Aryan Gupta
ECGR4181

These benchmarks were run on 2 possible machines:

- x86_64 | Lenovo IdeaPad 320 Laptop | Intel Pentium N4200 1.10 GHz Quad Core
- ARM | Samsung Galaxy Tab S | Exynos 5 Octa Processor (1.9 GHz Quad Core + 1.3 GHz Quad Core)

The only bench mark run on the ARM machine was the Drystone benchmark. All data presented are averages taken over 5 run times unless stated otherwise

I believe that the first time that the Drystone program ran was slower than the rest is because of caching. The first time the Drystone the computer does not know how the program will behave, but after the program has been run a few times, Windows will pick up what the program is doing and cache the data so it runs faster.

Running the Drystone program with no optimizations lead to low results. I would get 1000000 to 2000000 Drystones per second. After running a few times, the result equalizes to 2000000 Drystones per second. With -O0 optimizations the program runs about the same speed (2000000 Drystones per second). When run with -O1, the Drystone count increases to 6000000 Drystones per second and stays there for -O2 and -O3

X86_64	**-O3**	**-O2**	**-O1**	**-O0**
	6000000	6000000	6000000	2000000

I believe that -O1 offered the program the biggest optimizations that is optimal for this ISA.

I believe that different ISA's are optimized for different things. And because of this fact one ISA can vastly outperform another. Take x86/AMD64 and ARM. Their memory models are very different. x86/64 (hereinafter referred to as x86) is a very sequential and strongly memory ordered machine. Meaning that each memory access has implicit acquire/release schematics. Compared to x86, ARM is a weakly ordered machine and the programmer must explicitly (LDA, STL, DMB) apply these fences to prevent errors. If an application relying on the implicit schematics in x86, were to be compiled on an ARM and run, the performance on the machine would suffer greatly because now each load must be accompanied with a barrier.

Similarly, to the memory model example, if one ISA is highly optimized for branching and has a robust branch predictor, then it may execute loops and other branch instructions better than another. And this does not have to be with branch instructions, one ISA may have opcodes that can better handle a specific task than another. For example, NVIDIA uses an ISA called PTX or NVPTX for their CUDA GPU's, this ISA is highly optimized for running parallel tasks and would excel in applications like Computer Graphics, bit-coin hashing/mining, cryptography, and more. However, if any of these tasks were to be run on an Intel i7, the performance hit would be catastrophic, because the ISA the i7 uses (AMD64) is not a very parallelized ISA.

After running the Drystone benchmark, I came by with peculiar results. All the optimizations seem to have the proper effect on the scores. Moving from -O1 to -O2 had the biggest jump, almost doubling our scores. The peculiar result was -O3. It seemed to crash the program, sometimes it would just display 0

Aryan Gupta
ECGR4181

Drystones / second and other times it would display 6000000. Compared to the LINPAC benchmark, this benchmark seemed to have a very confusing output.

ARM	-O3	-O2	-O1	-O0
	----------	6000000	3000000	3000000

I was unable to get a profiler running on my Android table (ARM). I was getting a linker error: undefined reference to `mcount'. Even after some time on Google, I was not able to find a solution. I was also unable to get gprof working even on my laptop. The data seemed to be incorrect. The data in gprof told me that the function ran 0% of the time:

```
index % time    self  children    called          name
                0.00    0.00    100000/100000      Proc0 [5]
[1]      0.0    0.00    0.00    100000          Proc1 [1]
```

The LINPAC benchmark seemed to be more robust and more understandable. After running it on my laptop, I was returned with these results. I will be only documenting the Megaflops portion of the benchmark:

X86_64	NO -O FLAG	-O3	-O2	-O1	-O0
	213.973333	873.360544	807.446541	713.244444	205.743590

Just as it was seen in the Drystone benchmark, jumping from -O0 to -O1 offered the largest performance improvement. One of the common optimizations I see from Google is *-ffast-math*. This optimization offers performance while comprising safety. For example, it will treat all math operations as finite and not check for NaN or INF. The optimization could also reorder instructions to something that would execute faster, but may not be numerically accurate (but will be close enough). From GCC's website: These optimizations are *set -fno-math-errno, -funsafe-math-optimizations, -ffinite-math-only, -fno-rounding-math, -fno-signaling-nans and -fcx-limited-range*.

http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

https://stackoverflow.com/questions/7420665/what-does-gccs-ffast-math-actually-do

For my 3rd benchmark, I had difficulty finding a benchmark that would not only compile but also have a clean output. I chose a benchmark that I used to test the new Mosaic Linux servers with 128 cores and 1TB of RAM. This benchmark was created by me few years ago when I was learning to code multi-threaded applications. The source can be found here:

https://github.com/aryan-gupta/PrimeMap

Aryan Gupta
ECGR4181

This program finds prime numbers and outputs them into a nice bitmap. I have added command line options to run with variable amount of threads. When I wrote this program, I was very ignorant in the ways of creating make files so I decided it would be best if I manually compiled it:

```
g++ PrimeMap.cpp -o PrimeMap.out -lpthread -O0
```

and ran the benchmark with

```
./PrimeMap.out -t 4 -m 500000
```

After running the benchmark, I got gprof working and quickly realized that there is heavy inlining in gcc optimizations. Here are my results in seconds:

X6_64	-O3	-O2	-O1	-O0
	8.7071 seconds	8.99325s seconds	9.6014 seconds	11.1919 seconds

As you can see again the largest jump in performance was -O0 to -O1

This benchmark benchmarks the capabilities of the integer ALU, most of the code in this benchmark is looping and testing using the mod operator. This system would run better if ran on a Thread Ripper CPU from AMD. These processors have large number of cores (16-32 cores). Another architecture that this could run great on would be a GPU. I have done a bit of OpenGL programming and I am confident that it isn't much different from OpenCL (GPU/parallel computation framework).

For the 2 suits that are used to characterize a processor, I chose EEMBC Benchmark Suite and ParMiBench.

The EEMBC is a general benchmarking suit that "attempts to represent user programs from several embedded disciplines". This allows the benchmark to give a good idea how an average user application will perform on different processors. Because the suit has many components to it like Networking, Office, and Digital Entertainments, the metrics used to benchmark varies between the component used to benchmark.

1) https://ieeexplore.ieee.org/document/5325153/

2) https://ieeexplore.ieee.org/document/1430553/

The other suit is ParMiBench, this benchmark is aimed toward microprocessors that have multiple cores or can concurrently run threads. The benchmark is derived from the popular benchmark MiBench. Similarly, to the EEMBC, the ParMiBench has various sub-benchmarks that tests the performance of different things and the metrics vary.

1) https://ieeexplore.ieee.org/document/5550920/

2) https://ieeexplore.ieee.org/document/8250026/