | 163 | 154 | 157 | 154 | 147 | 148 | 156 | 1233 |
| 12 |  |  |  |  |  |  |  |  | 1230 |

This system has 8 cores, this can be seen in the data. The performance does not get capped until we reach 8+ BenchMarks. The curve does not seem to be linear however.

**Test 4: DELL PC (SFSU Csc lab PC)**

Compiler: DevC++ (release .exe)

System specs:

Clock resolution: 15 mSec

OS: Windows 7 Enterprise

CPU: i5 @ 3.30 GHz

System type: 64-bit OS

RAM: 4.0 GB

**Test data:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BMs ran | BM1 | BM2 | BM3 | BM4 | BM5 | BM6 | BM7 | BM8 | Total |
| 1 | 309 |  |  |  |  |  |  |  | 309 |
| 2 | 307 | 307 |  |  |  |  |  |  | 614 |
| 3 | 304 | 304 | 304 |  |  |  |  |  | 912 |
| 4 | 283 | 290 | 295 | 288 |  |  |  |  | 1156 |
| 5 | 264 | 219 | 237 | 216 | 236 |  |  |  | 1172 |
| 6 | 211 | 237 | 121 | 231 | 134 | 231 |  |  | 1165 |
| 7 | 258 | 213 | 107 | 101 | 101 | 191 | 210 |  | 1181 |
| 8 | 253 | 69 | 165 | 150 | 153 | 176 | 75 | 158 | 1199 |

This test run shows a very nice curve. The desktop performs just how we would expect a 4 core processor to perform. It increases linearly until 4 BenchMarks and then stays almost perfectly equal for the remaining runs. There is also some interesting processor optimization going on at 6+ BenchMarks where the score for the BenchMarks differs more drastically more BenchMarks are ran.

**Test 5: Acer AMD Athelon II x2 Duel core**

Compiler: DevC++

System specs:

Clock resolution: 1 mSec

OS: Windows 7 SP1

CPU: AMD Athelon II x2 Duel core

System type: 64-bit

RAM 4.0 GB

DATA:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BMs ran | BM1 | BM2 | BM3 | BM4 | BM5 | BM6 | BM7 | BM8 | Total |
| 1 | 113 |  |  |  |  |  |  |  | 113 |
| 2 | 124 | 126 |  |  |  |  |  |  | 250 |
| 3 | 57 | 98 | 100 |  |  |  |  |  | 255 |
| 4 | 117 | 30 | 30 | 75 |  |  |  |  | 252 |
| 5 | 94 | 42 | 42 | 42 | 35 |  |  |  | 255 |
| 6 | 126 | 26 | 25 | 25 | 25 | 26 |  |  | 253 |
| 7 | 63 | 23 | 25 | 46 | 45 | 25 | 42 |  | 269 |
| 8 | 37 | 36 | 23 | 35 | 35 | 62 | 23 | 23 | 274 |

This test was run on a machine with an AMD processor. The performance was noticeably slower than similar Intel machines. Here again we notice some false data as we get into 6+ BenchMarks ran.

**Combined Data: Comparisons**

After performing the tests we can combine all of the gather data and see the results of different systems compared to each other.

Here we set the slowest machine to have a value of one (1) and calculate the performance of all other machines relative to the slowest one.

**Conclusion:**

Besides the data shown above, I noticed a few other interesting things while running the test

* In laptops, performance decreases graduate as the batter life decreases, up to a maximum of about a 30% decrease in performance
* The process manager in most computers decreases in its ability to evenly break up processor load as the number of processes increases. This is especially noticeable in the DELL i5 desktop where the different of performance from core to core is as drastic as 253 and 69 between two different cores.
* Performance between debug and release versions of the program does not seem to make a noticeable difference
* A system with Linux OS has a slightly worse performance than Windows, running on a similar intel i5 processor
* While using DevC++ compiler, I noticed that it did not optimize my code even without explicitly making sure that all data is actually calculated and redundant loops are not removed.

Overall this project was interesting and very educational. After this project I have a much better understanding of computer performance and multi core performance. A system’s maximum performance is heavily dependent on the amount of physical and virtual cores it has. Although multiple cores are supposed to double the performance, this is not always the case.

This project has peaked my interest in Computer Performance Analysis, and I will be doing more research into this topic.