Lab 0: Benchmarking

Nikhil Goyal
2015CS50287

January 18, 2019

gcc/g++/gfortran - Version 7.3.0

clang - Version 6.0.0

icc/icpc/ifort - Version 19.0.1

S.No.	Benchmark	Language
1	500.perlbench_r	C
2	502.gcc_r	C
3	505.mcf_r	C
4	520.omnetpp_r	C++
5	523.xalancbmk_r	C++
6	525.x264_r	C
7	531.deepsjeng_r	C++
8	541.leela_r	C++
9	548.exchange2_r	Fortran 95
10	557.xz_r	C

Table 1: Benchmark numbers used in report and their corresponding language

1

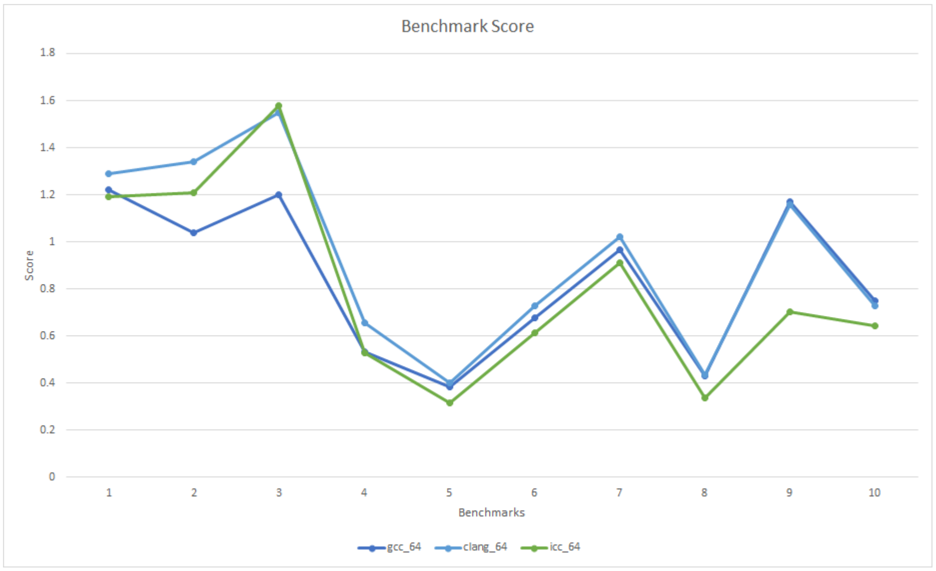

Figure 1: Scores on 10 benchmarks for gcc, clang and icc

Clang produces the fastest unoptimized code in almost all the benchmarks. Gcc/G++ unoptimized builds run comparable to that of clang and in most cases difference is with the margin of experimental error. In 2nd and 3rd benchmark (both written in C) GCC performs significantly worse than clang and icc. It could have happened that GGC doesn't produce optimized code by default while the others did. Intel's compiler produces slightly slower binaries than gcc and clang in most benchmarks and scores very low on the 9th benchmark. It is written in Fortran 95 and as we can see in Fig.1, ifort under performs in comparison to gfortran. However we cannot generalize this behavior as it's the only benchmark written in Fortran in the intrate benchmark suite.

2

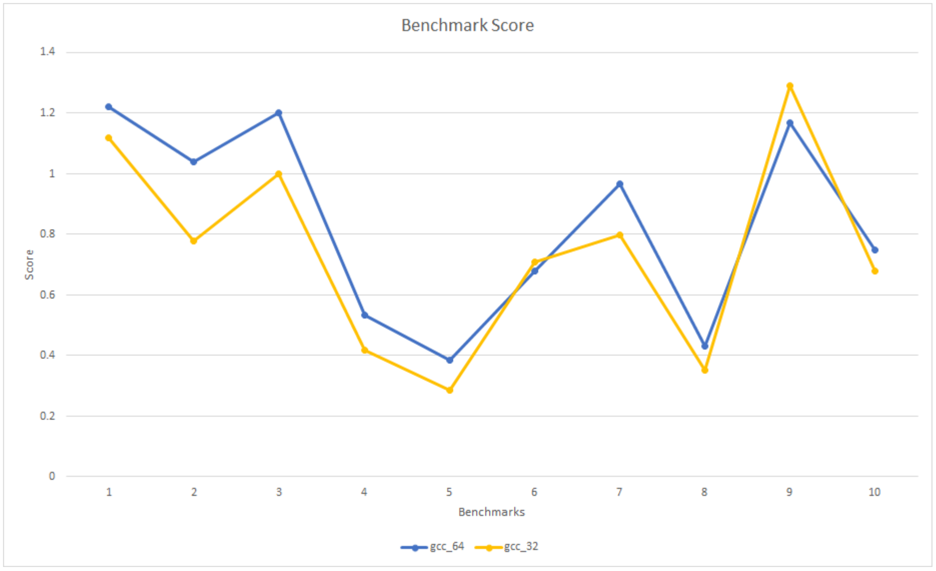

Figure 2: Scores on 10 benchmarks for gcc 32-bit and gcc 64-bit

All benchmarks have small memory demand and hence can run in 32-bit mode. 32-bit platform are almost deprecated and running a 32-bit executable on a 64-bit machine has some overheads mainly because of higher kernel-call cost because of mode switch. 64-bit platform has more registers and complex instruction set for the compiler to work with. So in general a well-written program will run faster in 64-bit mode than in 32-bit mode. This trend can be verified in Fig.2. Having said, because 32-bit mode has smaller address space, the native pointer size is smaller. So in cases where a lot of pointer indirection and pointer arithmetic has to be done, 32-bit application can have an advantage. This is exactly the case in 6th and 9th benchmark. 6th benchmark is alpha-beta tree search (alot of pointer indirection) compiled to execute in small memory. 9th benchmark tests many Fortran 95 array handling features (including some intrinsic functions) for use with integer arrays.

3

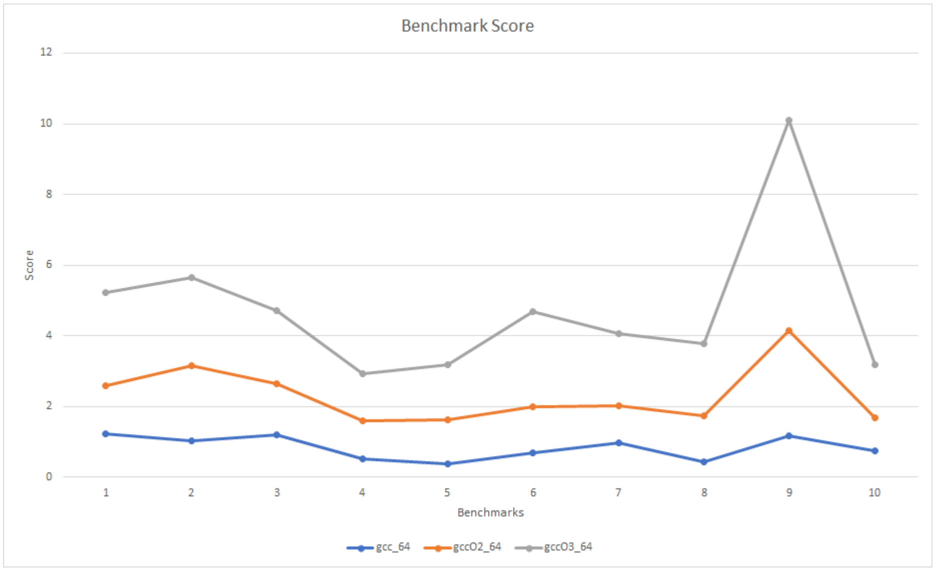

Figure 3: Scores on 10 benchmarks for gcc with O0, O2 and O3 optimization

The default unoptimized scheme takes least compilation time and mainly used for debugging purposes. O1 is the first optimization level. The compiler tries to produce faster, smaller code without taking much compilation time. O2 is level above O1 and activates a few more flags in addition to the ones activated by O1. With O2, the compiler will attempt to increase code performance without compromising on size, and without taking too much compilation time. SSE or AVX may be be utilized at this level. O3 is the highest level of optimization possible. It enables optimizations that are expensive in terms of compile time and memory usage.

O2 optimization level in gcc on average has benchmark score about 3 times that of unoptimized gcc score and O3 optimized benchmark executables are nearly twice as fast as O2 builds. The Fortran benchmark (9^{th}) really stands out as O3 flag outputs nearly 10 times faster code than the unoptimized code. It may be attributed to the fact that O3 enable ftree-vectorize *i.e* loop vectorization which could happen a lot in this benchmark as it uses alot of array operations.

4

Yes, the kind of benchmark influences results between compliers. This can easily verified by the variance in performance in different benchmarks in Fig.1. Different compilers use different optimization strategies while compiling to machine code. It may so happen that a complier does better than another compiler in one program and worse in the other program. A simple example could be that compiler A discards a piece of dead-code (code that doesn't affect the output) while complier B doesn't. Then a benchmark which has some dead-code present in it will perform better with the complier A than with compiler B.