# Performance Analysis of GCC Compiler Optimizations

CSC 5593 - Advanced Computer Architecture

Team 2: Stefani Moore, Aishwarya Mulkalwar, Courtney Peverley

## Table of Contents

## I. Introduction

This project uses the SPEC CPU2017 benchmarking suites to assess the performance of the various GCC optimizations levels (unoptimized, O1, O2, O3, Os, Ofast, and processor-specific optimizations) in the hopes of discerning when and where a programmer should use each optimization level. By attempting to determine the specific optimizations that result in a runtime slowdown, we try to narrow down the effects that compiler optimizations have on the underlying architecture and therefore program execution time.

## II. Motivation

The motivation for this project was to use what we have learned over the course of the semester and apply that knowledge to something programmers use every day: compilers. Not only is compiler optimization program-specific, but it is also architecture-specific. Depending on the underlying architecture (cache size, branch prediction strategy, etc), certain optimizations will result in an overall slowdown. We wanted to see if we could use what we know about computer architecture to analyze trends in optimized/unoptimized code performance.

## III. Background

A compiler is the middle man between a programmer and a computer. It takes the high-level language written by the programmer and translates it into a language the computer can understand. During this process, a compiler may also perform optimizations on the code with the following goals in mind:

1. Retain program meaning
2. Improve the source program somehow

When a program is sent to the compiler, it goes through (1) the front end, (2) the optimizer, and (3) the back end. In (1), the program is checked for syntax errors and its lexical components are categorized. In (3), registers are allocated and the machine code is generated. For this project, we are focusing on (2), the optimizer. When the program emerges from the front end, it is unoptimized. The code is sent through the optimizer repeatedly until all optimizations have been performed. Because of this, the more optimizations that are performed, the longer compilation will take. The idea is to incur this one-time slowness so as to allow for speedups for all future executions [1].

GCC, the GNU Compiler Collection, is an open-source compiler that is capable of performing optimizations on code, which can be turned on/off with a series of flags at compile time. It can compile programs in the high-level languages C, C++, Fortran, Ada, and Go. In addition to individual optimizations that a programmer can toggle on and off, GCC also offers six levels of optimization: O0, O1, O2, O3, Os, and Ofast. O0 is the unoptimized option (default) and is mostly used for debugging because it does compiles the code as written and therefore errors become easier to find. O1 is the lowest level of optimization, and O2 and O3 build upon that by adding optimizations. As mentioned earlier, each added optimization will increase compile time, so all of this is a balancing act. Os is optimizing for size, which is mostly useful when size is important (like on embedded systems). Finally, Ofast adds several non-standards-compliant optimizations to the O3 level in the hopes of speeding up execution even more. By using Ofast, we risk incorrect output for the sake of a faster runtime (ideally). GCC also includes multiple methods for optimizing for a specific architecture. In this project, we used the march=native flag for this purpose [2].

The SPEC CPU2017 benchmarking package contains four benchmarking suites used for assessing performance (speed, accuracy, etc). The four suites are: intrate, intspeed, fprate, and fpspeed. For this project, we mostly use intrate (with one exception we will get into later) to analyze the performance of GCC. Intrate consists of 10 benchmarks written in C, C++, and Fortran, and using common workloads like ray tracing, data compression, and language interpretation [3].

Over the course of this report, we will discuss the effects that compiler optimizations have on compilation (build) time and execution (run) time of various benchmarks in intrate. We also use the perf tool to look at total instructions executed at runtime, mispredicted branches, cache misses, and page faults. Those metrics will hopefully reveal the reasons for any runtime results that do not reflect our expectations (when adding optimizations causes an overall slowdown).

## IV. Project Description

### A. Overview

We began by analyzing the intrate suite as a whole for each GCC optimization level (O0, O1, O2, O3, Os, Ofast, and O3 with processor-specific optimizations), as well as for the Intel compiler. Once we got a feel for how these optimizations affect performance on average, we started to dwindle down our data set. The project will be outlined in the following 3 phases: (1) assessing performance at the intrate benchmark suite level, (2) assessing performance of the individual benchmarks in the intrate suite, and (3) categorizing applications and determining which optimizations work best for which types. The overview of our project methodology can be seen in Figure 1.

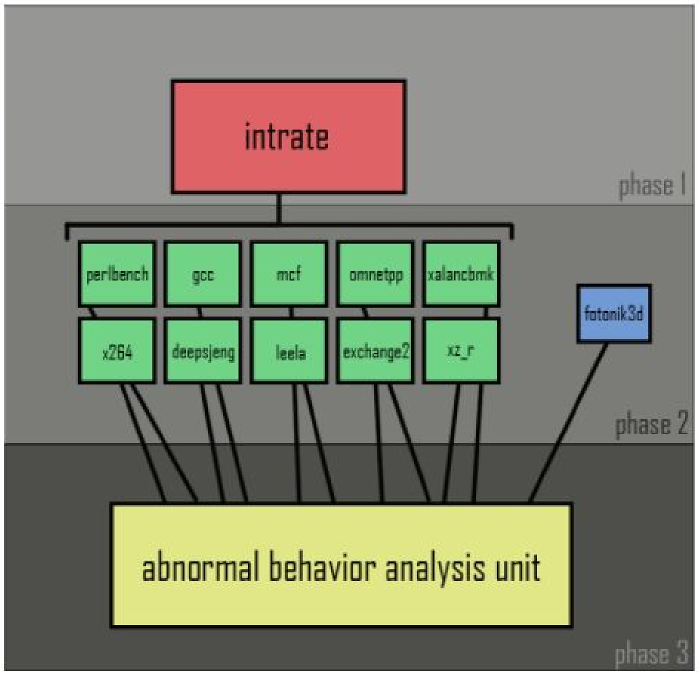


***Figure 1:*** Project outline

### B. Hypothesis/Expectations

We had a number of expectations going into this project:

1. As the number of optimizations increases, the build times will also increase.
2. Adding more optimizations will decrease execution time.
3. The processor-specific optimizations will give us a small speedup.
4. Os will take less time to compile than O2, but will execute more slowly.

*C. Phase 1*
By looking at the performance (build times, runtimes, cache misses, etc) of the entire intrate benchmarks, we establish a baseline for average performance for next phases. This phase gives us a clearer picture of what we were looking for during Phases 2 and 3. Specifically, when a benchmark is demonstrating “odd” behavior and what might be the root cause of that behavior.

*D. Phase 2*
Phase 2 involves performing the same tests as in Phase 1, but on each of the ten benchmarks in the intrate suite with each of eight optimization levels: O0, O1, O2, O3, Os, Ofast, O3 + march (processor specific), and Intel. We create further graphs comparing performance metrics across all tested optimizations.

*E. Phase 3*
Phase 3 takes the results from the previous two phases and combines them to identify benchmarks that exhibit abnormal behavior (when compared to the expected results) or exhibit behavior that explains abnormalities from Phase 1. We then take all of those “abnormal” benchmarks and systematically remove specific optimizations from the underperforming optimization level(s) until the performance improves. This will give us an idea of what types of optimizations negatively impact the performance of what types of applications.

## V. Experimental Results & Analysis

*A. intrate*
We ran the entire benchmarking suite “intrate” with the perf stat -d command to get a detailed look at cache misses, page faults, branch mispredictions, etc. Though the results are difficult to interpret effectively since they represent the results from ten different benchmarks, the relative differences between the performance for the optimization levels help define what is to be expected and what to look for during individual benchmark testing. Results can be seen in Appendix A. As we predicted, adding more optimizations results in longer build times. Also, for the most part (with one exception) more optimizations gave us an overall speedup during runtime. The exception is O3, which took longer to run than O2 despite having more optimizations enabled. We will discuss the “problem” programs that led to this discrepancy in the following sections. In addition to runtimes, we also, using the perf stat -d command, analyzed branch misprediction, cache miss rates, and page faults. A more in-depth discussion of these metrics in regards to compiler optimization can be found in the “Architectural Effects of Compiler Optimization” section below.

*B. 500.perlbench_r*
The first benchmark in the intrate suite is perlbench. Perlbench builds a variation of minimal version of Perl and runs three scripts through it. These scripts deal with spam email detection

and email-to-HTML conversion [4]. When performing our abnormal behavior analysis, we found that O3 ran more slowly than O2 for this benchmark, which was unexpected. It wasn’t excessively slow, but, for optimizations that are supposed to cause a speedup, the performance was disappointing. Some key charts can be seen in Figure 2, and further can be found in Appendix B.

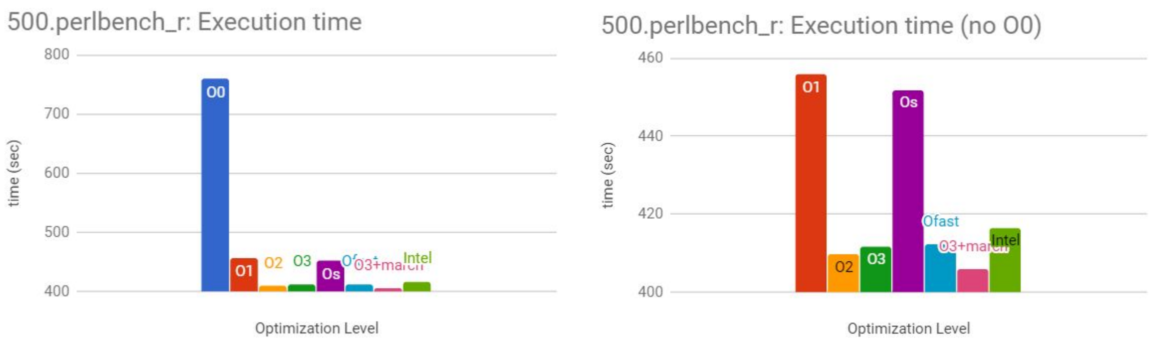


***Figure 2:*** Execution times for the perlbench benchmark.

It is difficult to discern what the cause of the slowdown is. O3 executed fewer instructions and had a lower average IPC. This IPC decrease may be attributed to the higher branch misprediction rate at the O3 level. In an effort to nail down the exact optimization(s) that are causing this slowdown at O3, we removed optimizations one by one. The results of this process can be seen in Figure 3. The figure is a bit confusing, so for clarity: the chart maps removed optimizations to associated speedup or slowdown. This means that a bar in the positive region corresponds to a specific optimization that, when removed, allowed for execution time speedup, which means that this optimization had a negative impact on performance in our original test. On the other hand, the one bar in the negative region represents the optimization that did help speed up execution time. It is worth noting here that four optimization enabled at the O3 could not be turned off without compilation-time errors. Those optimizations are tree-loop-distribution-patterns, tree-loop-vectorize, split-paths, and loop-unroll-and-jam.

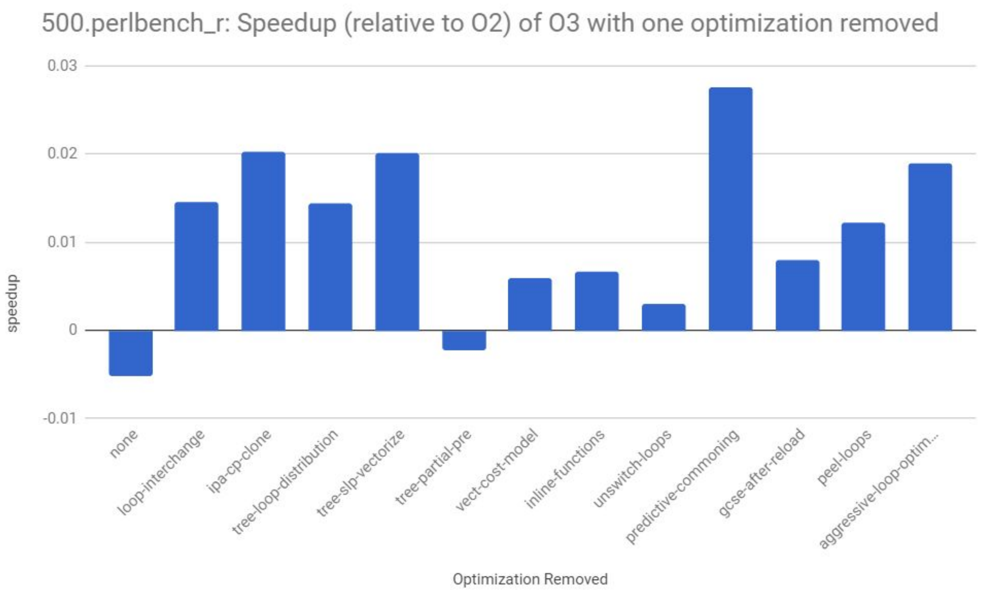


***Figure 3:*** speedup/slowdown of O3 with optimizations removed

The optimizations causing the most trouble for this program are predictive-commoning, ipa-cp-clone, and tree-slp-vectorize. Predictive commoning is a loop optimization similar to loop peeling. It is a means to decrease the number of loads and stores within a loop. IPA refers to interprocedural analysis, and CP to constant propagation. Ipa-cp-clone takes functions with constant arguments and clones them with various constants. Tree-slp-vectorize uses SLP (superword level parallelism) to vectorize and unroll loops. All of these will increase code size to various degrees. It is currently unclear what exactly is causing the performance loss when these optimizations are applied, but the main difference between O3 and, for example, O3 without predictive commoning, is the number of instructions executed. By removing predictive commoning, we decreased the number of instructions by 3%, which appears to be the main reason for the speedup. In theory, loop optimizations should decrease the overall instruction count despite increasing code size. Loop optimizations ideally decrease the number of instructions executed *inside* of the loop, which will decrease instructions overall even if the loop is unrolled. Unfortunately, for some reason, loop optimizations are increasing instructions for this benchmark. To see all other comparison metrics, see Appendix M.

Perlbench is difficult to classify in terms of what attributes of the code may impact the performance of optimizations. This is because the Perl interpreter has a large and varied code base. That said, the conclusion we can draw here is that aggressive loop optimizations applied by the O3 level are not always going to give us a performance boost. In this case, the slowdown when using O3 is minimal, so it would not be a drastic mistake to use O3.

## *C. 502.gcc_r*

502.gcc_r is a benchmark that compiles the GCC compiler and runs several programs through it for testing [5]. We like to refer to this benchmark as “GCC Inception” because we are effectively optimizing the GCC compiler via GCC. The charted results can be found in Appendix C, but the graph of execution times can be found in Figure 4.

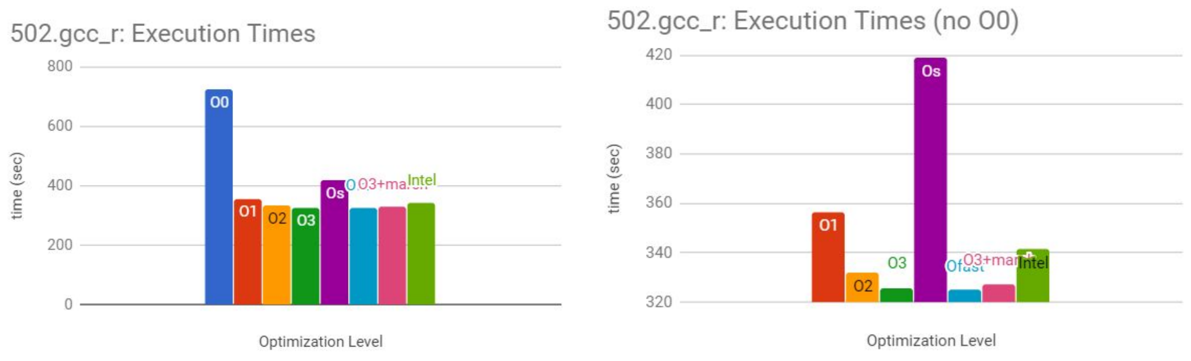


***Figure 4:*** Execution times for the GCC benchmark.

The execution times follow the pattern we expect: O0 is the slowest and increasing the optimizations makes execution faster. The processor-specific flag (-march=native) actually slowed us down here, which is something we will see again and again. Os is slower than O1 in this case. This is a slight deviation on what we expected. Os has all of the O2 optimizations, but removes 8 that are known to increase code size. Six of these removed optimizations are first introduced at the O2 level (reorder-blocks-and-partition and align-functions to name two), one is introduced at the O1 level (reorder-blocks), and one is a default optimization (prefetch-loop-arrays). Since Os has overall more optimizations than O1, we expected to see faster execution times from Os. We will see repeatedly in this experiment that Os typically performs worse than O1.

In an effort to nail down why this disparity exists, we looked at the individual optimizations removed. Because we found that removing these optimizations caused a slowdown even relative to O1, it seems like the reorder-blocks and prefetch-loop-arrays optimizations had been very helpful in the previous tests. Reorder-blocks reorders instructions to make them more cache-efficient and to decrease the number of taken branches. Prefetch-loop-arrays does just as one would guess: prefetch arrays used by loops. The main performance drain on Os is the increase in instructions (IPC is higher for Os than its optimized counterparts). Since, if anything, prefetching loop arrays would actually increase the instruction count (you'd need extra instructions for those prefetchings), we think the reason for the slower Os performance is specifically the block reordering.

With all of that in mind, we attempted to prove our theory by taking out the reorder-blocks optimization from O1 to see if this had a large performance impact. We had previously tried adding reorder-blocks to the Os level, but this resulted in a huge increase in execution time, presumably because adding the flags manually means that it is not done in the most effective order. Optimizations are performed one at a time as we repeatedly send the code through the optimizer, so order plays a large role in how well these optimizations work individually and as a whole. This problem also arises when removing individual optimizations from the level. It is hard to tell what the impacts of only this optimization are since it is so tied to how it interacts with the other optimizations. The moral of the story is that we were not able to verify our theory. Taking prefetching and reorder blocks out of O1 did give us a slowdown, but not it was not as dramatic as anticipated and was not enough to conclude that it is one of these two optimizations that is causing our Os slowdown for this benchmark.

Depending on the program, Os will likely perform slower than O1 or slower than O2 and faster than O1. If, like with the GCC benchmark, the reorder blocks optimization has a large positive influence on the performance, Os may perform worse than O1. Otherwise, the extra optimizations it has over O1 should give it a performance more-or-less in between O1 and O2. Why this benchmark specifically has such a performance drop without reorder-blocks is a mystery. Like Perlbench, GCC is a large code base so it’s challenging to nail down the specifics that would lead to this behavior.

*D. 505.mcf_r*

MCF is a combinatorial vehicle-scheduling benchmark [6]. When testing this benchmark, we got very similar results as we had with the GCC benchmark, in terms of execution times and instructions executed (see Figure 5 for execution times). You may find additional charts and metrics in Appendix D. Again, we see that Os executed more slowly than O1 (this time not as dramatically). We also see that O3 with the processor-specific flag enabled again gave us a slowdown. It is hard to know exactly why, since we don’t know what specifically is happening with the processor-specific optimizations.

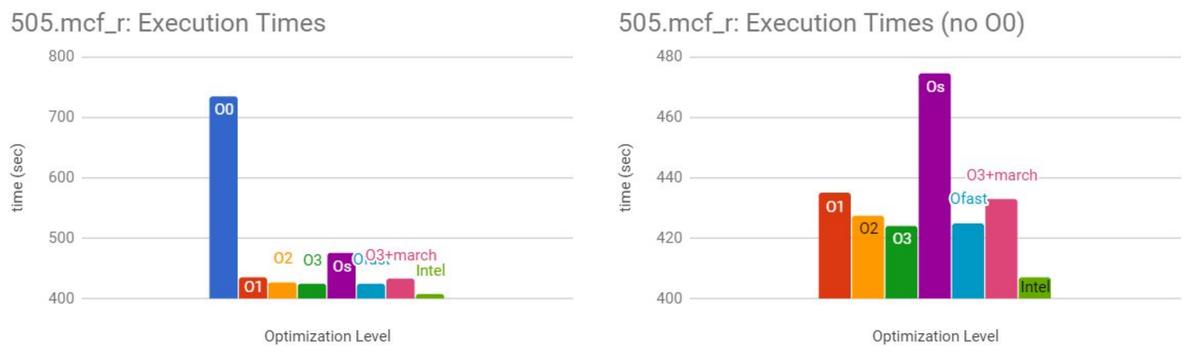


***Figure 5:*** execution times for the MCF benchmark

We won’t go into too much detail here since our analysis was similar to that for the GCC benchmark. What we will note is that for both this benchmark and the previous (GCC), our findings are still consistent with what we expected: Os is slower than O2, and also sometimes slower than O1. In terms of choosing optimization levels, though, if you, as a programmer, need to optimize for size, you are unlikely to worry about whether it is worse than O1.

*E. 520.omnetpp_r*

520.omnetpp performs a discrete event simulation on a large Ethernet network [7]. Our findings were that the performance of the various optimization levels basically followed our expectations. With more optimizations, the runtimes were faster. The processor-specific optimization did again give us a slower result, providing further evidence that the GCC processor-specific options are very limited and often detrimental overall. Execution time charts can be seen in Figure 6. As you can see, we again see that Os is slower than O1. As it turns out, it appears Os is frequently slower than O1 despite having more optimizations overall. Additional charts can be found in Appendix E.

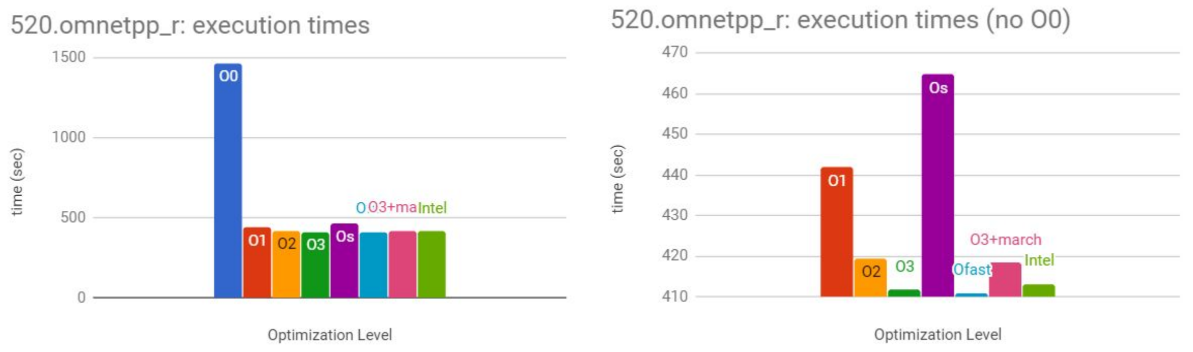


***Figure 6:*** execution times for the omnetpp benchmark

*F. 523.xalancbmk_r*

523.xalancbmk_r is an XSLT processor for turning XML documents into HTML, text, or other XML document types. For testing, it takes, as input, an XML document and an XSL stylesheet, and outputs an HTML document [8]. The results are very similar (in terms of runtime comparison between optimizations) to the omnetpp benchmark. Further charts can be found in Appendix F.

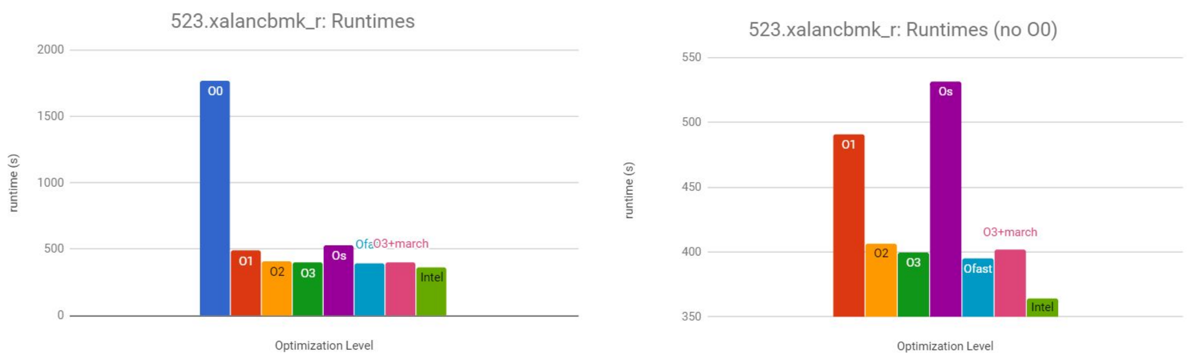
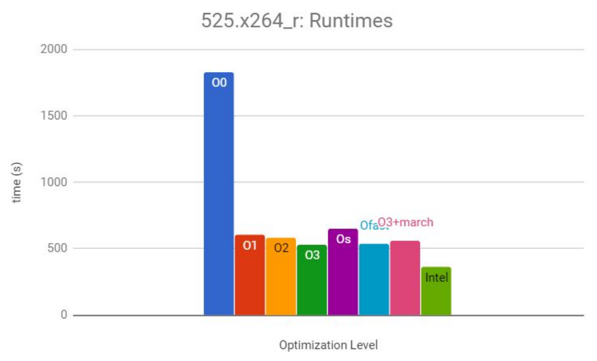


***Figure 7:*** execution times for xalancbmk

*G. 525.x264_r*

525.x264_r is a library and application for encoding video streams. It is used for video compression [9]. Our test results showed nothing terribly abnormal, except for the two things we've been seeing repeatedly: Os is slower than O1, and O3 with processor-specific optimizations is slower than O3.



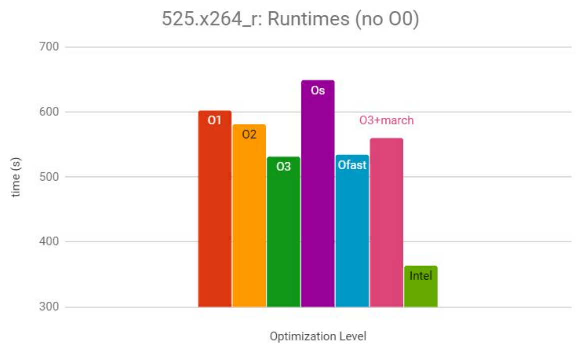
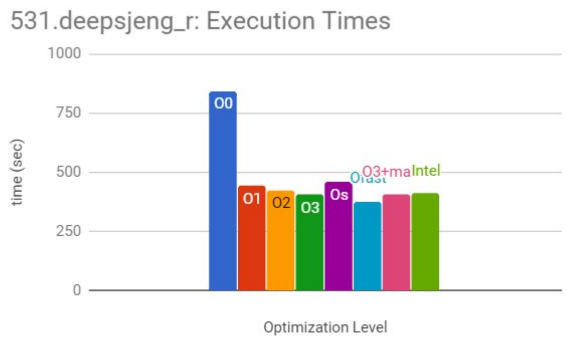


***Figure 8:*** execution times for x264

## *H. 531.deepsjeng_r*

531.deepsjeng_r is a C++ artificial intelligence benchmark. It attempts to find the best move in a chess game [10]. The results for deepsjeng are also fairly "normal" (see Figure 9). More graphs can be seen in Appendix H. Os is still worse than O1, and execution time decreases when we add more optimizations. Ofast performs particularly well here. Ofast turns on optimizations that reorder operations to produce faster code. The risk is, as referenced earlier, this type of optimization could lead to incorrect results. Since the calculations are all integer ones, the result is less likely to be incorrect. And since there are many integer calculations, the extra Ofast optimizations are very helpful. Like before, we also see that the processor specific optimizations are not helping.



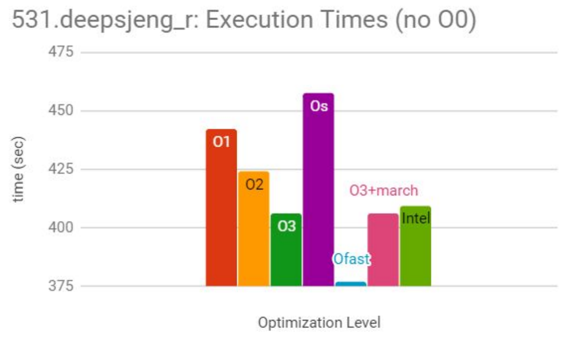


***Figure 9:*** execution times for deepsjeng

## *I. 541.leela_r*

Like deepsjeng, 541.leela_r is an artificial intelligence benchmark. It runs a Monte Carlo simulation that plays an input Go game [11]. The execution time results (seen in Figure 10) are again fairly "normal." More charts can be seen in Appendix I. Increasing optimizations decreases execution time and O3 plus processor-specific optimizations is slower than O3. The main difference here is that Os is actually faster than O1. This happened in perlbench, but we were mainly discussing the discrepancy between O2 and O3 then. Here we have a case where Os is faster than O1, meaning those extra optimizations we have in Os outweigh the negatives of taking out the ones that cause a code size increase. The implication is that loop prefetching and block reordering do not have as much of an effect on runtime as with the previous tests. As previously discussed, these facts will likely not impact a developer's decision to compile using Os. If you're on an embedded system and need to minimize space used, you'll probably use Os no matter what. It's still better (in execution time) than O0, and takes up less space (on average) than all of the other optimization levels.

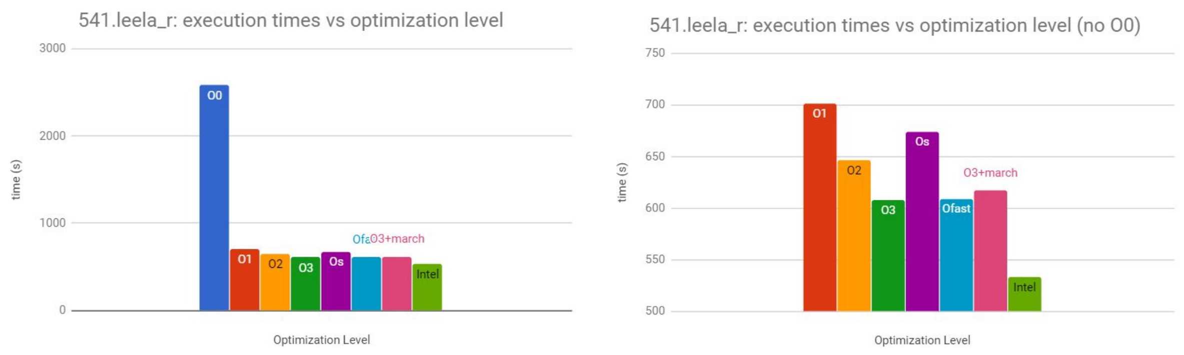


***Figure 10:*** execution times for leela

*J. 548.exchange2_r*

The results for exchange2 are the most unexpected. O3 was the second-slowest optimization level. In fact, of all of the GCC optimization levels, O1 was the fastest, but even O1 was beat handedly by Intel. Os had fewer instructions than O2, but took longer to execute, which indicates that there was some reason Os had a relatively low IPC. The exchange2 program is used, outside of the benchmarking world, to create Sudoku puzzles for competitions. It is recursive, which is likely the reason for the strange behavior we are seeing [12]. We will discuss this further in the Analysis section below. All graphs can be seen in Appendix J and key graphs are shown in Figure 11.

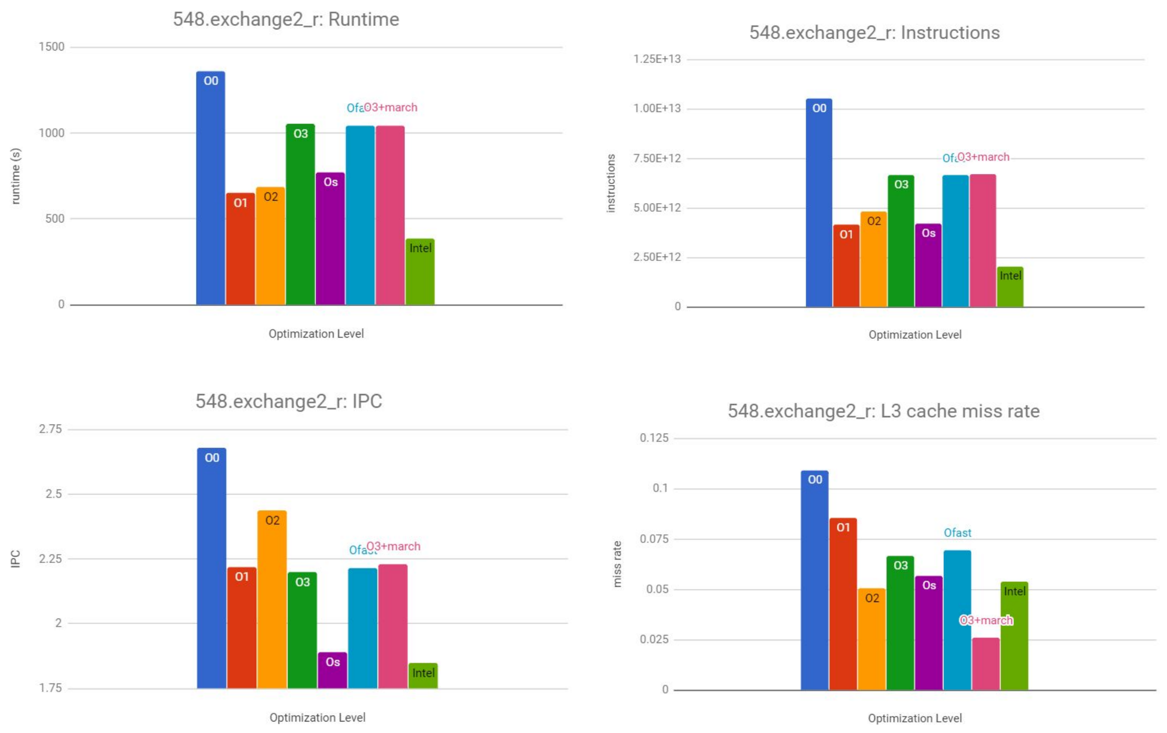


**Figure 11:** Runtime, instruction count (during program execution), IPC, and L3 cache miss rate for the exchange2_r benchmark.

We identified the L3 cache misses as the main reason for the low IPC for Os. The branch misprediction rate, page fault rate, and L1 cache miss rate for Os are lower than O2, so it seems like the L3 cache miss rate is the culprit. The L1 cache miss rate for all of the optimization levels is very low (mostly under .04%). Specifically, for Os, the L1 cache miss rate is 0.03%, but the L3 miss rate is 5.7%. The low L1 miss rate means that we don't have too many L3 accesses, but the L3 miss rate means that, of those accesses, a (relatively) high percentage are misses.

In an attempt to determine the precise optimizations that led to the poor performance by GCC optimization levels for this benchmark, we first re-built the benchmark using the O0, O2, and O3 GCC optimization levels (as well as Intel), but this time with the -S flag enabled in order to see the pre-assembled code. What we found is that Intel performs so well because it seems like it may be performing some sort of tail call recursion optimization where the other are not. Tail call recursion can be optimized into a loop, which can then be further optimized. Theoretically, this could be what is giving Intel such an edge here. In the pre-assembled code, the Intel one had much fewer call instructions and more jump instructions since these recursive calls had been optimized into loops instead. As for the GCC optimization levels, the biggest performance disappointment is O3. So, in order to try and discern why O3 performs so poorly, we eliminated several individual optimizations in the hopes of figuring out which ones are part of the problem.

This testing proved unhelpful. Eliminating the additional optimizations from O3 (when compared to O2) one at a time, in groups, and also all at once, led to only a minimal speedup.

As previously discussed, the process of adding/removing optimizations is not an effective testing method because the order the operations are performed is so important. Our conclusion from all of this is: be wary if you have a recursive algorithm. GCC is not very good at optimizing for recursion, so it might be best to just use O1. If you have the Intel compiler, for sure use that, though.

*K. 557.xz_r*

556.xz_r is a data compression benchmark that does compression entirely in memory (minimal I/O) [13]. In this benchmark, as seen in Figure 12, the results are fairly standard for these tests. Further graphs can be seen in Appendix K.

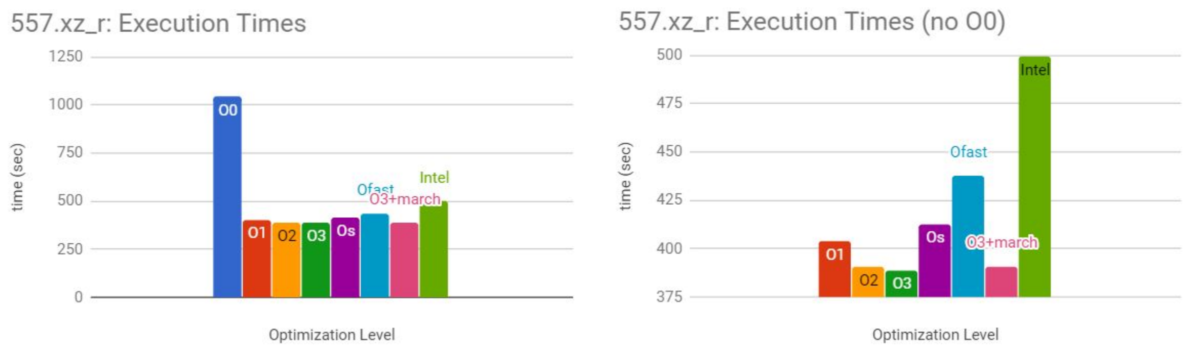
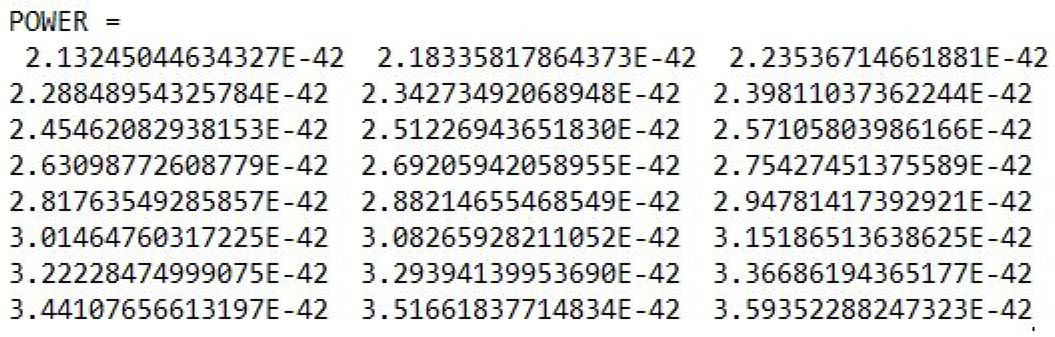


***Figure 12:*** execution times for xz

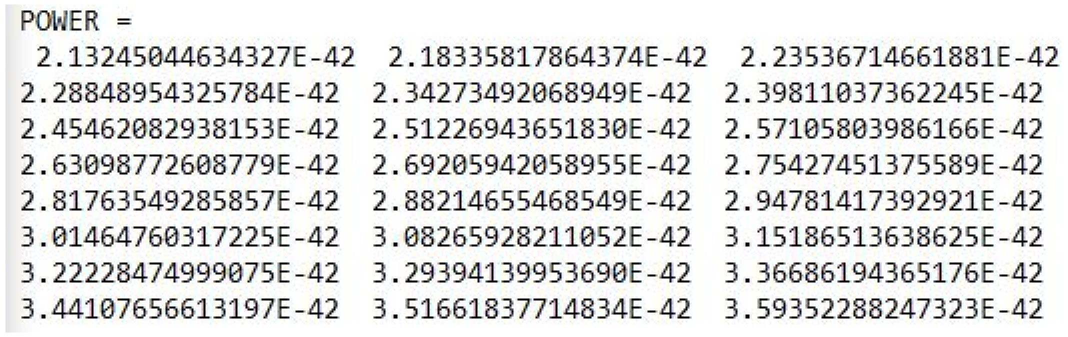
*L. 549.fotonik3d_r*

We ran our optimization tests with 549.fotonik3d_r because it was a floating point test (it is within the fprate suite) that could be used to gauge the accuracy of Ofast. 549.fotonik3d_r is a computational electromagnetics program that computes electric/magnetic fields of power planes [14]. Charts depicting relative performance can be found in Appendix L. We will focus on output in this section. Figure 13 contains the output for O0-O3 and Os, and for Ofast.

In the figure, there are numerous differences in the two outputs. For example, the second line, middle number is different at the last digit. We also tested several other benchmarks from the fprate suite to see if Ofast consistently gave incorrect results, but that wasn't the case. The takeaway is: if speed is more important than anything else, use Ofast. The floating point issues will only affect the results minimally, if at all. That said, if accuracy is the end goal, do not use Ofast.



```
POWER =
 2.13245044634327E-42  2.18335817864373E-42  2.23536714661881E-42
2.28848954325784E-42  2.34273492068948E-42  2.39811037362244E-42
2.45462082938153E-42  2.51226943651830E-42  2.57105803986166E-42
2.63098772608779E-42  2.69205942058955E-42  2.75427451375589E-42
2.81763549285857E-42  2.88214655468549E-42  2.94781417392921E-42
3.01464760317225E-42  3.08265928211052E-42  3.15186513638625E-42
3.22228474999075E-42  3.29394139953690E-42  3.36686194365177E-42
3.44107656613197E-42  3.51661837714834E-42  3.59352288247323E-42
```



```
POWER =
 2.13245044634327E-42  2.18335817864374E-42  2.23536714661881E-42
2.28848954325784E-42  2.34273492068949E-42  2.39811037362245E-42
2.45462082938153E-42  2.51226943651830E-42  2.57105803986166E-42
2.63098772608779E-42  2.69205942058955E-42  2.75427451375589E-42
2.81763549285857E-42  2.88214655468549E-42  2.94781417392921E-42
3.01464760317225E-42  3.08265928211052E-42  3.15186513638625E-42
3.22228474999075E-42  3.29394139953690E-42  3.36686194365176E-42
3.44107656613197E-42  3.51661837714834E-42  3.59352288247323E-42
```

***Figure 13:*** Program output for O0, O1, O2, O3, and Os (top), compared to program output for Ofast (bottom).

## VI. Architectural Effects of Compiler Optimization

On average, the branch misprediction rate increases as we add optimizations. This can likely be attributed to the loop optimizations that kick in as we go through the optimization levels. For example, loop unrolling will likely have a negative effect on branch prediction because you lose the history associated with n executions of a branch instruction when you unroll the loop and turn it into n separate instructions. Similarly, loop unswitching will decrease the number of branches overall by taking branches with loop-invariant conditions out of the loop. This means that, although we decrease the total instructions to execute (good), we only take the branch once and thus cannot predict its outcome as easily.

As for the cache miss rates and page faults, there are several optimizations in O2 and O3 that improve locality and therefore decrease page faults and cache misses. One of these, at the O2 level, is reorder-blocks-and-partition, which categorizes blocks by the likelihood of referencing them (if they're referenced a lot, then they are deemed "hot" and placed with other hot blocks; the same goes for "cold" blocks). This improves locality because it increases the likelihood that a memory access will be found in the cache or page table. Another optimization is loop-interchange, which is enabled at the O3 level. Loop interchange involves switching the order of nested loops to improve locality. This is particularly useful for iterating through arrays such that you don't get a cache hit for every single entry. Also enabled at the O3 level is tree-loop-distribution, which takes a large loop and breaks it into smaller loops so that we have everything we need for the loop in the cache or page table until the loop is complete, and then we move to the next loop. The trend for both page faults and cache misses is that they increase steadily with O1, then O2, but drop off a bit at O3. In general, as code size increases, the likelihood for page faults and cache misses increases since there's a larger base of data we're working with. The aforementioned optimizations help to abate this, which is why O2 and O3 do not have exorbitantly large cache miss rates and page fault numbers.

The overall IPC for optimized code is typically smaller than for unoptimized code. This is because of what we've discussed: branch misprediction rates will go up, as well as cache misses and page faults. Despite this, the net effect of compiler optimizations is positive because the optimizations will result in (for the most part) fewer instructions to run, even if they increase the code size. With fewer instructions, the end result can be a speedup even if the IPC is lower thanks to all of those mispredictions and misses of various types.

## VII. Conclusion

After running many tests using the GCC optimizations and analyzing the data we found results that agreed with our expectations as well as some results that were unexpected. Overall, tests that were unoptimized (O0) took the longest to run. So much so that we had to take them out of our graphs in order to discern other run time results for all of our tests. This was as expected because the general goal of most optimizations is to decrease overall run time. There is a tradeoff however, as the number of optimizations is increased, so is the compile time. This was a consistent result among almost all of the tests and in general it can be said that as we increase the number of optimizations, we lower execution time with the side effect of increasing build time. Another result that agreed with our expectations was that tests compiled with the Os optimization resulted in a smaller executables than O2, but had longer execution times. This is another prime example of tradeoffs that must be made in order to optimize for a specific system. For the most part, Os was slower than both O2 and O1, which indicates that the space-for-speed tradeoffs made have a big payoff. When we remove them (for Os), the result is much slower. Our recommendation in regards to Os is to only use it when there are legitimate space concerns, like on embedded systems. Other than that, the runtime costs are too high to feasibly use Os on normal systems.

We ran into a few things in our results that were definitely unexpected.The first being that we twice saw an increase in execution time when using the O3 optimization over O2, as seen in the tests results of 500.perlbench_r and 548.exchange2_r. For exchange2, we believe that this is a result of the test's recursive nature. For perlbench, we are less sure about the reasons for this discrepancy. Despite extensive testing, we were unable to pinpoint the exact reason for the performance loss with O3; however, for perlbench, the performance drop was not very large and thus supports our general conclusion that choosing O3 when compiling is best in non-recursive situations. If you do have a recursive program, consider lessening the number of optimizations used (compile with O1 or O2), especially if time is a factor.

We also saw that the addition of the processor specific optimization (march=native) with O3 did not yield a speed up in program execution for most of the tests (compared to just using O3). Additionally, It appears that cache miss rates and branch mispredictions don't have much of an overall effect on program execution time. Unoptimized tests, for example, had the lowest cache miss rate and often the highest overall IPC, but took the longest to execute of all of the other optimizations by far. The main driving force behind compiler optimization performance is the total number of instructions executed during runtime. Optimizations increase code size, but decrease the number of instructions by shortening loops, eliminating dead code, etc. Fewer instructions means faster execution, even if the IPC takes a hit from an increase in branch mispredictions and cache misses.

Everything with optimizations is a constant trade-off with space versus speed versus accuracy. It all depends on the application and machine you wish to execute the program on. For space constrained systems, such as embedded systems, optimizing for space is more important than optimizing for speed. If you are only concerned about program speed and not so concerned about program accuracy, Ofast might be the way to go. Most programmers just use O3 by default every time, and our data supports this decision for the most part. The only exception we found was that a recursive program may not benefit from optimizations. GCC does not do a great job of optimizing for recursion, while Intel is able to do so.

## VIII. Proof of Correctness

To run a test and compare with our results, follow the instructions in "Instructions for Running a Test Benchmark." The full results for the 548.exchange2_r benchmark are listed in Figure 13 below for comparison. The perf results give you instructions and page faults (from which you can calculate page faults per 10,000,000 instructions), cycles (IPC), branches and missed branches (branch misprediction rate), L1 loads and misses (L1 miss rate), and LLC loads and misses (L3 cache miss rate).

Should you choose to run a different benchmark (GCC on average runs the fastest at only 300-400 seconds), you can compare your results with ours visually using the graphs in Appendices A-L. Keep in mind that if you choose to run the entire intrate suite, it will take between 2 and 6 hours, depending on the optimization level.

| Optimization Level | Build Time (s) | Execution Time (s) | IPC | branch mispredicti on rate | L1 cache miss rate | L3 cache miss rate | page faults per 10,000,000 instructions |
|---|---|---|---|---|---|---|---|
| O0 | 4.05640 | 1361.1309 | 2.67877 | 0.0104317 | 0.0003209 | 0.1092518 | 2.5141027 |
| O1 | 5.7246 | 650.0618 | 2.219865 | 0.012531 | 0.0003742 | 0.0854699 | 4.6837288 |
| O2 | 6.6448 | 686.5831 | 2.438949 | 0.012116 | 0.0003168 | 0.0505441 | 4.4223589 |
| O3 | 9.1196 | 1052.8154 | 2.200417 | 0.011836 | 0.0002230 | 0.0667550 | 4.3080674 |
| Os | 6.0423 | 771.62591 | 1.889963 | 0.0107979 | 0.0002967 | 0.0567413 | 3.6688829 |
| Ofast | 9.1983 | 1042.446 | 2.213634 | 0.011986 | 0.0002004 | 0.0693567 | 4.3373890 |
| O3+march | 10.2566 | 1044.4879 | 2.229739 | 0.0117766 | 0.0002548 | 0.0260019 | 4.4410979 |
| Intel | 12.2901 | 387.42824 | 1.850243 | 0.0187595 | 0.0005012 | 0.0538056 | 6.4547749 |

***Figure 13:*** exchange2 results for proof of correctness comparison

# IX. Instructions for Running a Test Benchmark

*A. Overview*

All instructions found in this section are for the Heracles system available through University of Colorado Denver. The running of the SPEC CPU2017 benchmarks for the purpose of this project requires previous knowledge of PuTTY (SSH client) for accessing the Heracles cluster and File Transfer Protocol (FTP) either through the command line or using a client such as FileZilla or WinSCP for transferring necessary configuration files from your local machine to Heracles.

The steps provided in the following sections are important to ensuring reproducible and consistent results when running the benchmark suites. In particular, the configuration files used to define the system under test and how it should build, run, and report the benchmarks in a given environment put all of the options in one easy to read environment. These configuration files are key to ensuring correct and reproducible results in conjunction with the commands used for test execution found in part C. In section Ca it provides instructions on how to download the configuration files used for this project and that are used in the commands seen in sections Cb, Cc, and Cd.

Part B of this section details how to install the mounted distributable to your local Heracles directory. Part C is a breakdown of how to execute the benchmarks and is broken down into further sections. The other sections explain how to download our configuration files and where they should be located as well as steps to build benchmarks without running them, running individual tests, and running whole test suites. To run a quicker test in order to see the whole process without waiting for an entire suite to finish, see part C, section b for an example of how to run an individual test from the benchmark suite.

You can monitor the cluster and find an open node number on Heracles here
https://heracles.ucdenver.pvt/mcms/

A list of all the individual tests offered by SPEC CPU2017 can be found here
https://www.spec.org/cpu2017/Docs/overview.html#Q13.

A list of test suites offered by SPEC CPU2017 can be found here
https://www.spec.org/cpu2017/Docs/overview.html#Q12.

*B. Installation*

1. Create a folder in your local Heracles directory to be used as an installation destination. The directory name ‘spec2017’, is used throughout this section as an example.
   **cd /home/firstname.lastname**
   **mkdir spec2017**

2. Change directories to the location of the mounted distributable
   **cd /mnt**

3. Execute install.sh and point the installation to the directory you created in step 1
   **./install.sh –d /home/firstname.lastname/spec2017**

4. When prompted verify the installation info and type ‘yes’ if correct
5. In order to use the Intel compiler you need to copy the license file to your local directory.
   **cp /opt/intel/licenses/heracles.lic /home/firstname.lastname/**

*C Test Execution*

a. *Downloading Configuration Files*
   Before executing tests a configuration file is needed to define the options used for test execution and to describe the system under test for the purpose of logging. These steps detail how to download the configuration files used for this project and where to put them on the Heracles system to prepare you for testing.

   1. Download the .zip of the SPEC configuration files from the below repository and extract all contents. Zip also available from the website where you got this paper.
      **https://github.com/moorStef/SPEC_Configuration_Files.git**
      **https://peverwhee.github.io/architecture/**

   2. Using a FTP client, transfer the extracted contents to the config file found at the location below
      **/home/firstname.lastname/spec2017/config**

      The configuration files used for this project are:
      Heracles_O0.cfg, Heracles_O1.cfg,
      Heracles_O2.cfg, Heracles_O3.cfg,
      Heracles_O3_proc_specific.cfg*, Heracles_Ofast.cfg,
      Heracles_Os.cfg, Heracles_Intel.cfg

   *Heracles_O3_proc_specific.cfg is the O3+march test referred to in this paper.

b. *Building Tests*
   In order to get build times in a more efficient manner this section details how to build the tests without executing the full test and how you can ‘clean up’ afterwards to start fresh for each new build. In order to do this section you must have completed the previous sections (B and Ca).

   1. After logging into the Heracles cluster, change to you installation directory
      **cd /home/firstname.lastname/spec2017**

   2. Execute the following command and verify that no errors occur. (This will need to be done every time you log into Heracles)

**source shrc**

If error occurs try
**. ./shrc** → *that is dot-space-dot-slash-shrc*

3. a. If you wish to run a test using the Heracles_Intel.cfg file, which uses the Intel compiler you must first move the license file located at /opt/intel/licenses to your local directory /home/firstname.lastname/
   **cd /opt/intel/licenses**
   **cp heracles.lic /home/firstname.lastname**

   b. set the INTEL_LICENSE_FILE environment variable. (This will need to be done every time you log into Heracles)
   **INTEL_LICENSE_FILE=/home/firstname.lastname/**

4. Using SLURM, perf, and runcpu, build a test with the following command. The following example builds the 520.omnetpp_r test. If you wish to run this command for other individual tests or test suites just exchange 520.omnetpp_r with the name of the test or suite you would like to build.

   **srun -n1 -w "node[node #]" perf stat -d runcpu --config=Heracles_O0 --action=build 520.omnetpp_r**

5. If you wish to start with a fresh by cleaning old run and build directories as well as corresponding executables you can run the following example that is scrubing the 520.omnetpp_r test. If you wish to run this command for other individual tests or test suites just exchange 520.omnetpp_r with the name of the test or suite you would like to scrub.

   **srun -n1 -w "node[node #]" perf stat -d runcpu --config=Heracles_O0 --action=scrub 520.omnetpp_r**

c. *Executing Individual Tests*
This section details how you can run individual tests that are found in the test suites. In order to successfully execute tests you must have completed sections B and Ca.

1. After logging into the Heracles cluster change to your installation directory
   **cd /home/firstname.lastname/spec2017**

2. Execute the following command and verify that no errors occur. (This will need to be done every time you log into Heracles)
   **source shrc**

   If error occurs try
   **. ./shrc** → *that is dot-space-dot-slash-shrc*

3. a. If you wish to run a test using the Heracles_Intel.cfg file, which uses the Intel compiler you must first move the license file located at /opt/intel/licenses to your local directory /home/firstname.lastname/
   **cd /opt/intel/licenses**
   **cp heracles.lic /home/firstname.lastname**

   b. set the INTEL_LICENSE_FILE environment variable. (This will need to be done every time you log into Heracles)
   **INTEL_LICENSE_FILE=/home/firstname.lastname/**

4. Using SLURM, perf, and runcpu, execute a single benchmark with the following command. Substitute the configuration file name to be the optimization you wish to test (a list of configuration files is found in section 3a). If you wish to run a different test, substitute the 505.mcf_r test name with the name of the test you wish to run.

   **srun –n1 –w “node[<node#>]” perf stat -d runcpu --reportable --config=Heracles_xx --iterations=1 --action=run 505.mcf_r**

d. *Executing a Full Test Suite*
For this project the intrate test suite is used as seen in step 3. Other test suites can be executed with the same command by interchanging intrate with the name of another test suite.

1. Change to your installation directory
   **cd /home/firstname.lastname/spec2017**

2. Execute the following command and verify that no errors occur. (This will need to be done every time you log into Heracles)
   **source shrc**

   If error occurs try:
   **. ./shrc** → *that is dot-space-dot-slash-shrc*

3. a. If you wish to run a test using the Heracles_Intel.cfg file, which uses the Intel compiler you must first move the license file located at /opt/intel/licenses to your local directory /home/firstname.lastname/
   **cd /opt/intel/licenses**
   **cp heracles.lic /home/firstname.lastname**

   b. set the INTEL_LICENSE_FILE environment variable. (This will need to be done every time you log into Heracles)
   **INTEL_LICENSE_FILE=/home/firstname.lastname/**

4. Using SLURM, perf, and runcpu, execute the following command.
   **srun –n1 –w "node[<node#>]" perf stat -d runcpu --config=Heracles_O2 --reportable --size=ref --iterations=1 --action=run intrate**

## References

[1] R. D. Escobar, A. R. Angula, and M. Corsi, "Evaluation of GCC Optimization Parameters", in *Ing. USBMed,* Vol. 3, No.2, 2012, pp. 31-39. [Online]. Available: http://web.usbmed.edu.co/usbmed/fing/v3n2/v3n2a4.pdf.

[2] "Options that Control Optimizations." *Using the GNU Compiler Collection.* 2018. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[3] Henning J. "SPEC CPU2017 Overview/What's New." *Standard Performance Evaluation Corporation.* 2017. [Online]. Available: https://www.spec.org/cpu2017/Docs/overview.html.

[4] Wall, L. "500.perlbench_r SPEC CPU2017 Benchmark Description." May, 2017. https://www.spec.org/cpu2017/Docs/benchmarks/500.perlbench_r.html.

[5] Stallman, R. "502.gcc_r SPEC CPU2017 Benchmark Description." April, 2018. https://www.spec.org/cpu2017/Docs/benchmarks/502.gcc_r.html.

[6] Lobel, A. "505.mcf_r SPEC CPU2017 Benchmark Description." April, 2018. https://www.spec.org/cpu2017/Docs/benchmarks/500.perlbench_r.html.

[7] Rousselot, J., Varga, A., Horning, R., "520.omnetpp_r SPEC CPU Benchmark Description." May, 2017. https://www.spec.org/cpu2017/Docs/benchmarks/520.omnetpp_r.html.

[8] Wong, M., Cambly, C. "523.xalancbmk_r SPEC CPU2017 Benchmark Description." IBM Corporation. May, 2017. https://www.spec.org/cpu2017/Docs/benchmarks/523.xalancbmk_r.html.

[9] Garrett-Glaser, J. "525.x264_r SPEC CPU2017 Benchmark Description." May, 2017. https://www.spec.org/cpu2017/Docs/benchmarks/525.x264_r.html.

[10] Pascutto, G. "531.deepsjeng_r SPEC CPU2017 Benchmark Description." https://www.spec.org/cpu2017/Docs/benchmarks/531.deepsjeng_r.html.

[11] Pascutto, G. "541.leela_r SPEC CPU2017 Benchmark Description." April, 2016. https://www.spec.org/cpu2017/Docs/benchmarks/541.leela_r.html.

[12] Metcalf, M. "548.exchange2_r SPEC CPU2017 Benchmark Description." September, 2017. https://www.spec.org/cpu2017/Docs/benchmarks/548.exchange2_r.html.

[13] Collin, L., Pavlov, I., Novy, J. "557.xz_r SPEC CPU2017 Benchmark Description." September, 2017. https://www.spec.org/cpu2017/Docs/benchmarks/557.xz_r.html.

[14] Andersson, U. "549.fotonik3d_r SPEC CPU Benchmark Description." https://www.spec.org/cpu2017/Docs/benchmarks/549.fotonik3d_r.html.

[15] J. Pallister, S. Hollis, and J. Bennett, “Identifying Compiler Options to Minimise Energy Consumption for Embedded Platforms,” Department of Computer Science, University of Bristol, UK. 2013. [Online]. Available: https://arxiv.org/pdf/1303.6485.pdf.

[16] Z. Pan and R. Eigenmann, “Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning,” *Proceedings of the International Symposium on Code Generation and Optimization.* 2006.

[17] J. Kaur and A. Kaur, “Role of Compiler in Computer Architecture,” *International Journal of Engineering Research and General Science*, Vol. 4, Issue 4, 2015, pp. 247-254. [Online]. Available: http://pnrsolution.org/Datacenter/Vol4/Issue3/33.pdf.

[18] Jones, M.  Optimization in GCC. 2005 [online] linuxjournal. Available at: http://www.linuxjournal.com/article/7269?page=0,0.

[19] Fog, A. “The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers,” *Technical University of Denmark*. 2018. [online] http://www.agner.org/optimize/microarchitecture.pdf

## X. Appendices

## A. intrate

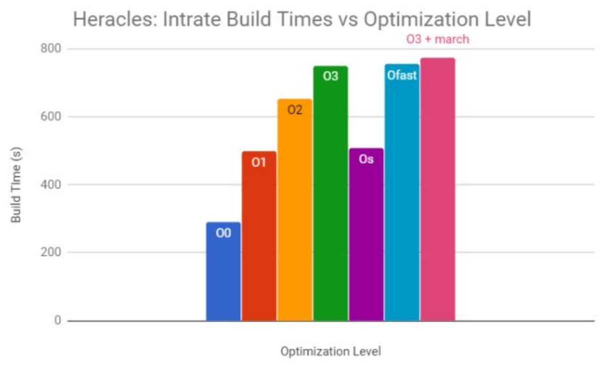


***Figure A1:*** *intrate build times vs optimization level*

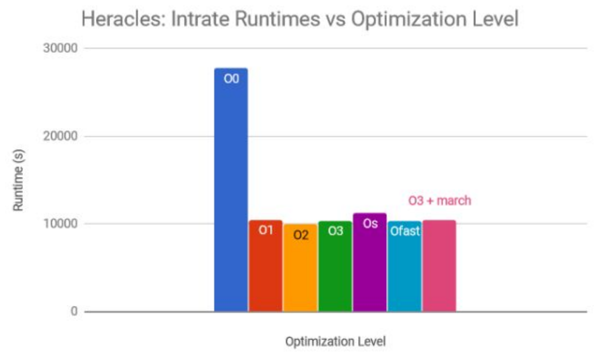


***Figure A2:*** *Intrate runtimes versus optimization levels*

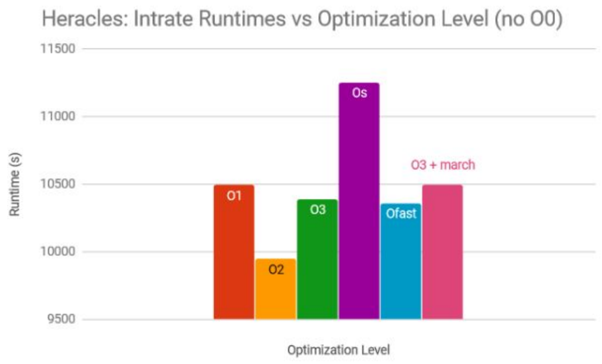


***Figure A3:*** *Intrate runtimes versus optimizations level (no O0)*

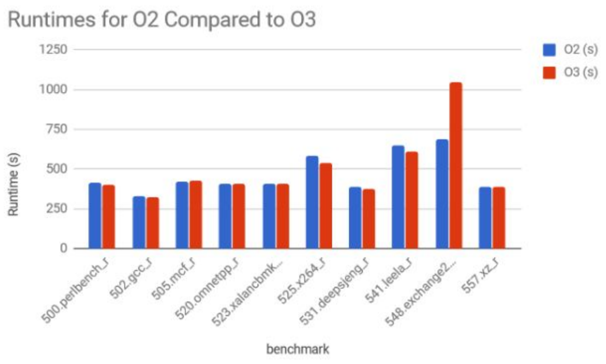


***Figure A4:*** *Intrate individual test runtimes - O2 vs O3*

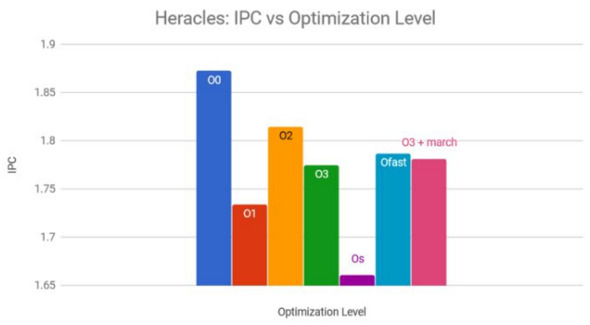


***Figure A5:*** *Instructions per cycle versus optimizations levels*

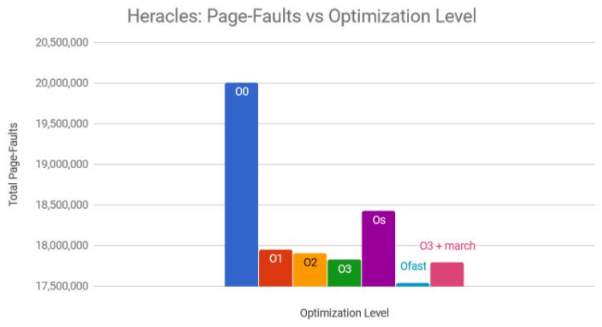


***Figure A6:*** *Page-faults versus optimization levels*

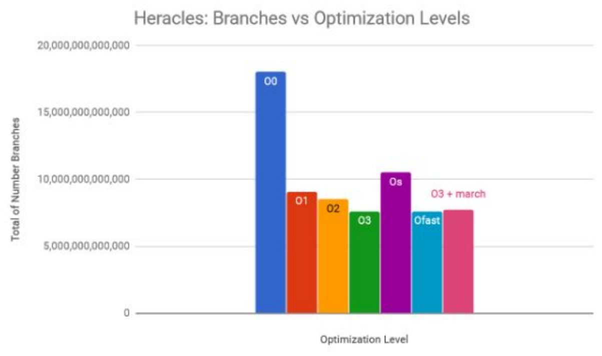


***Figure A7:*** *Intrate Branches versus optimizations levels*

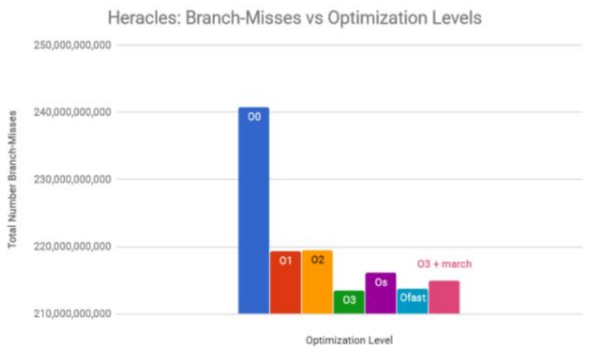


***Figure A8:*** *Intrate branch-misses vs optimization levels*

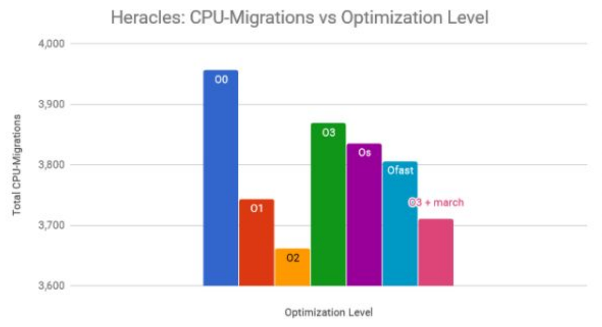


***Figure A9:*** *Intrate CPU-migrations vs optimizations levels*

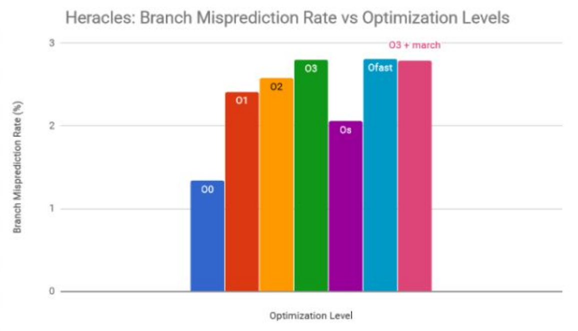


***Figure A10:*** *Intrate branch misprediction rate vs optimizations*

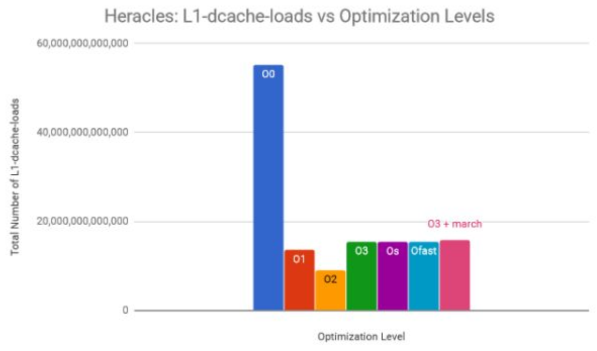


***Figure A11:*** *L1-dcache-loads vs optimization levels*

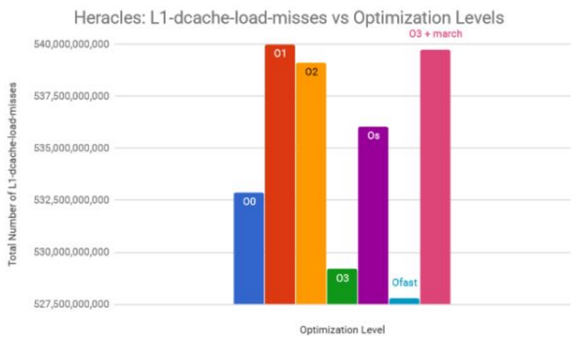


***Figure A12:*** *L1-dcache-load-misses vs optimizations*

## B. 500.perlbench_r

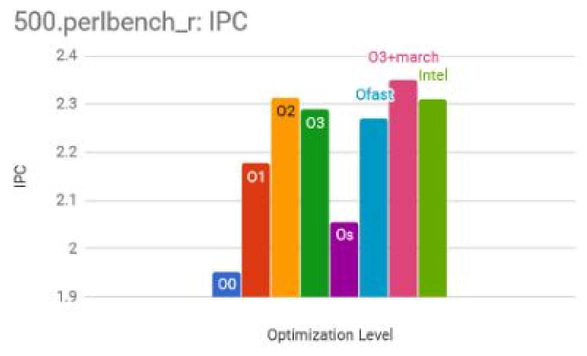


***Figure B1:*** *Perlbench IPC versus optimizations*

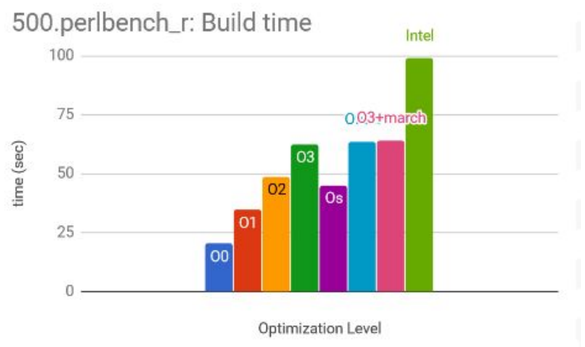


***Figure A2:*** *Perlbench Build times*

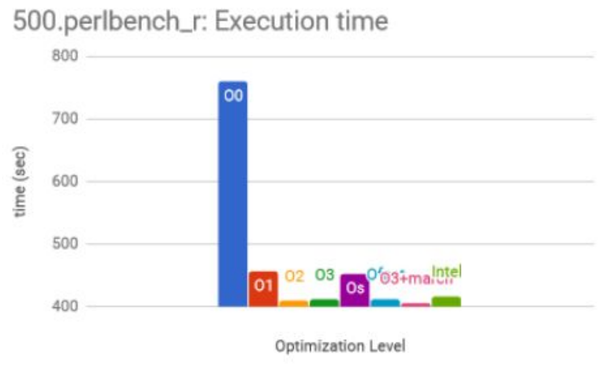


***Figure B3:*** *perlbench execution time vs optimizations*

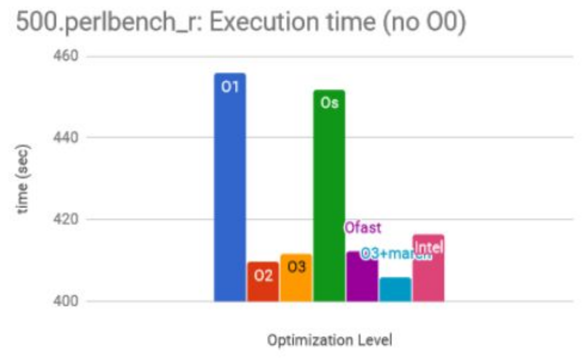


***Figure B4:*** *Perlbench execution time (no O0)*

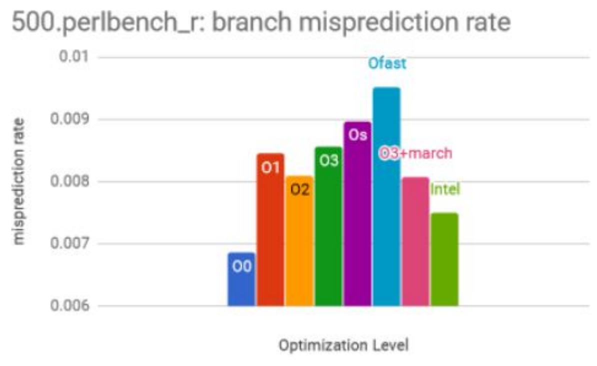


***Figure B5:*** *Perlbench branch misprediction rate*

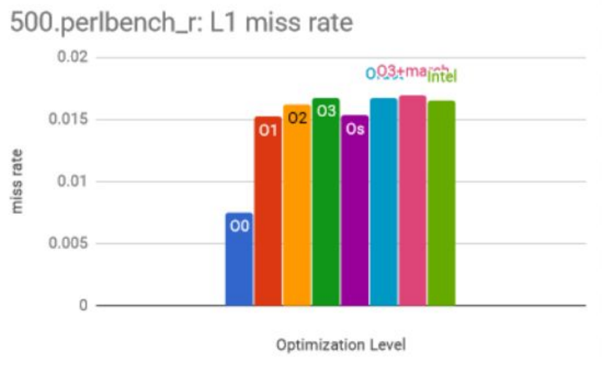


***Figure B6:*** *perlbench L1 miss rate*

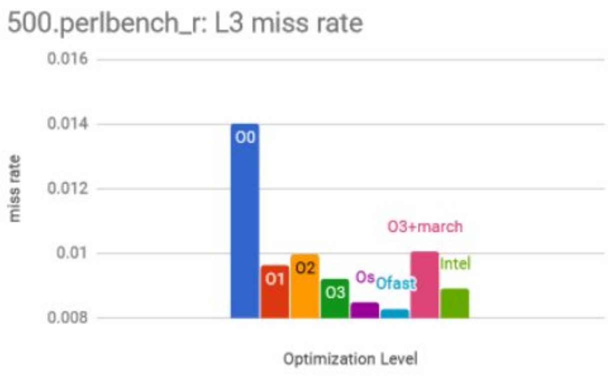


***Figure B7:*** *perlbench L3 miss rate*

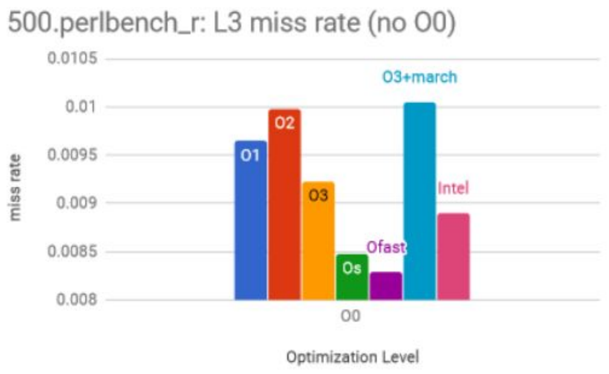


***Figure B8:*** *perlbench L3 miss rate (no O0)*

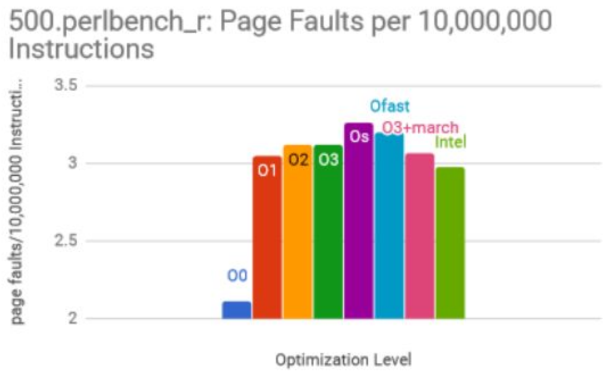


***Figure B9:*** *perlbench page faults per 10 million instructions*

## C. 502.gcc_r

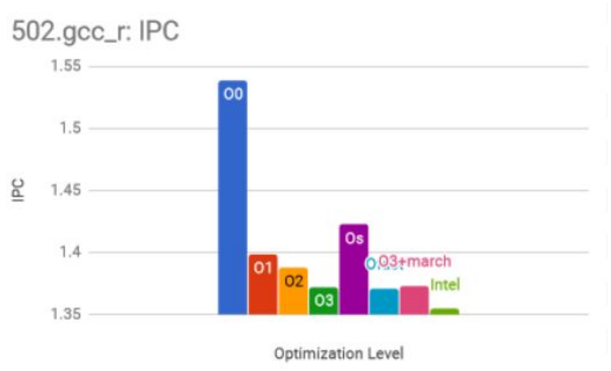


***Figure C1:** gcc IPC versus optimizations*

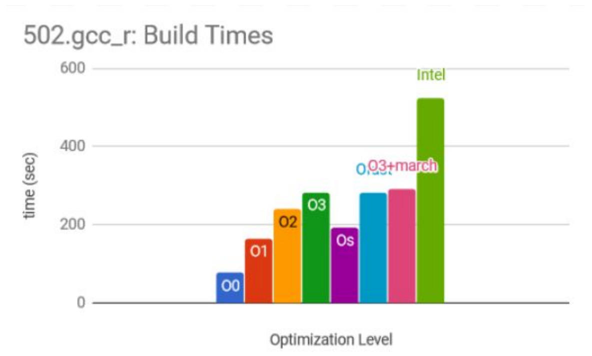


***Figure C2:** gcc build times vs optimizations*

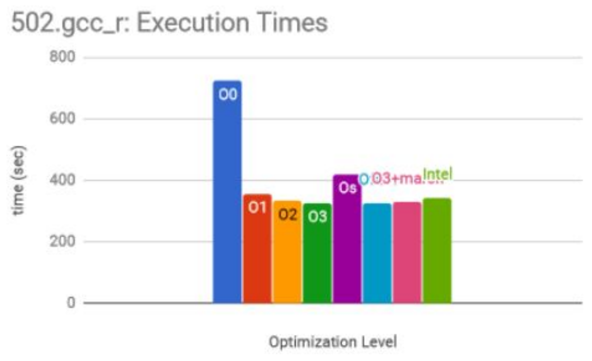


***Figure C3:** gcc execution times*

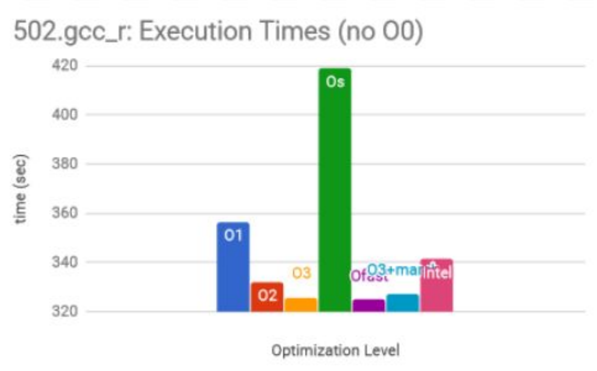


***Figure C4:** gcc execution times (no O0)*

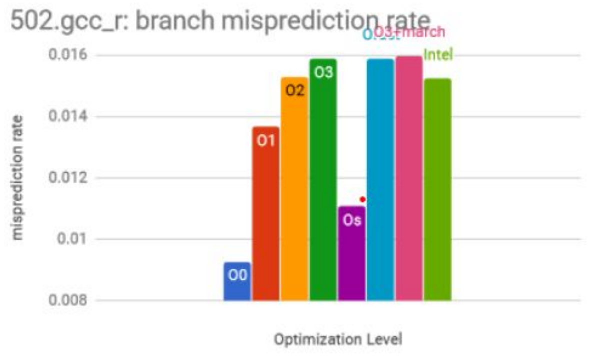


***Figure C5:** gcc branch misprediction rate*

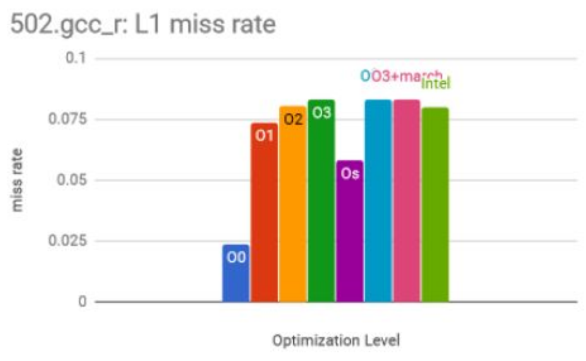


***Figure C6:** gcc L1 miss rate*

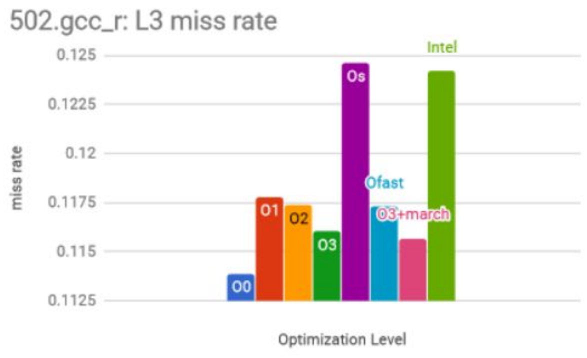


***Figure C7:*** *gcc L3 miss rate*

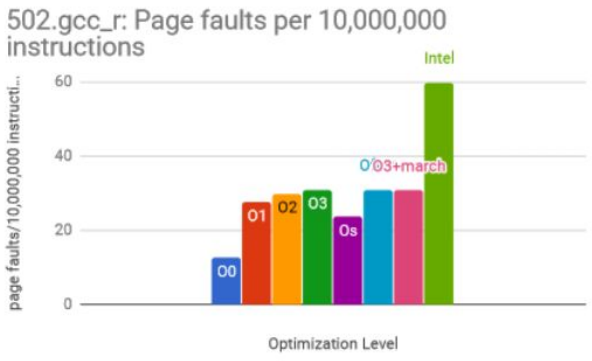


***Figure C8:*** *gcc page faults per 10 million instructions*

## D. 505.mcf_r

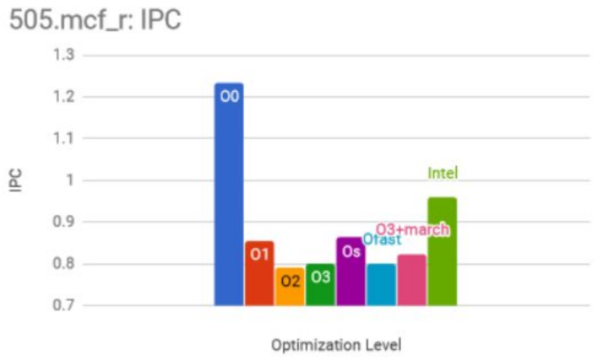


***Figure D1:*** *mcf IPC vs optimization levels*

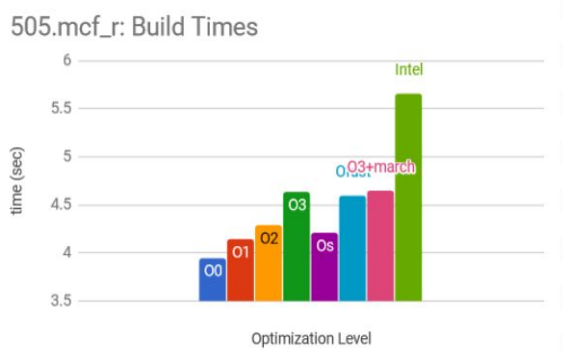


***Figure D2:*** *mcf build times vs optimizations*

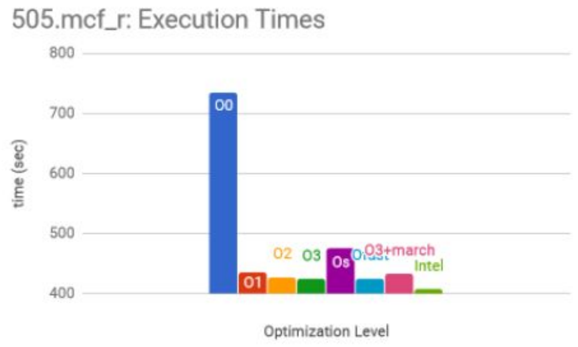


***Figure D3:*** *mcf execution times*

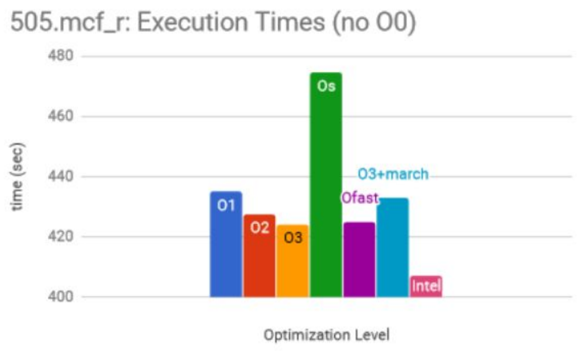


***Figure D4: mcf execution times (no O0)***

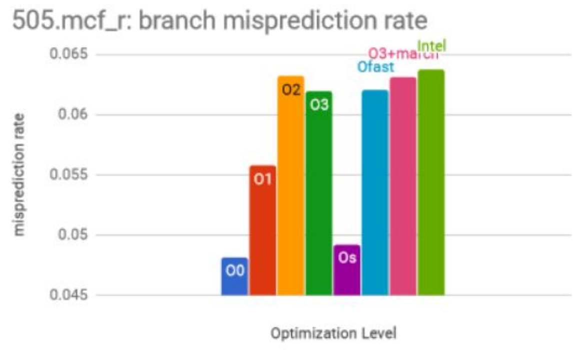


***Figure D5:*** *mcf branch misprediction rate*

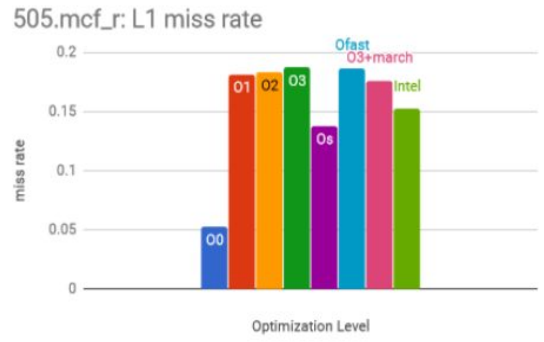


***Figure D6:*** *mcf L1 miss rate*

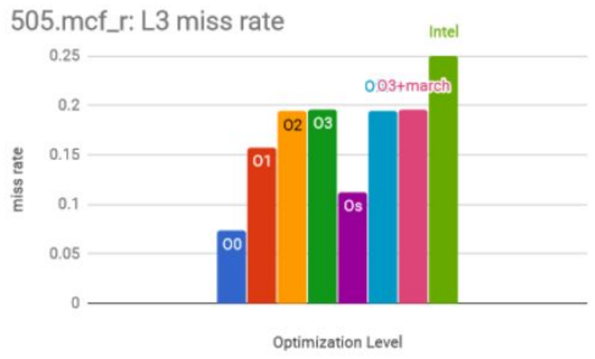


***Figure D7:*** *mcf L3 miss rate*

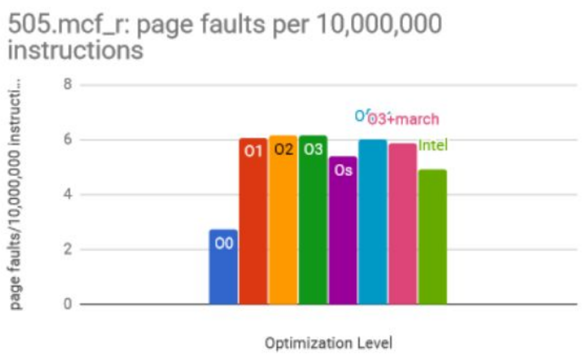


***Figure D8:*** *mcf page faults per 10 million instructions*

## E. 520.omnetpp_r

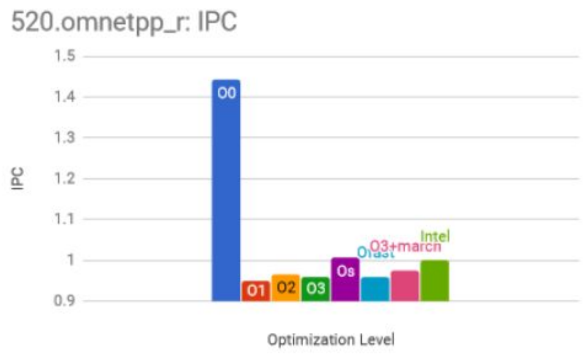


***Figure E1:*** *omnetpp IPC*

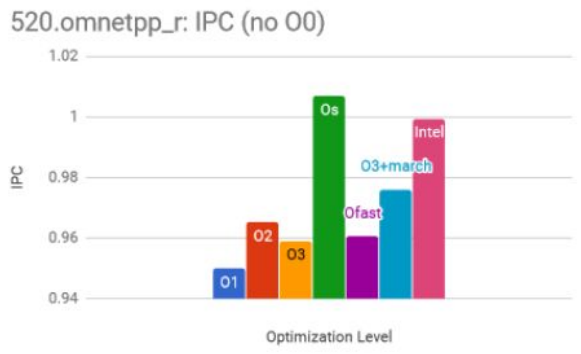


***Figure E2:*** *omnetpp IPC (no O0)*

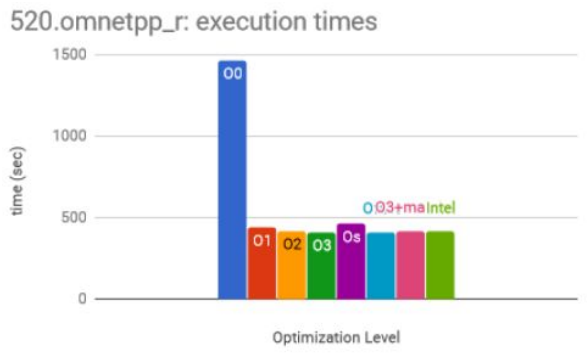


***Figure E3:*** *omnetpp execution times*

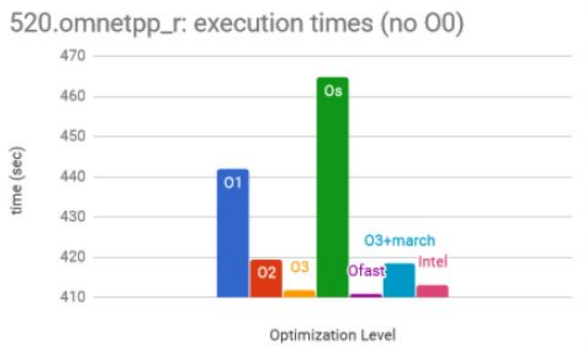


***Figure E4:*** *omnetpp execution times (no O0)*

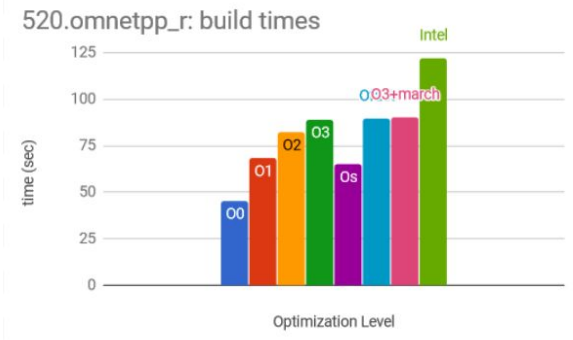


***Figure E5:*** *omnetpp build times*

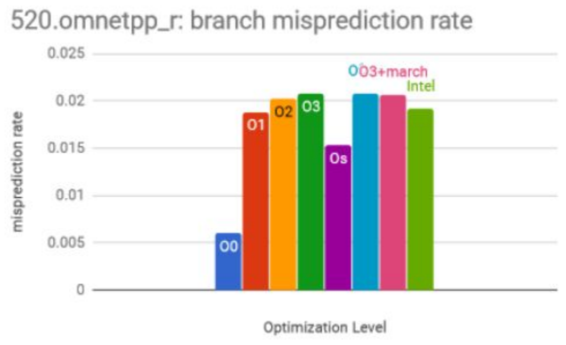


***Figure E6:*** *omnetpp branch misprediction rate*

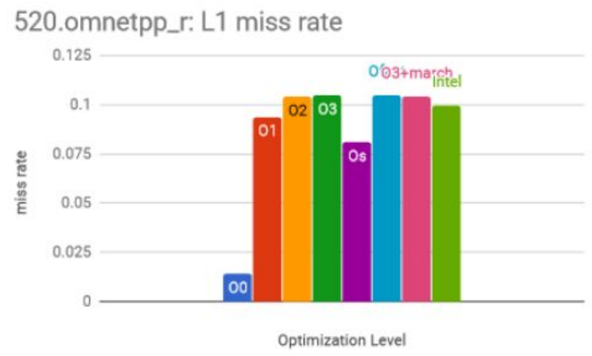


***Figure E7:*** *omnetpp L1 miss rate*

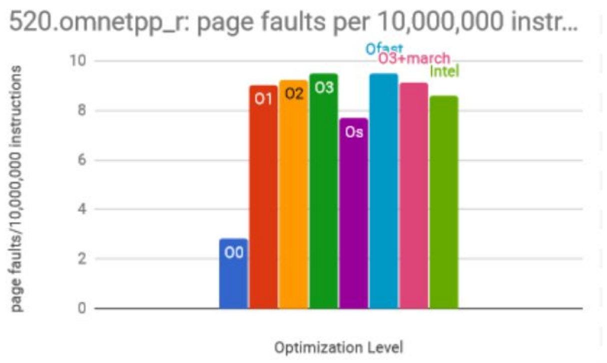


***Figure E8:*** *omnetpp page faults per 10 million instructions*

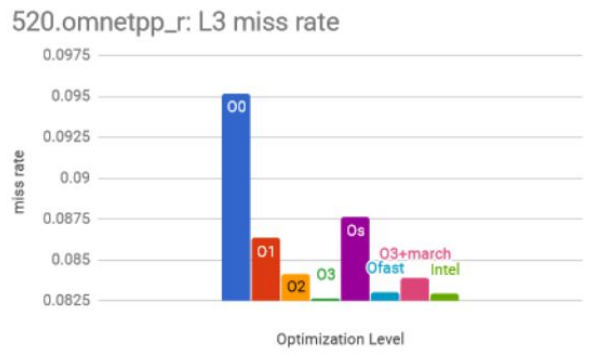


***Figure E9: omnetpp L3 miss rate***

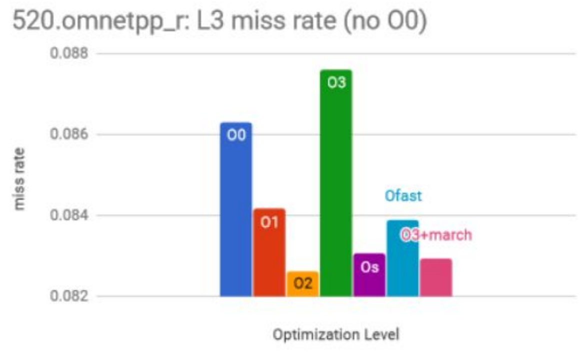


***Figure E10:*** *omnetpp L3 miss rate (no O0)*

## F. 523.xalancbmk_r

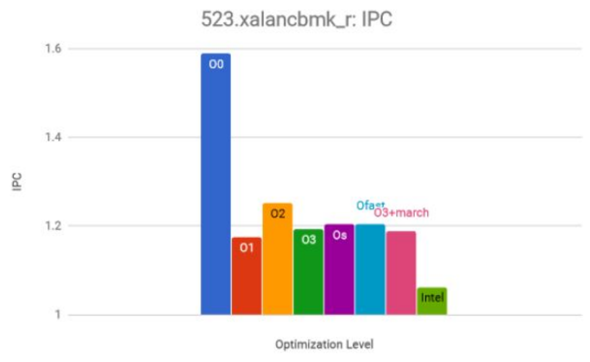


***Figure F1:** xalancbmk IPC vs optimizations*

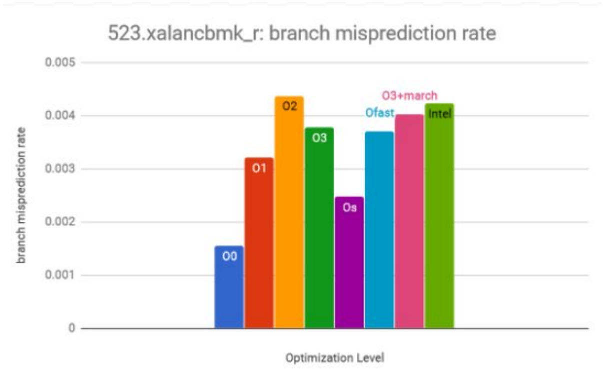


***Figure F2:** xalancbmk branch misprediction rate*

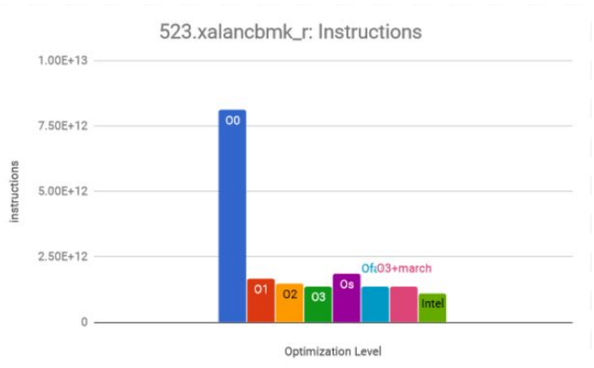


***Figure F3:** xalancbmk number of instructions*

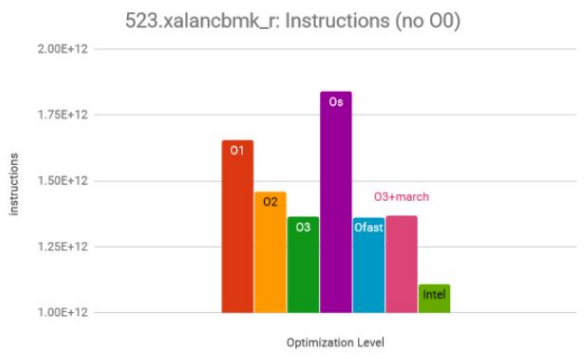


***Figure F4:** xalancbmk number of instructions (no O0)*

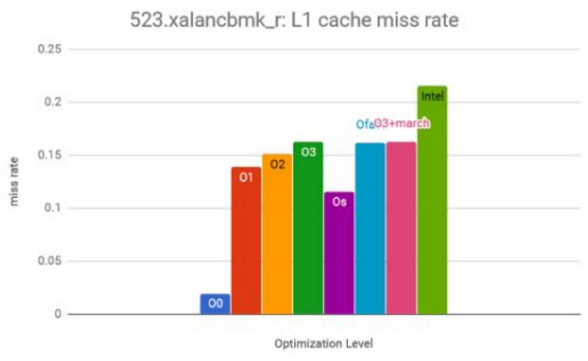


***Figure F5:** xalancbmk L1 cache miss rate*

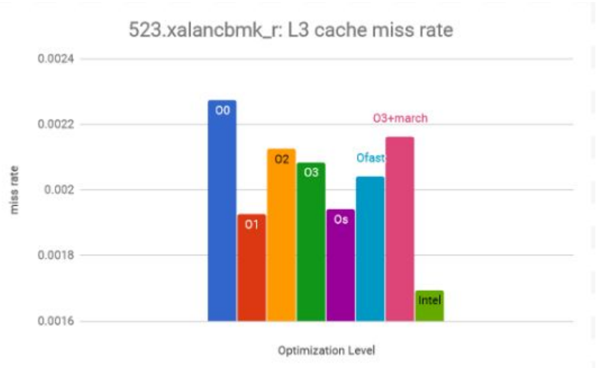


***Figure F6:** xalancbmk L3 cache miss rate*

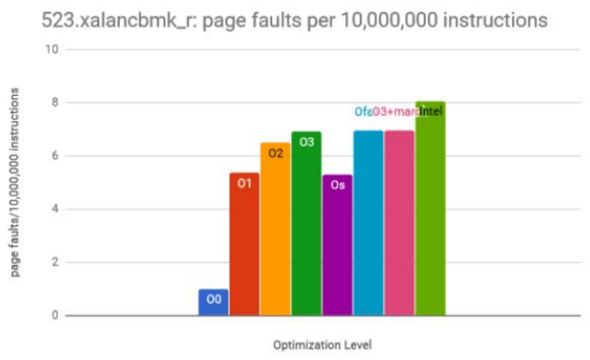


***Figure F7:*** *xalancbmk page faults per 10 million instructions*

## G. 525.x264_r

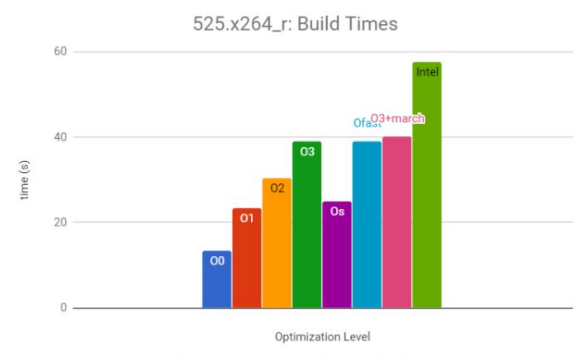


***Figure G1:*** Build times vs optimization level for x264

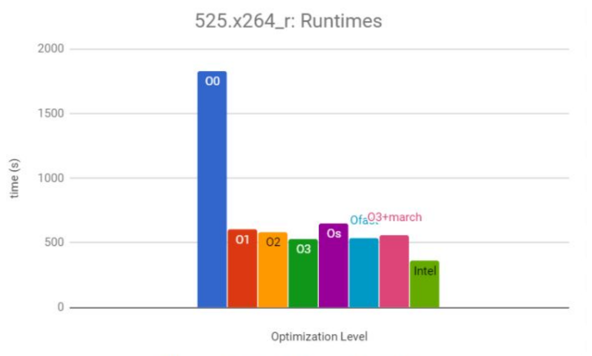


***Figure G2:*** Runtimes for x264

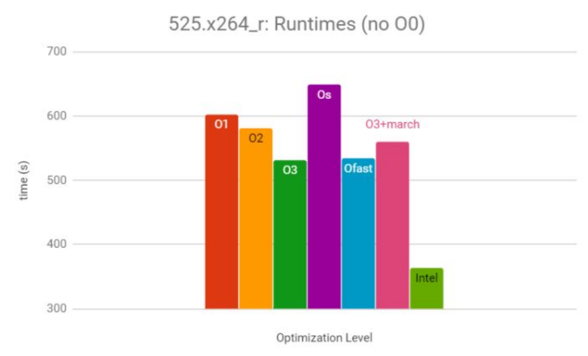


***Figure G3:*** runtimes for x264 (without unoptimized)

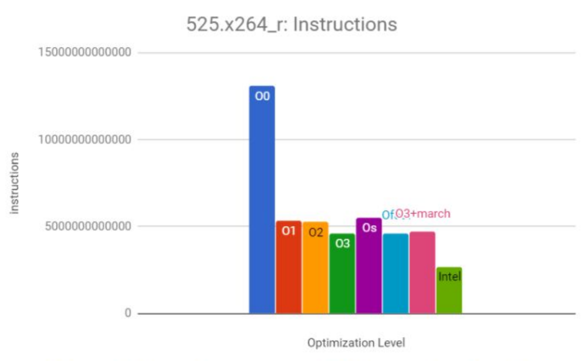


***Figure G4:*** Instructions executed during runtime for x264

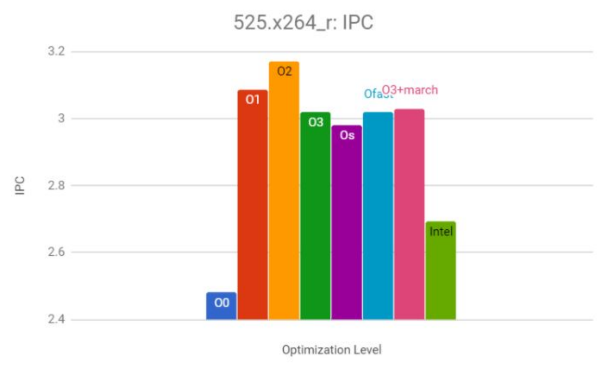


***Figure G5:*** IPC vs optimization level for x264

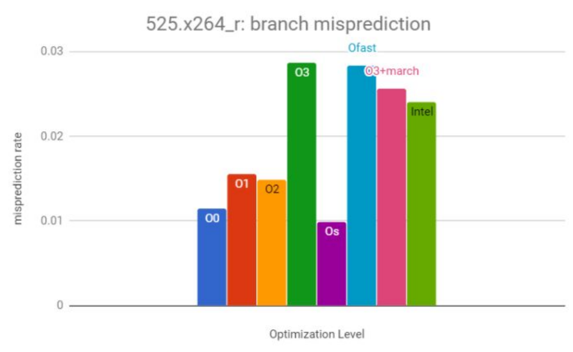


***Figure G6:*** branch misprediction rate for x264

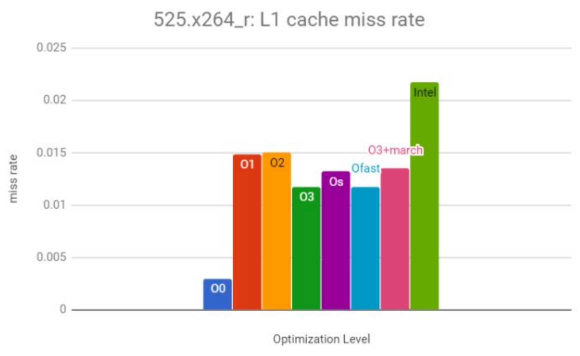


***Figure G7:*** L1 cache miss rate for x264

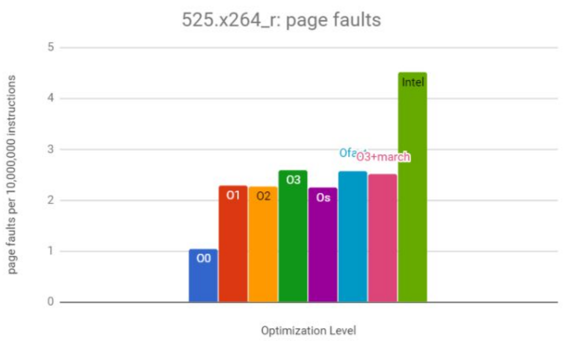


***Figure G8:*** Page faults per 10,000,000 instructions for x264

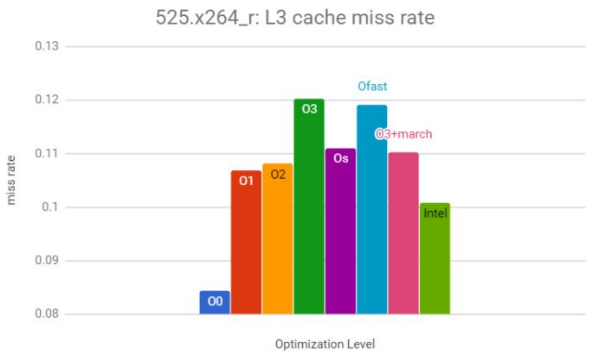


***Figure G9:*** L3 cache miss rate for x264

## H. 531.deepsjeng_r

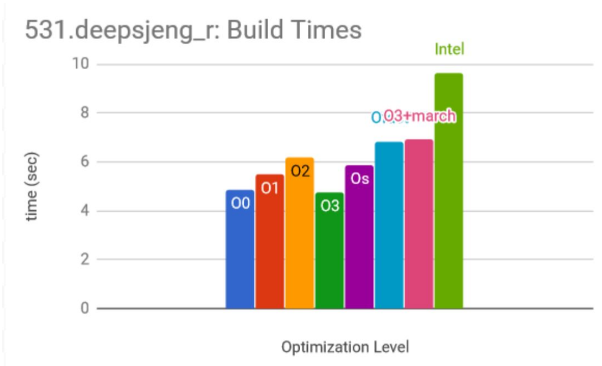


***Figure H1:*** *Build times vs optimization levels*

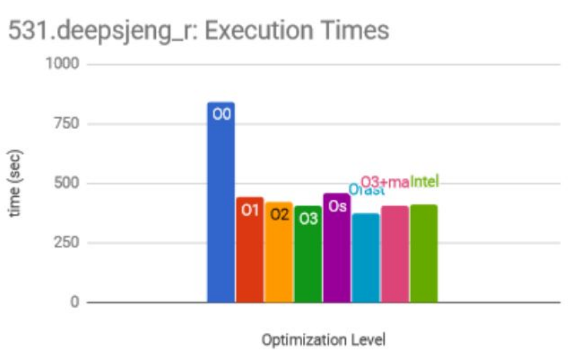


***Figure H2:*** *execution times vs optimization levels*

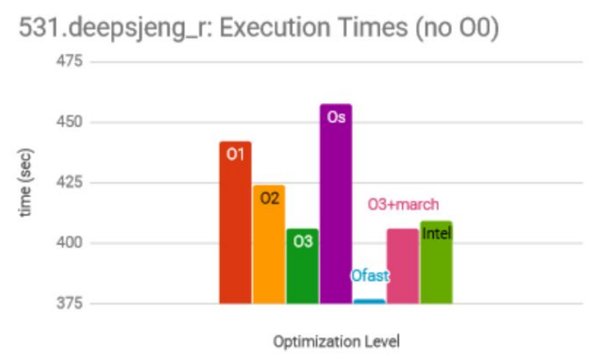


***Figure H3:*** *Execution times (no O0) vs optimizations*

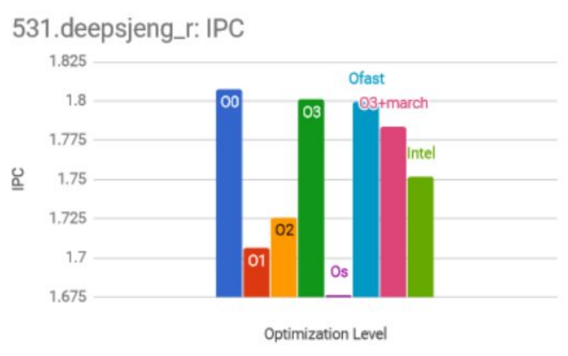


***Figure H4:*** *IPC vs optimization levels*

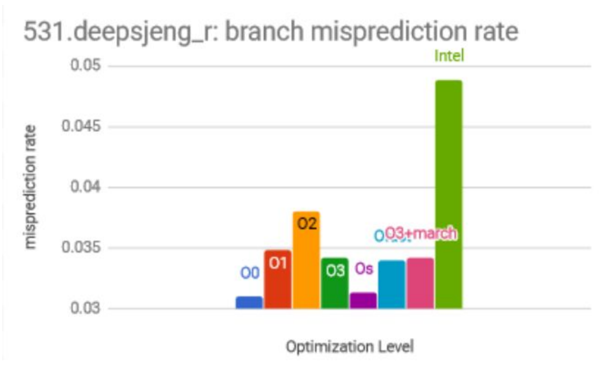


***Figure H5:*** *Branch misprediction rate*

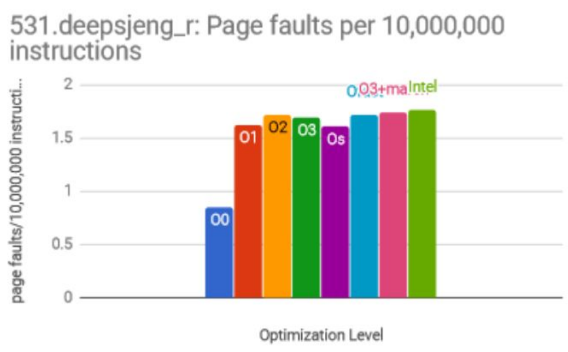


***Figure H6:*** *Page faults per 10 million instructions*

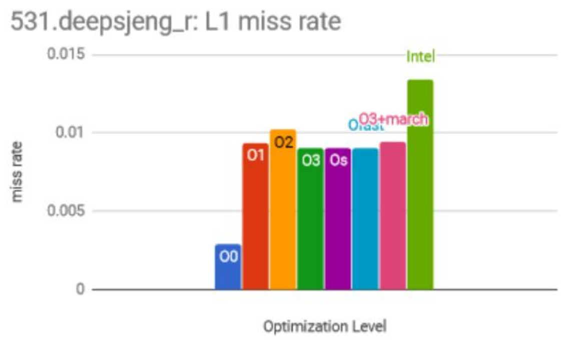


***Figure H7:*** *L1 miss rate vs optimization levels*

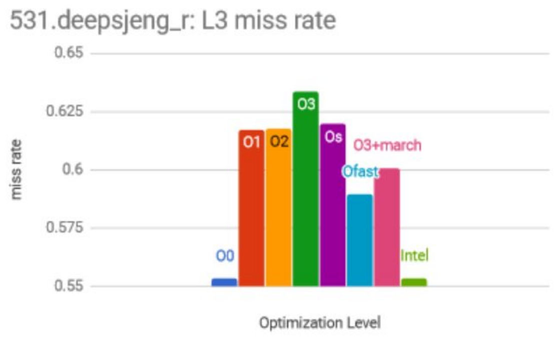


***Figure H8:*** *L3 miss rate vs optimization levels*

## I. 541.leela_r

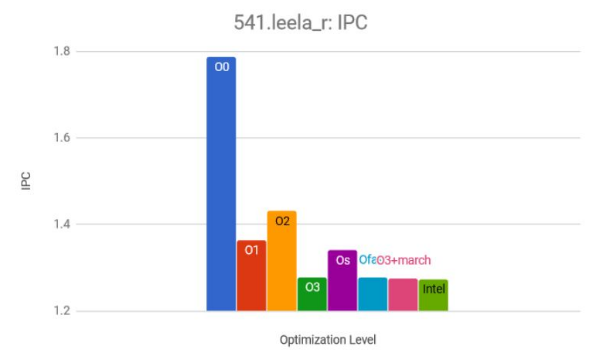


***Figure I1:*** *leela IPC versus optimizations*

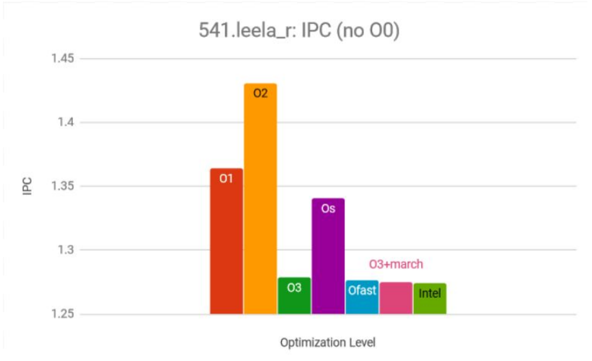


***Figure I2:*** *leela IPC (no O0) versus optimizations*

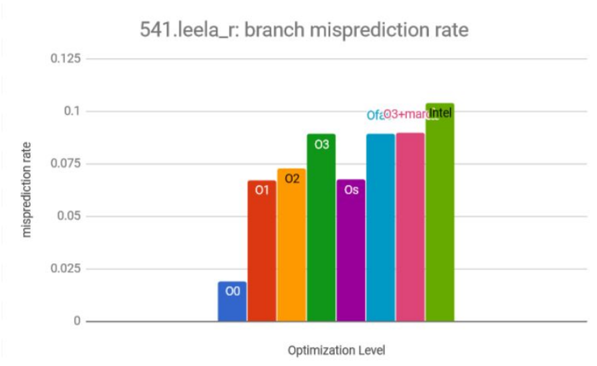


***Figure I3:*** *leela branch misprediction rate*

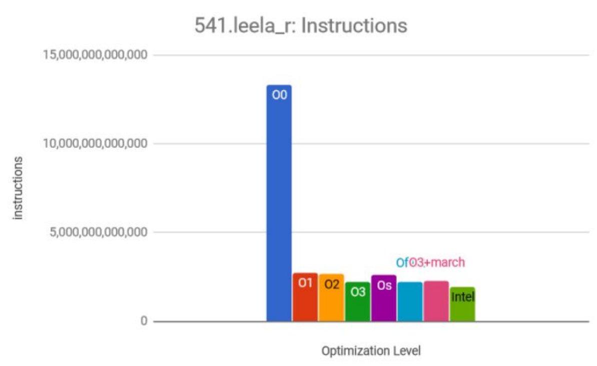


***Figure I4:*** *leela number of instructions vs optimizations*

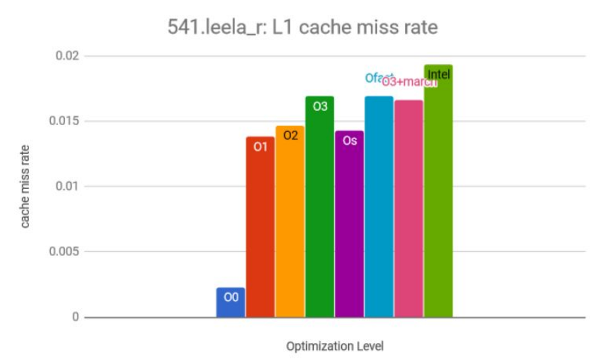


***Figure I5:*** *leela L1 cache miss rate vs optimizations*

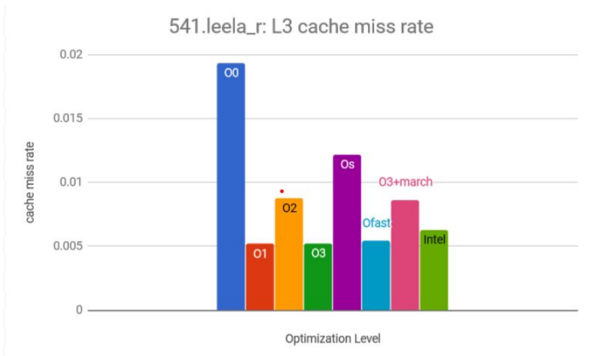


***Figure I6:*** *leela L3 cache miss rate*

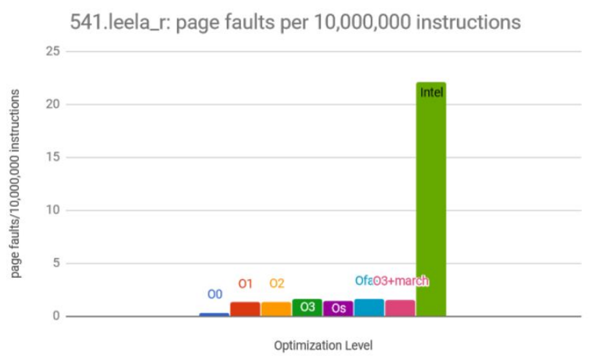


**Figure I7:** *leela page faults per 10 million instructions*

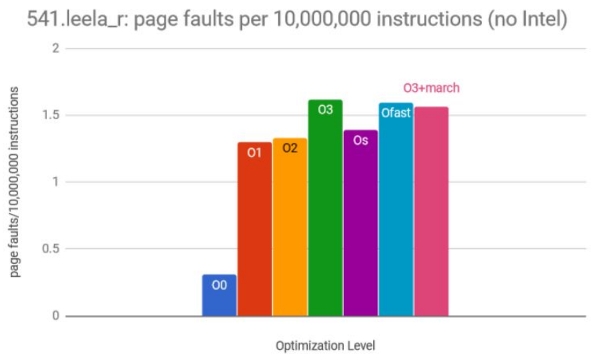


***Figure I8:*** *leela page faults per 10 million instruction (no intel)*

## J. 548.exchange2_r

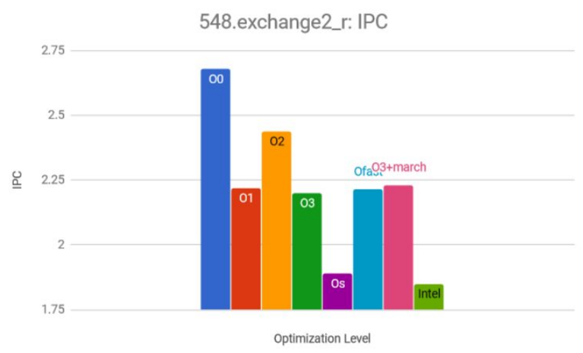


***Figure J1:*** *exchange2 IPC vs optimization levels*

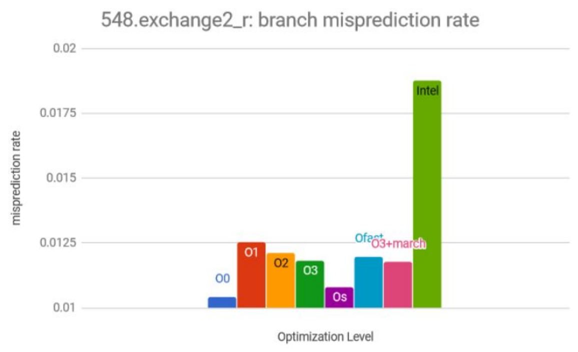


***Figure J2:*** *exchange2 branch misprediction rate*

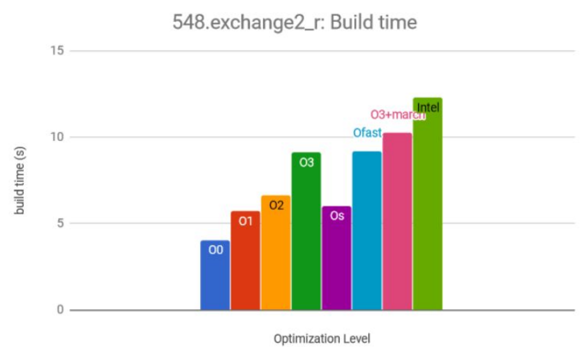


***Figure J3:*** *exchange2 build time vs optimizations*

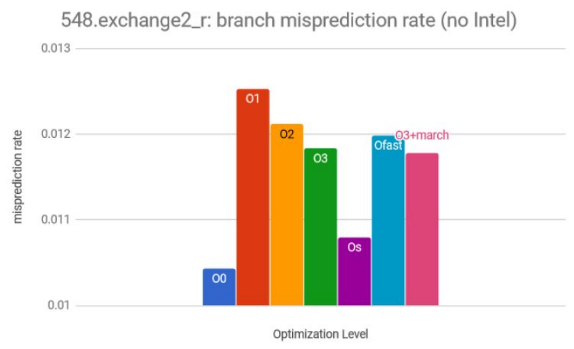


***Figure J4:*** *exchange2 branch misprediction rate (no intel)*

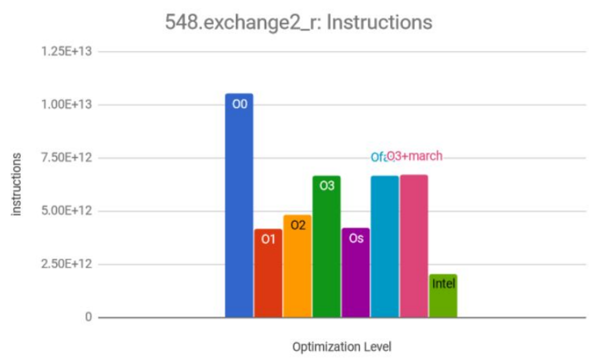


***FIgure J5:*** *exchange2 number of instructions*

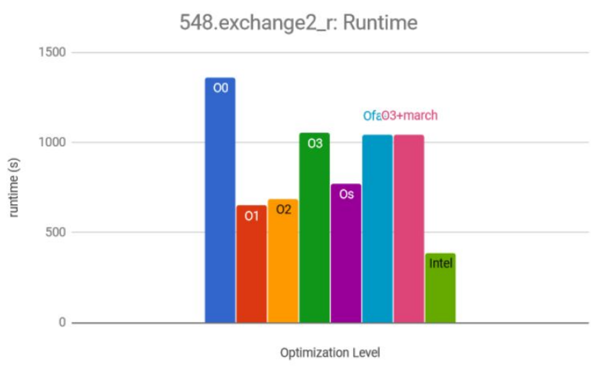


***Figure J6:*** *exchange2 runtime vs optimizations*

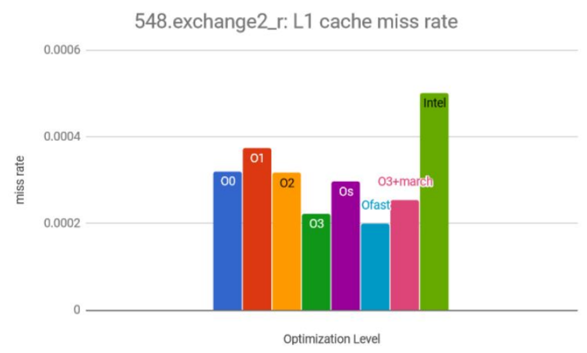


***Figure J7:*** *exchange2 L1 cache miss rate*

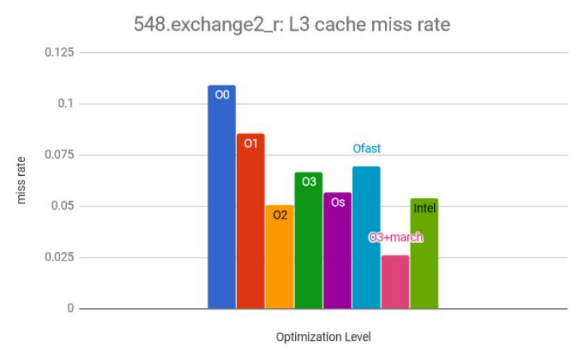


***Figure J8:*** *exchange2 L3 cache miss rate*

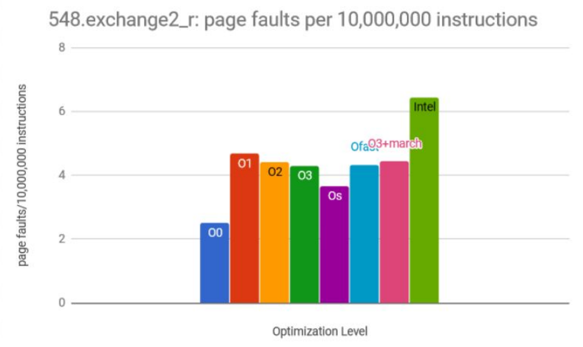


***Figure J9:*** *exchange2 page faults per 10 million instructions*

## K. 557.xz_r

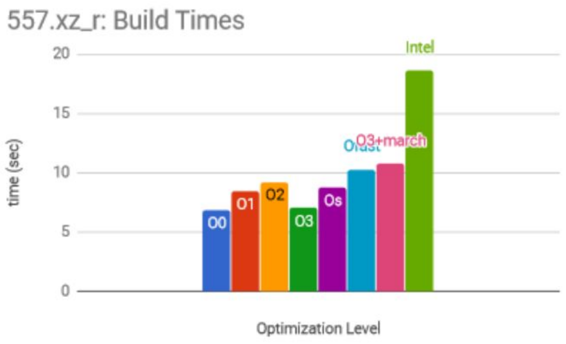


*Figure K1: Build times vs optimization levels*

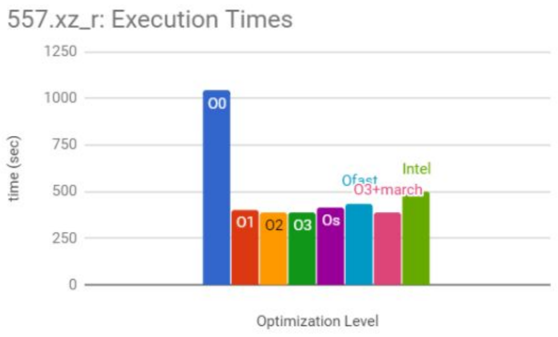


***Figure K2:*** Execution times for xz.

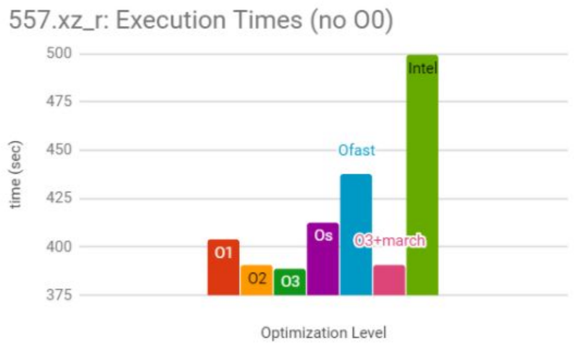


***Figure K3:*** execution times vs optimization (no O0)

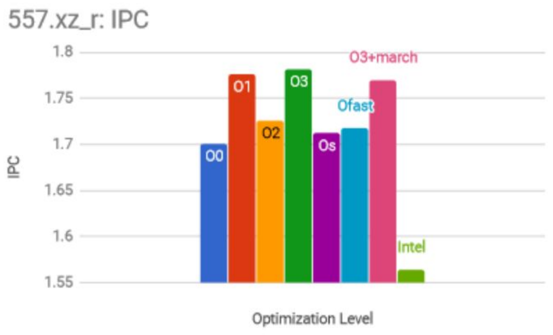


***Figure K4:*** *IPC vs optimizations levels*

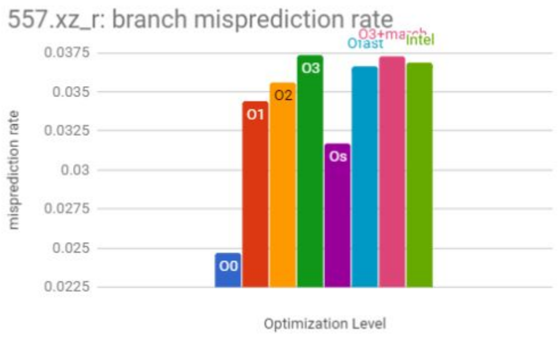


***Figure K5:*** branch misprediction vs optimization

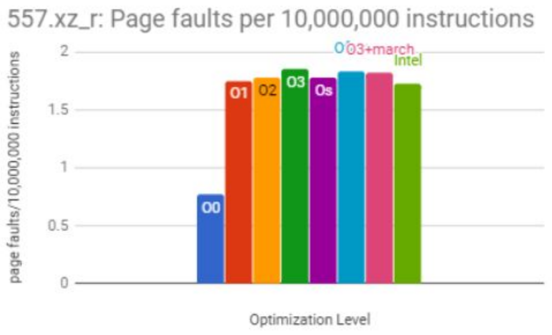


***Figure K6:*** page faults vs optimization level

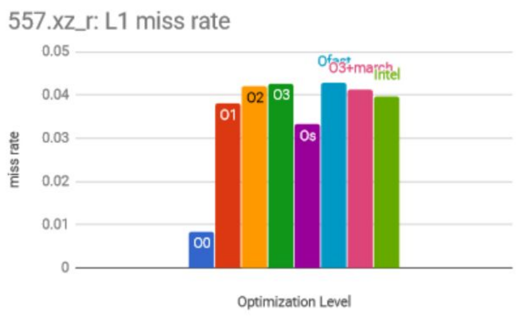


***Figure K7:*** *L1 miss rate*

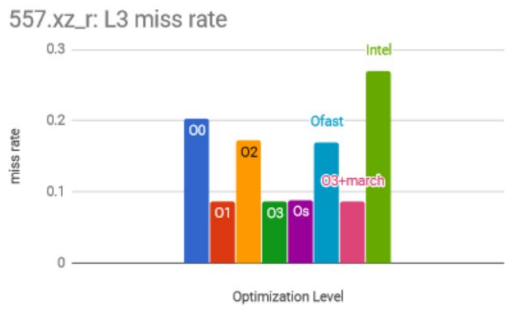


***Figure K8:*** *L3 miss rate*

## L. 549.fotonik3d_r

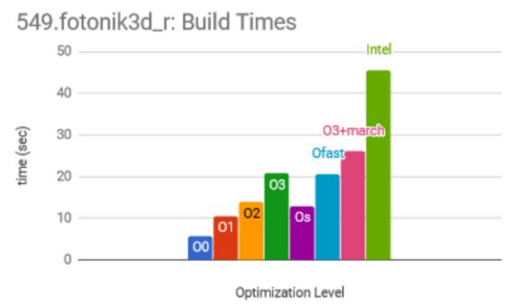


***Figure L1:*** *Build times vs optimization levels*

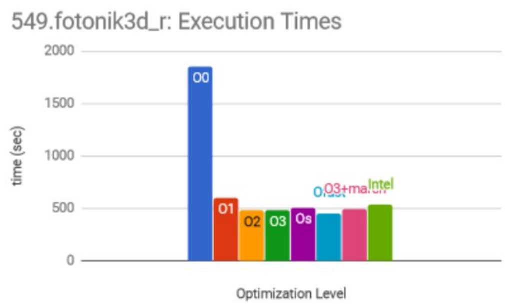


***Figure L2:*** *Execution times vs optimization levels*

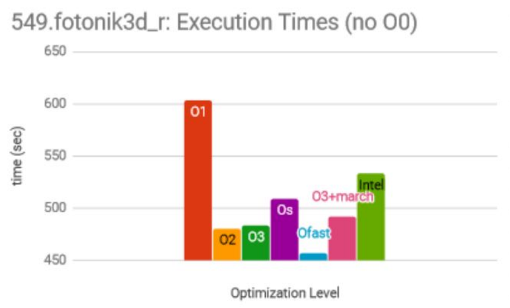


***Figure L3:*** *Execution times (no O0)*

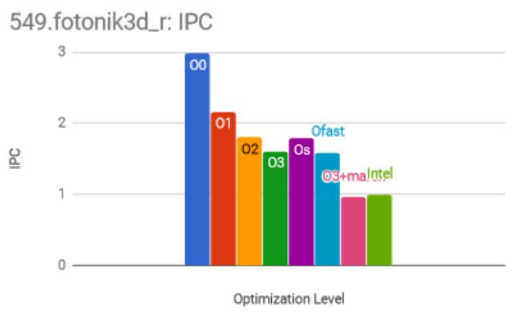


***Figure L4:*** *IPC vs optimization levels*

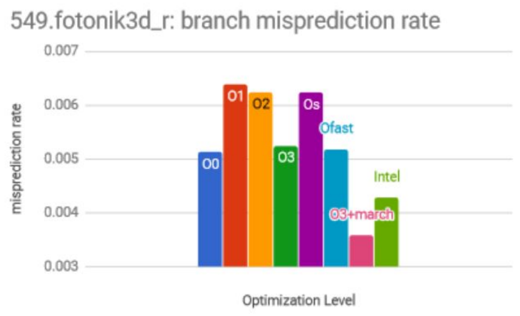


***Figure L5:*** *Branch misprediction rate*

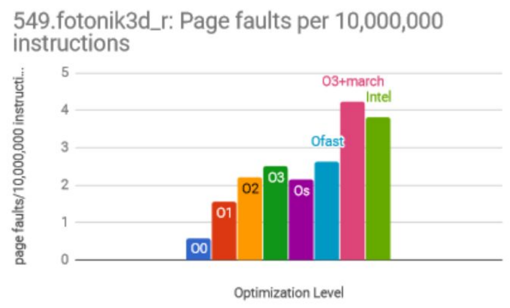


***Figure L6:*** *Page faults per 10 million instructions*

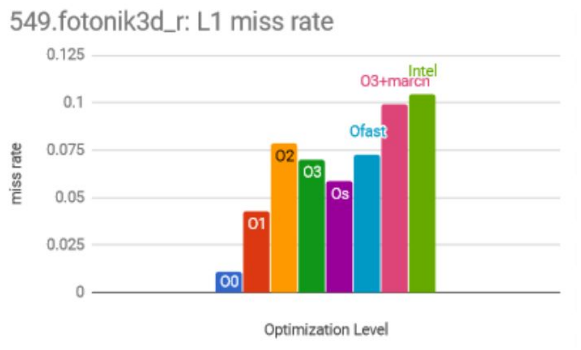


*Figure L7:* *L1 miss rate*

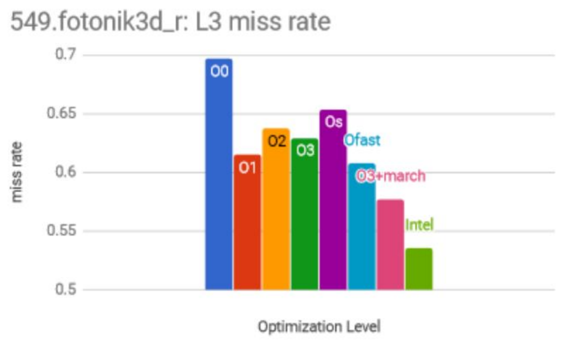


*Figure L8:* *L3 miss rate*

## M. Further Perlbench Analysis

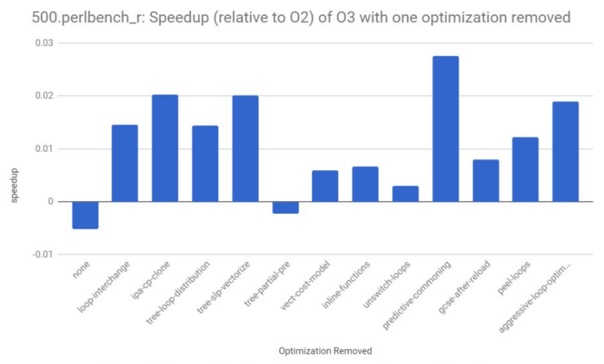


***Figure M1:*** speedup of O3 when removing one optimization

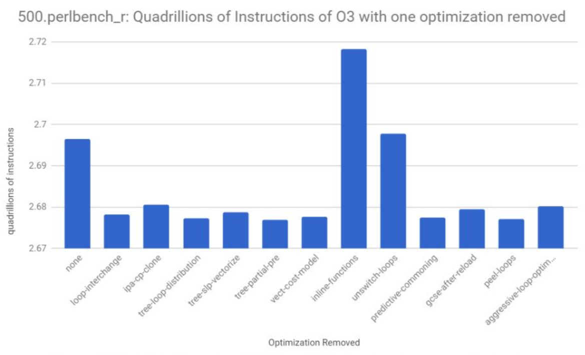


***Figure M2:*** instructions executed by O3 when removing one optimization

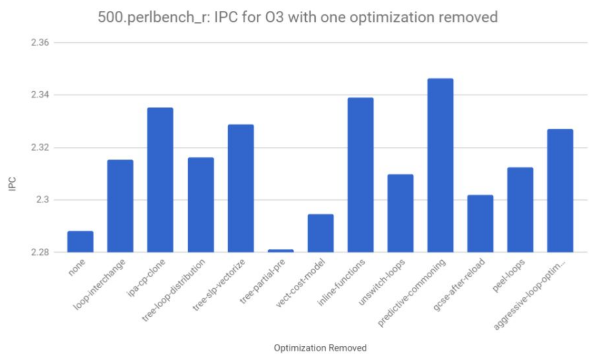


***Figure M3:*** IPC of O3 with one optimization removed

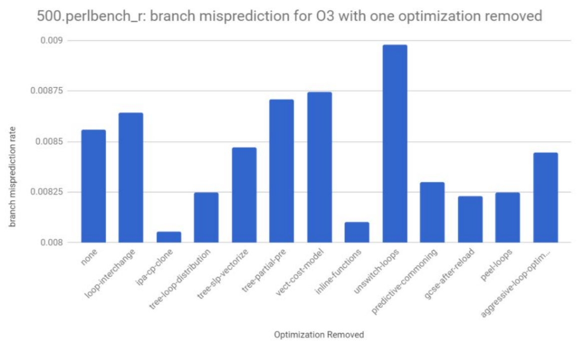


***Figure M4:*** Branch Misprediction for O3 with one optimization removed

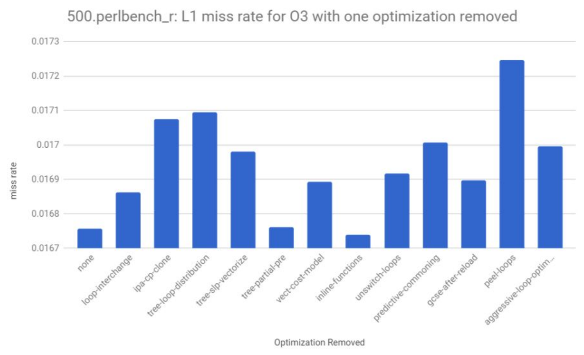


***Figure M5:*** L1 miss rate for O3 with one optimization removed

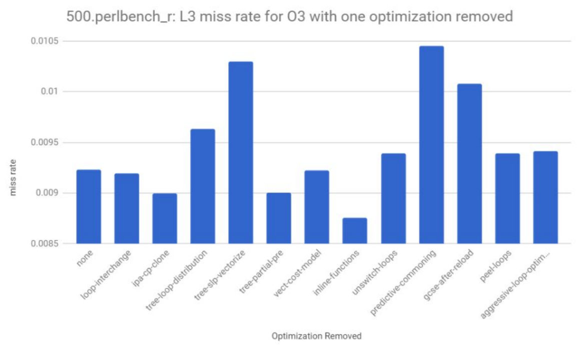


***Figure M6:*** L3 miss rate for O3 with one optimization removed