Lab0: Benchmarking apparatus

COL729: Compiler Optimizations

Priyanka Singla (2018ANZ8387)

January 14, 2019

1 Comparison: GCC vs. ICC vs. Clang

1.1 Experimental Setup

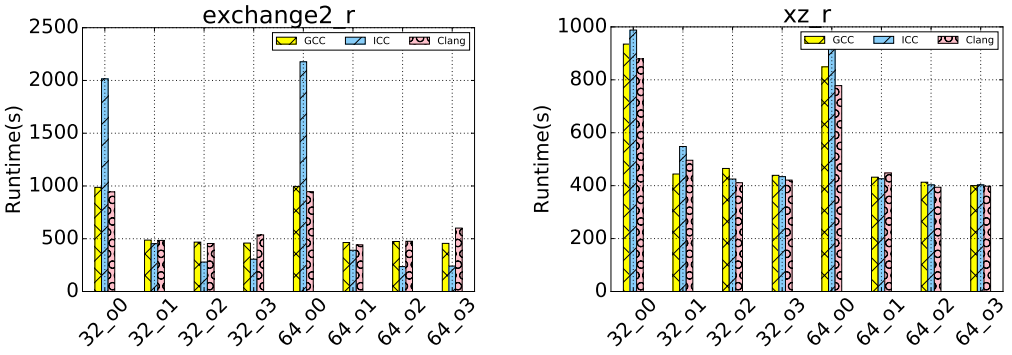
The experiments were run on a Desktop system with Intel(R), Core $i7-6700$, processor running at a frequency of $3.40GHz$. The system ran Ubuntu Linux 16.04. Following compiler versions were used:

1. GCC: version 5.5
2. ICC: Intel Parallel Studio XE 2019
3. Clang: version 3.8

For all the experiments, each benchmark was executed twice and the greater of the two values is reported. The experiments were performed in isolation (total cpu utilization for any other process in the system was less than 10%), so that the results reflect the actual performance.

Analysis The results clearly show that unoptimized applications underperform than the optimized applications. Most effective optimization (best): icc (in most cases) Least effective optimization (not good): clang

1. Which compilers are better in what aspects? I present the comparison for the compilers based on two parameters: i)Build times ii)Execution times **Build time:** The build times for the 32/64 bit executables were as follows (for O0 optimization): i)gcc 73-76 sec, ii)icc 104-111 sec iii)clang 74-80 sec Similarly for building benchmarks with o3 (64-bit)



(a) (b)

Figure 1: Benchmark Results

flag, the times taken were: i) gcc 190s, ii)icc 500s, iii) clang 164s So from compile time perspective gcc and clang are far better than icc.

Execution time: However, at O0 optimization level, the executables compiled by icc have higher runtimes in comparison to gcc and clang. Infact the unoptimized clang executable has the best running time for almost all the benchmarks (except *gcc_r*). While when optimizations are applied, clang's performance is similar to other compilers for most of the benchmarks. Infact, for certain benchmarks like leela, xalancbmk, omnetpp, the optimized clang executables perform quite poorly than icc's and gcc's cptimized executables.

In contrast, an unoptimized icc executable (O0) has the worst performance. However, the optimized icc executable performs the best for most of the benchmarks (except *gcc_r*). Another observation is that $x264$ benchmark compiled using icc at o1 does not perform very well. Thus in conclusion: if we are concerned with total (compile + execution) time, then using gcc compiler is a good option, while if only execution time is considered then using icc (with optimization level $>$ 1) is best suited.

2. Which are faster: 32-bit or 64-bit executables? Why? Running times were found to be similar for both 32-bit and 64-bit executables for most of the benchmarks, however 64-bit executables were on faster side.

The 64-bit benchmarks had large size, around $25\% - 30\%$ more, than 32-bit executables. Due to their large size, they would have required more memory and hence probably would have more time for accessing RAM (upon a cache miss). However, despite their large size, they were faster because according to the article[1], where a math-heavy code when compiled in a 64-bit environment can get the benefits of new capabilities of 64-bit processor. Our intrate benchmark suite performs compute intensive integer operations and hence performs better as a 64-bit executable. Also 64-bit executables benefits from extra general purpose registers.

3. How are the various optimisation levels different? How do these differ across compilers? As the optimization level is increased from 1 to 3, the size of the code generated is also increased and the program execution becomes faster. Also the compilation time increases with increase in optimization level. At the first level (-O1) the focus is on reducing code size and execution time, and very basic optimizations are performed (the optimizations which take more compilation time are not done). The next level (-O2) further perform optimizations which do not involve space-speed tradeoffs and increases the compilation time and performance. The next level (-O3), in addition to optimizations by -O2, includes features like loop unrolling, function inlining which involve space-time trade offs, thus resulting in a larger executable which will probably be faster. There are two more optimization levels, namely -Os and -Ofast. The former optimizes size, and includes optimizations similar to -O2 except those which increase the size. Infact it performs optimizations to reduce size. The later optimization, -Ofast, is even more agressive than -O3 and can result in incorrect code, and hence is not recomended.

   The results show that the performance increases dramatically when optimization is applied to an unoptimized code. The behaviour across compilers is as follows:

   - For gcc, the runtime decreases by $6\% - 10\%$ with increase in optimization level (from 1 to 2), while from -O2 to -O3 the performance does not increase much ($around3\%$).
   - Similarly, for icc the runtimes follow a decreasing trend, i.e. from -O1 to -O2 the run time decreases by $10\% - 45\%$. However for cer-

tain benchmarks like X264 the runtime from -O1 to -O2 decreases considerably. The runtimes from -O2 to -O3 are not affected much.

- For clang, similar decrements $8\% - 35\%$ were observed for most of the benchmarks. However, for Xalan and Leela benchmarks, the runtimes from -O1 to -O2 decreased considerably. In contrast, for exchange(-O2 to -O3) and mcf(-O1 to -O2) benchmarks the runtimes increased with optimization level, thus indicating that higher optimization level does not always lead to higher performance.

4. Does the kind of benchmark influence the results between compilers? Why? Yes, as can be seen in the results, omnetpp's and xalan's runtime decrease with more rate with increase in optimization level for clang compiler (from -O1 to -O2), while for other benchmarks similar rates were followed.

2 References

1. Moving from 32-bit applications to 64-bit applications.

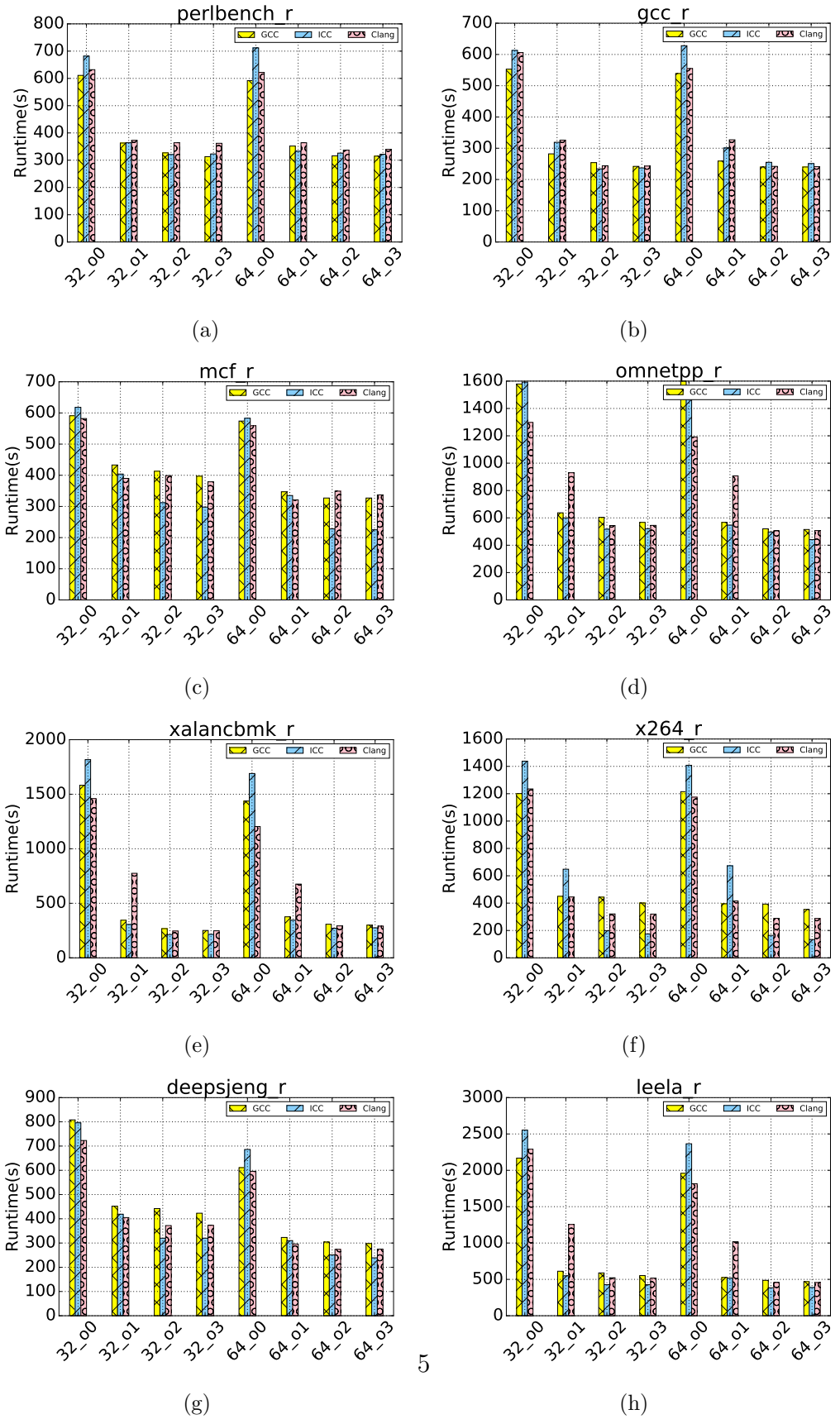


Figure 2: Benchmark Results contd.