Performance Impact of Undefined Behavior Optimizations in C/C++

Lucian Popescu[*], Razvan Deaconescu[*] and Nuno Lopes[**]

[*]Facultatea de Automatică și Calculatoare, Universitatea Politehnică din București
[**]Instituto Superior Técnico, Universidade de Lisboa
[*]*lucian.popescu187@gmail.com, razvan.deaconescu@cs.pub.ro*
[**]*nuno.lopes@tecnico.ulisboa.pt*

Saturday 10th June, 2023

Abstract

Clang/LLVM uses undefined behavior to issue optimizations. In this report, we present the impact of this class of optimizations on a benchmarking suite built using Phoronix. In this suite we focus on a diverse set of application categories, ranging from compression algorithms and image processing to web-servers and databases. To extensively cover the impact on these applications, we provide 11 configurations that modify the behavior of the compiler when exploiting undefined behavior in optimizations. Current results show that in TODO: get number of cases the impact of undefined behavior optimizations is insignificant.

1 Introduction

The work in this semester was broken down into two parts. First, we started by creating the infrastructure for running the benchmarks. Our main goal in this regard was to gather a set of popular applications written in C and C++. After exploring existing solutions for this problem, we decided to use the Phoronix Test Suite for this task. The applications provided by Phoronix were later compiled with various configurations that modify the behavior of the compiler when exploiting undefined behavior. For each configuration, we gathered a set of results that we will present later in this work.

Second, we were interested in exploring configurations for undefined behavior exploitation. We started from a set of already-implemented configurations, such as -fwrapv or -fno-strict-aliasing, and moved to configurations implemented by us, such as -fno-constrain-bool-value. We benefited from 5 already-implemented configurations and we managed to implement 6 more configurations. For some of the configurations the level of difficulty was relatively low, because we only had to modify the frontend of the compiler, i.e. Clang. However there existed cases where we also had to modify the middle-end of the compiler, i.e. LLVM.

This work is structured as follows: in Section 1 provides detailed information about the process of creating the benchmarks for our use case, Section 2 talks about each compiler configuration that modifies the behavior of the compiler when exploiting undefined behavior, Section 3 presents the results of the benchmarks for each configuration, and Secton 4 discusses the results.

2 Creating the Benchmarking Infrastructure

To simplify the benchmarking process, we needed an infrastructure that would be triggered for each undefined behavior configuration. The requirements for such an infrastructure were: contain benchmarks for C/C++ applications and avoid synthetic benchmarks. We wanted to avoid synthetic benchmarks because we aimed to evaluate the impact of undefined behavior optimizations on real application loads.

The investigation of the state-of-the-art benchmarking infrastructure for real application loads resulted in two candidates, i.e. Phoronix Test Suite [2] and Geekbench [1]. We chose to go further with Phoronix because it sofware is open-source and it can be extended with new benchmarks as per our needs, as opposed to Geekbench.

Phoronix offers a wide variety of benchmarks, written in various programming languages. We were only interested in the ones written in C and C++. To do that we had to filter all benchmarks by their dependencies and choose only the ones that had a C/C++ compiler as a dependency. This resulted in a list of nearly 200 applications.

We had to further filter out the benchmark applications because a number of them had problems while they were compiled with Clang. The effort of making them work with this compiler was not worth as we had a good amount of applications that already worked.

Next, we had the problem of benchmarking time. Spending too much time on benchmarking was not beneficial for us so we decided to limit one round of benchmarking to 24 hours. Because of that we had to further cut applications tht required a significant amount of time to benchmark. One example of such application is GCC. One of the benchmarks in our suite was measuring the time of building GCC form sources. GCC was a complicated 3-step build process, that takes a few hours to complete, thus we wanted to avoid it.

This process resulted in the benchmark suite presented in Table 1. At this moment the suite mostly contains applications that are either CPU bound or memory bound. We discarded completly GPU, OpenMP and MPI applications as they are more difficult to benchmark.

3 Compiler Configurations based on Undefined Behavior

After we defined the benchmark suite, we focued on evaluating the performance impact of various undefined behavior optimizations. In this section we present first the already-implemented configurations for the above mentioned optimizations and second we present the configurations that we implemented.

Each configuration is made available through a compiler flag that can be used when compiling the benchmarks.

3.1 Already-implemented compiler flags

We benefited from 5 flags in this category.

-fwrapv instructs the compiler to assume that signed arithmetic overflow of addition, subtraction and multiplication wraps around using twos-complement representation. In LLVM, this has the impact of dropping the *nsw* attribute in the above mentionted arit-metic operations.

-fno-strict-aliasing instructs the compiler to apply the strictest aliasing rules available. In LLVM, this has the impact of dropping the *tbaa* attribute that is used for type based alias analysis.

-fstrict-enums instructs the compiler to optimize using the assumption that a value of enumerated type can only be one of the values of the enumeration (as defined in the C++ standard; basically, a value that can be represented in the minimum number of bits needed to represent all the enumerators). In LLVM, this has the impact of adding the *range* attribute to memory operations.

-fno-delete-null-pointer-checks instructs the compiler to assume that programs can safely dereference NULL pointers and thus to not delete NULL pointer checks that are proved to be redundant.

-fno-finite-loops instructs the compiler to assume that no loop is finite. In LLVM, this has the impact of dropping the *mustprogress* attribute from all loops and functions.

3.2 Added compiler flags

-fconstrain-shift-value instructs the compiler to mask the right-hand-side (RHS) of the shift operation so that it does not produce undefined behavior when the RHS is bigger that the bitwidth. On x86, this add an additional *and* instruction for masking.

-fno-constrain-bool-value instructs the compiler to not constrain bool values to 0 and 1. In LLVM, this has the impact of dropping the *range* attribute from memory operations that work with booleans.

-fno-use-default-alignment instructs the compiler to use alignment 1 for all memory operations including load, store, memcpy, etc. The alignments of global variables and allocas remain unaffected. This has the impact of not generating the most efficient code because the compiler cannot find the best alignment for each operation that was forcefully aligned to 1.

-mllvm -zero-uninit-loads instructs the compiler to replace uninitialized loads with zero loads. This does not automatically initialize all memory with zero, instead it fills the memory with zero only when the memory is requested.

-fdrop-inbounds-from-gep -mllvm -trap-on-oob instructs the compiler to trap when it detects an out-of-bounds (OOB) memory access. By trapping, the compiler is blocked from doing any further optimizations based on OOB. This is a combination between a Clang flag (*-fdrop-inbounds-from-gep*) and a LLVM flag(*-trap-on-oob*). We needed the Clang flag to make sure that no optimization is triggered on the OOB access before we add the trap.

At the moment of writing this report, the last two flags are still in development.

4 Results

TODO: talk about the experiment setup

To gather relevant results, we compiled the whole benchmark suite presented in Table 1 with a single flag presented in Section 3 at a time. Furhtermore we used as baseline the benchmarks compiled with no flag presented in Section 3. After this step we used Phoronix to run the generated binaries for each benchmark.

While running a benchmark, Phoronix tries to reduce the noise as much as possible by rerunning the benchmark until the deviation between the results becomes minimal. This helped us because we did not have to do any further calibration to the results.

After compiling and running the benchmarks with all the configurations, we started to compare each configuration against the baseline. In this step we recorded the percentage of positive and negative performance impact relative to the baseline. To take into account noise in the results, we defined the noise threshold as -2% for negative performance impact and +2% for positive performance impact.

Plots presenting the performance impact for each flag are presented in Subsection 8.1 from Appendix.

For each flag, in nearly 90% of the cases, the performance impact is insignificant, i.e. can be considered noise as it is placed between -2% and +2%.

-fwrapv has the biggest overall negative peformance impact, i.e. 11% of the results for *-fwrapv* have negative performance impact. This flag also exhibits the highest negative performance impact as presented in Figure 1. Other benchmarks with negative impact: FFTW - Float + SSE - Size: 1D FFT Size 256, uvg266 (Video Encoder). Other benchmarks with positive impact: OpenSSL - RSA4096.

-fno-strict-aliasing is the most balanced flag up until this moment. 3% of the results are positive pefrormance impact and 8.2% of the are negative performance impact. The outliers for this flag are also very close to the noise thresholds.

-fconstrain-shift-value has many results in the positive impact area, i.e. 8.1%. It also has the biggest overall positive performance impact, i.e. it increases the performance with 35% relative to baseline. This outlier exhibits for the benchmark presented in Figure 2. In this benchmark, *-fconstrain-bool-value* also presents a considerable positive performance impact.

-fno-use-default-alignment exhibits 2 times more positive performance impact than negative peformance impact. This result is unexpected because in theory the behavior that this flag exhibits should decrease the peformance of memory accesses.

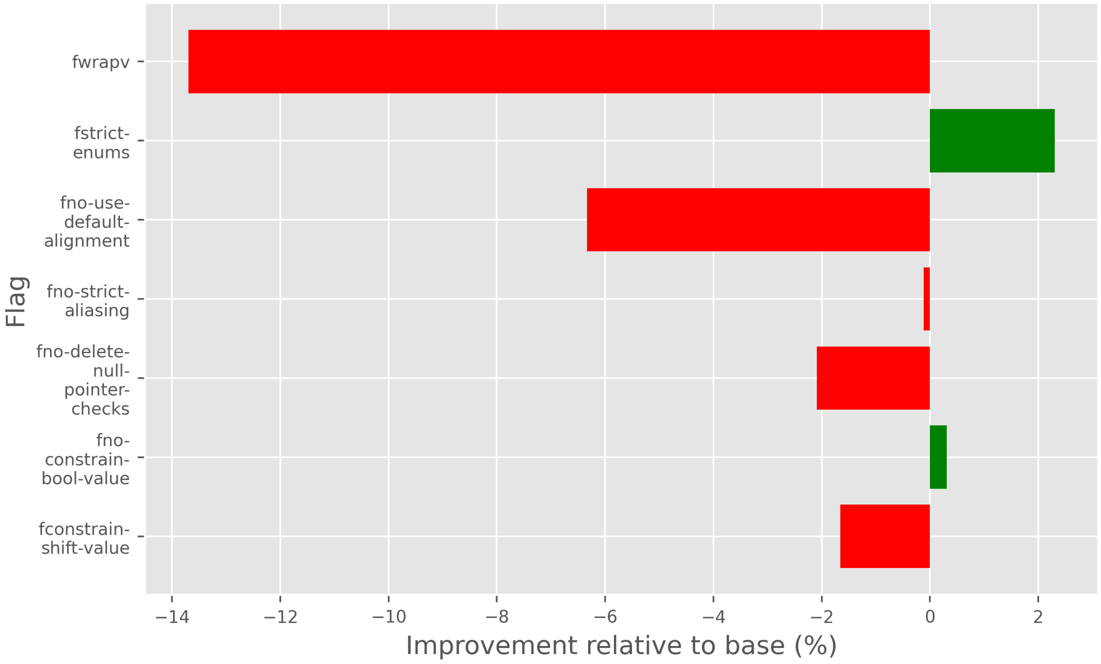

Figure 1: eSpeak-NG Speech Engine - Text-To-Speech Synthesis Benchmark, Baseline: 41.59 Seconds

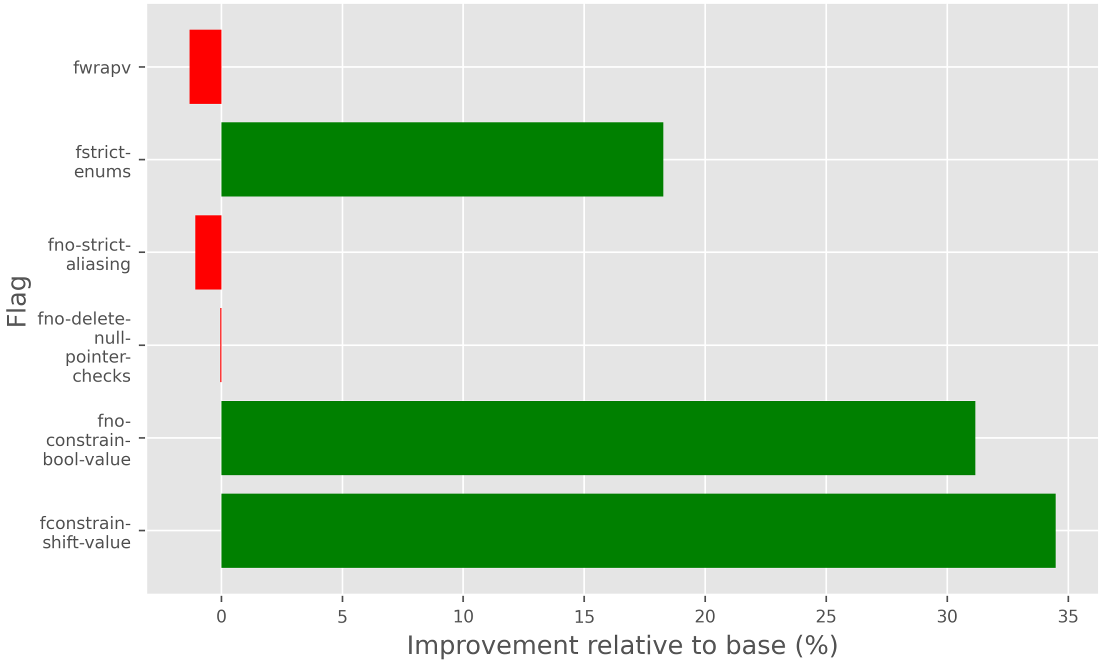

Figure 2: GtkPerf - GTK Widget: GtkDrawingArea - PixBufs, Baseline: 170.08 Seconds

5 Discussion

TODO: talk about the benchmarks that we plan to add TODO: talk about the limitations of current benchmark strategy TODO: talk about the current method of discovering UBs

6 Conclusions

In this report we started the work of evaluating the peformance impact of undefined behavior optimizations. We split the work in two parts. In the first part we focused on developing a benchmark suite that contains C/C++ applications with real loads, as opposed to synthetic loads. Then we started benchmarking 10 compiler flags that control the behavior of the compiler optimizations with regards to undefined behavior. 5 of them were already implemented, but we also added 5 new flags. Early results show that in nearly 90% of the cases the performance impact of undefined behavior in optimizations is insignificant.

7 Further Work

In the next semester we plan to explore new compiler configurations. This includes optimizations based on use-after-free, optimizations based on alias analysis that uses object-based rules or optimizatoins based on arithmetic related undefined behaviors, such as division by 0.

We also plan to run the benchmarks on other hardware architectures such as AMD or ARM to explore how they behave in comparison with the current hardware setup that is based on Intel.

It's also an interest for us to combine the flags to see how they behave together with regards to performance but must important is to analyse the current impact and discover the root causes of current peformance numbers.

References

[1] Geekbench. `https://www.geekbench.com/`, last visited Saturday 10th June, 2023.

[2] Phoronix test suite. `https://www.phoronix-test-suite.com/`, last visited Saturday 10th June, 2023.

8 Appendix

Benchmark Suite	
Application Category	Phoronix Application Identifier
LLVM Build Speed	build-llvm
HPC	fftw
Video Encoding	aom-av1 uvg266
Simulation	brl-cad
Bioinformatics	mrbayes hmmer
Image Processsing	jpegxl graphics-magick
Raytracing	tungsten
Parallel Processing	tjbench simdjson
Security	aircrack-ng openssl
Password Cracking	john-the-ripper
Database	redis
Audio Encoding	encode-flac
Texture Compression	basis draco
Compression	compress-zstd compress-pbzip2
Speech	espeak rnnoise
Software Defined Radio	liquid-dsp
GUI	gtkperf
Finance	quantlib
Telephony	pjsip
Circuit Simulator	ngspice
Webserver	apache nginx
Theorem Prover	z3

Table 1: Final benchmark suite that uses Phoronix applications

8.1 Performance Impact for Undefined Behavior Flags

TODO: Add CDF for -fno-finite-loops

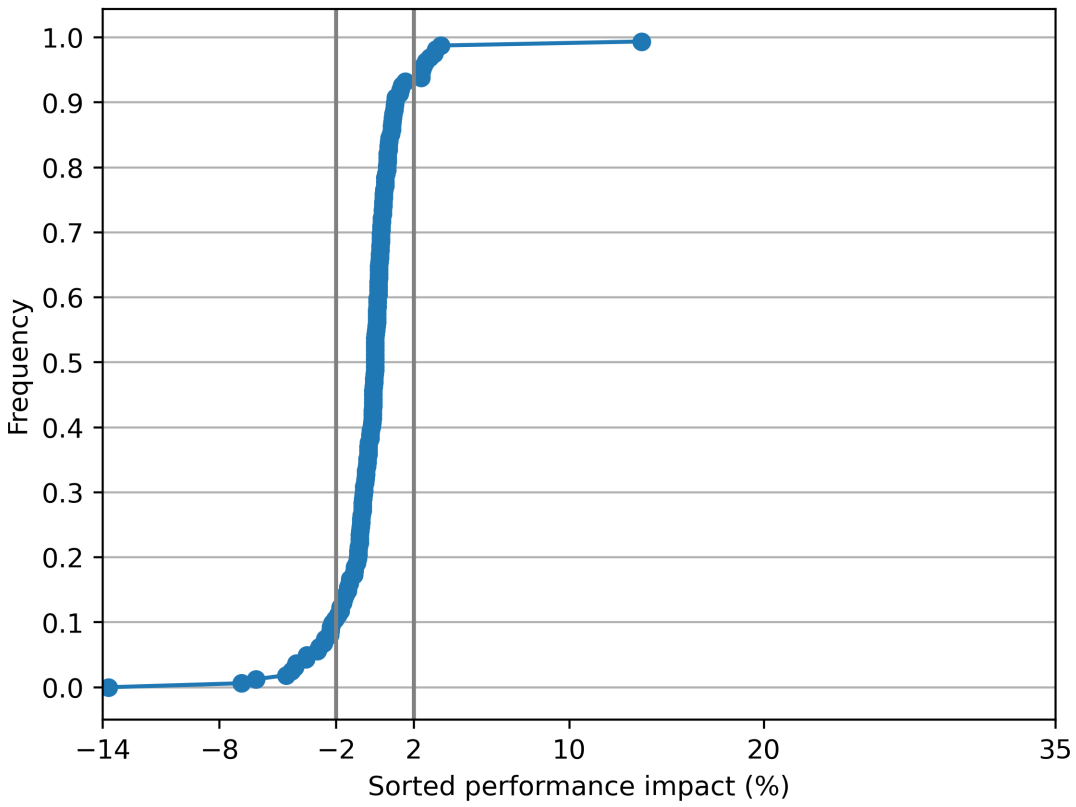

Figure 3: CDF of performance impact for -fwrapv

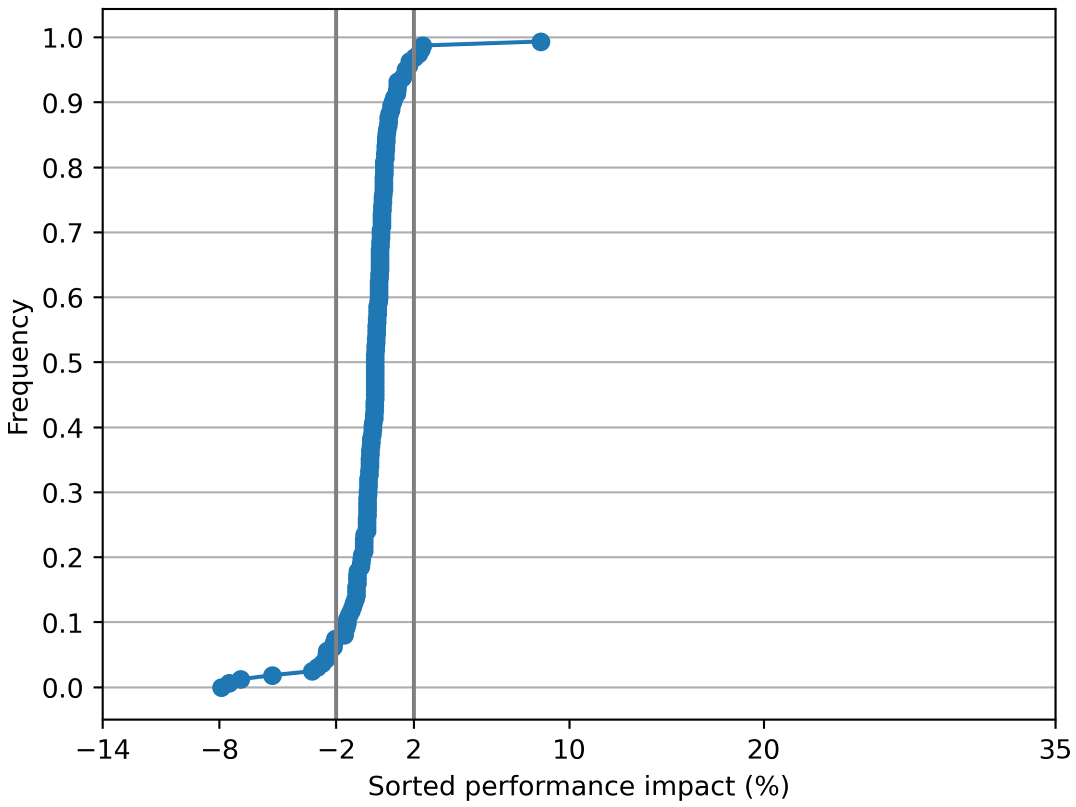

Figure 4: CDF of performance impact for -fno-strict-aliasing

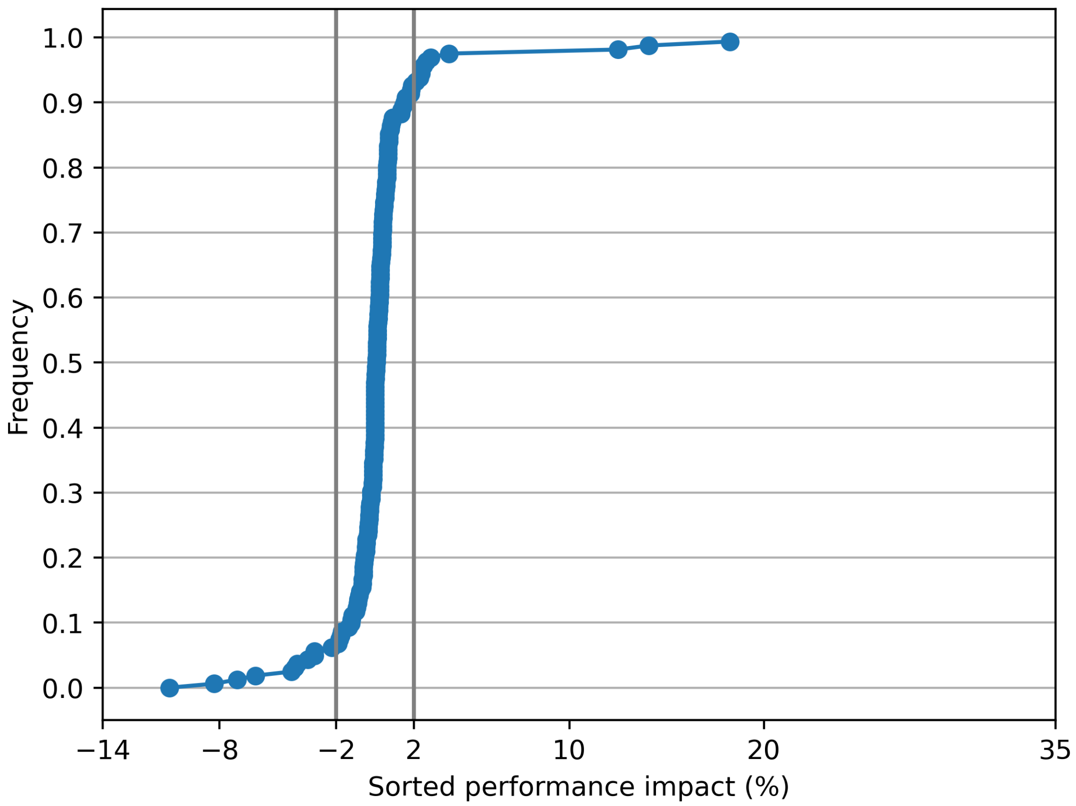

Figure 5: CDF of performance impact for -fstrict-enums

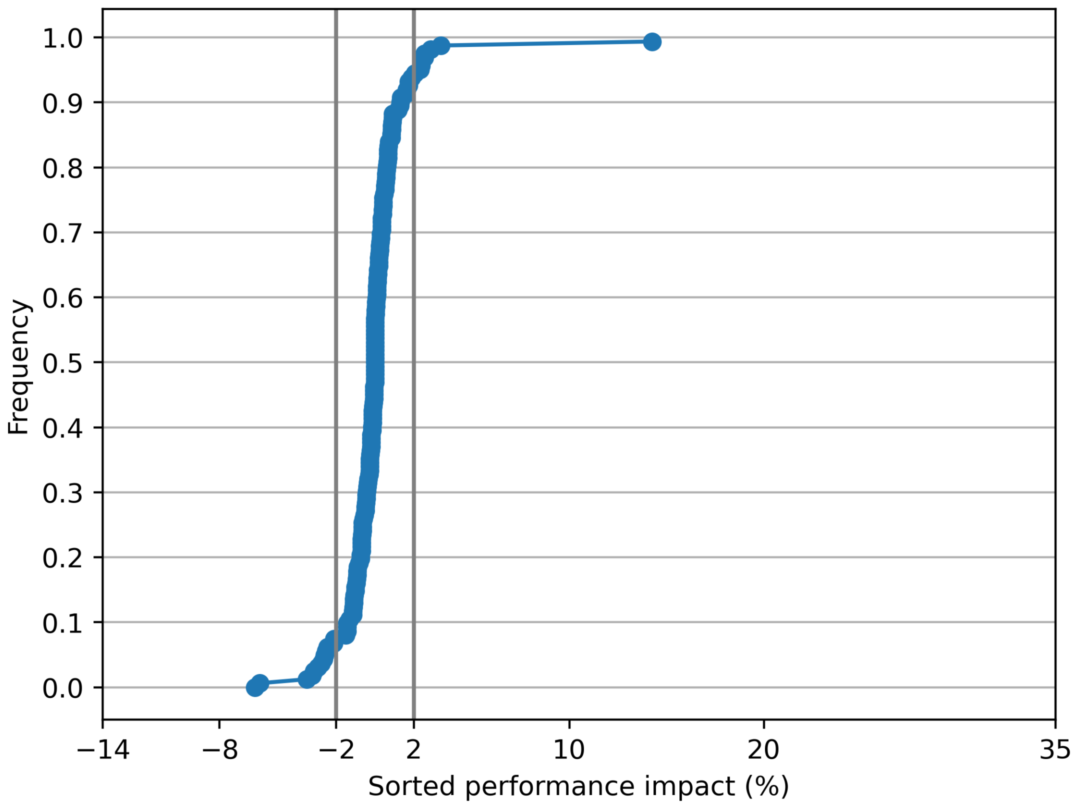

Figure 6: CDF of performance impact for -fno-delete-null-pointer-checks

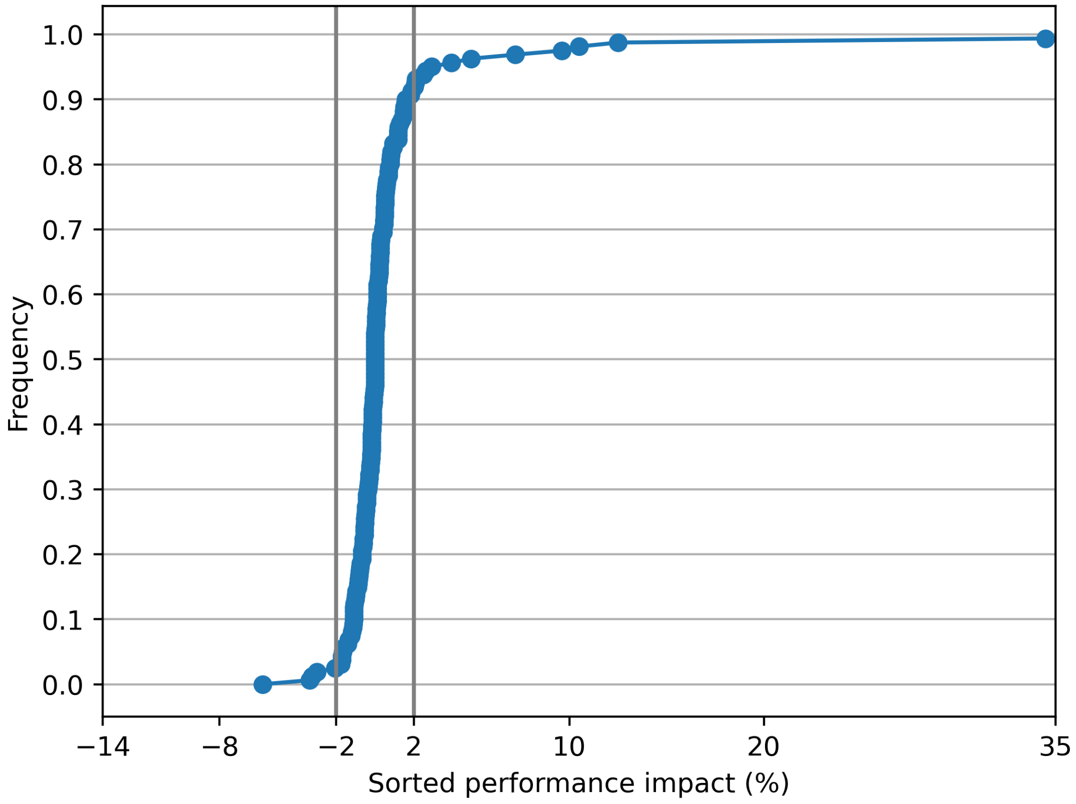

Figure 7: CDF of performance impact for -fconstrain-shift-value

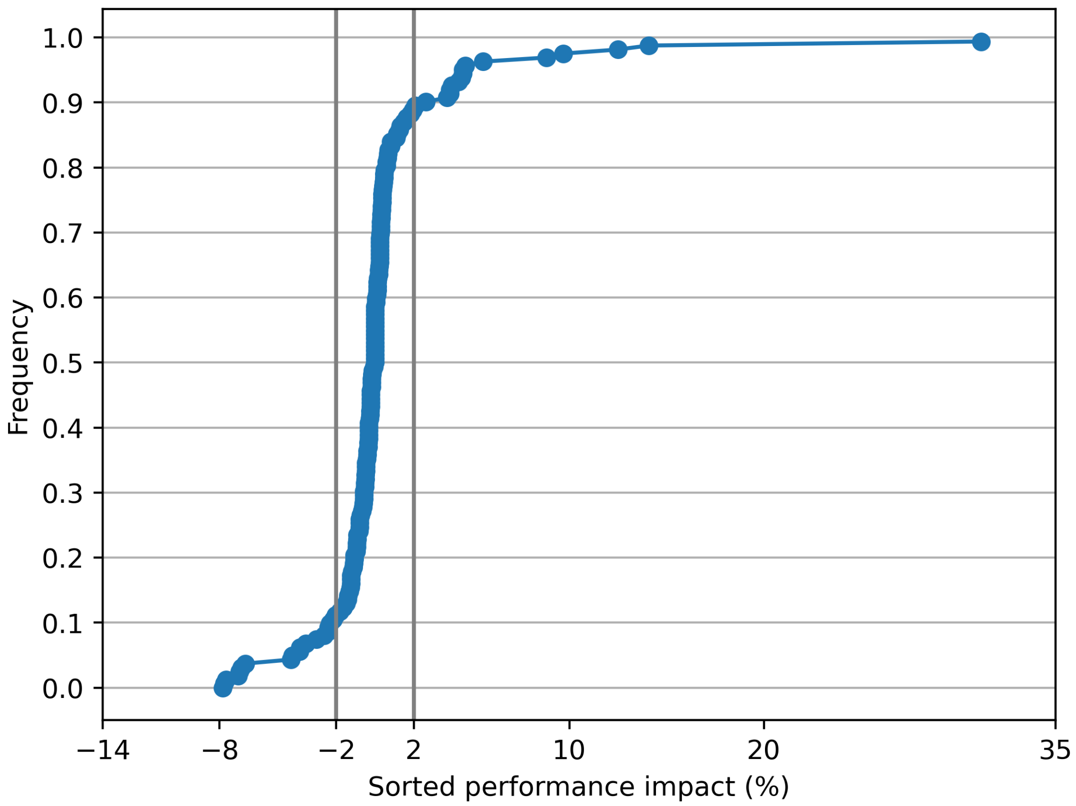

Figure 8: CDF of performance impact for -fno-constrain-bool-value

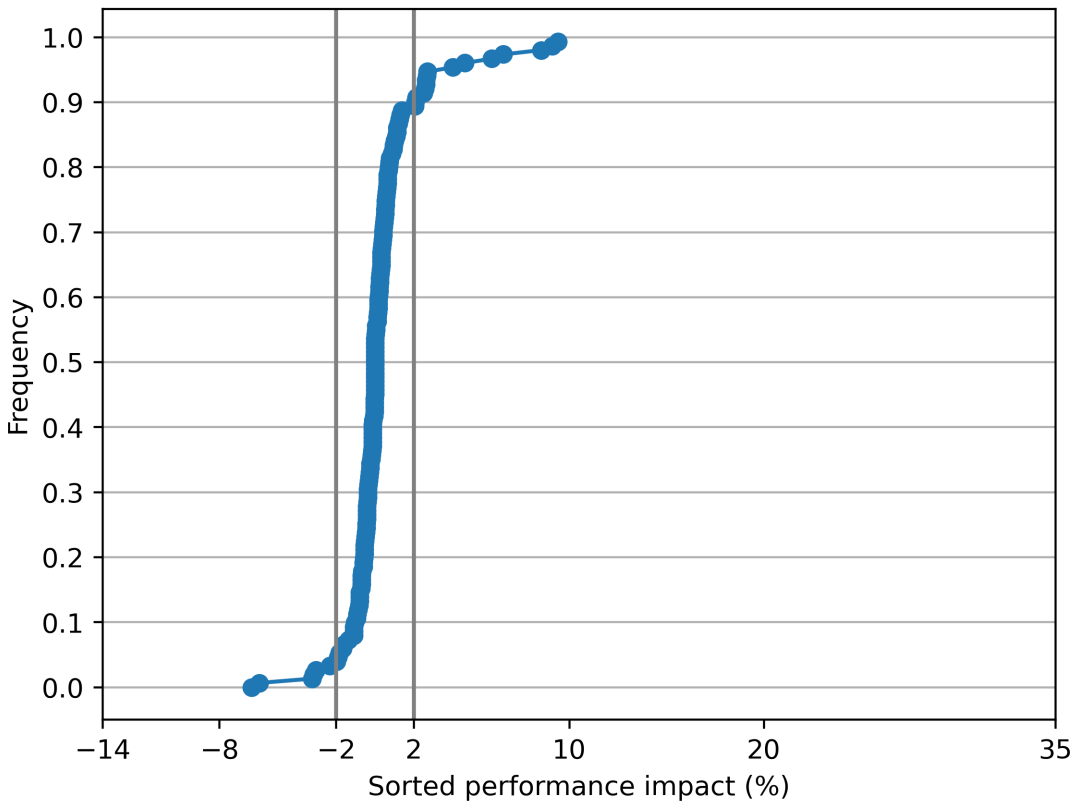

Figure 9: CDF of performance impact for -fno-use-default-alignment