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*

Friday 2nd February, 2024

Abstract

Clang/LLVM uses undefined behavior to issue optimizations. We analyze the impact of this class of optimizations on a benchmarking suite built using Phoronix. Before we ran the performance experiments, we used the suite and Alive2 to discover additional cases where the compiler exploits undefined behavior. We found six cases that we integrated in our benchmarking infrastructure. Then we carried on with running the benchmarks and stabilize their results on both x86 and arm. Finally, we started the analysis of the performance regressions on x86.

1 Introduction

Our work is focused on evaluating the performance impact of undefined behavior optimizations. Traditionally, undefined behavior was used as a free ticket to issue various types of optimizations targeting either code speed or code size. The impact of this type of optimization is already well known for micro-benchmarks, however, there is no study covering the impact for real loads. We fill this gap by benchmarking various instances of undefined behavior optimizations in Clang/LLVM on real application loads written in C/C++.

In the past semester we compiled a benchmarking suite 1 using Phoronix Test Suite that focuses on various types of applications ranging from compression algorithms to web servers. Besides that, we gathered and created various configurations, controlled by compiler flags, that exploit undefined behavior in the optimizations pipeline of LLVM.

In this semester we continued the work on developing new compiler flags for controlling undefined behavior exploitation. Some of them were mentioned in the Further Work section of the past report, i.e. flags for optimizations based on use-after free, optimizations based on alias analysis that uses object-based rules and optimization based on arithmetic related undefined behaviors, such as division by 0. However, we also used Alive2 [14], an

automatic verification tool for LLVM optimizations, to find new flags for our benchmarking infrastructure. We found six more flags that are mainly concerned with various LLVM attributes that can cause undefined behavior, for example *noalias*, *align* or *dereferenceable*. In total we have 18 flags 2 for controlling undefined behavior exploitation.

After finding all possible sources of undefined behavior, we carried on with running the performance experiments. For this we compiled each benchmark in the suite of 25 benchmarks with the following configurations:

- baseline, i.e. without any undefined behavior flag
- one undefined behavior flag at a time
- all undefined behavior flags combined

This results in 20 builds for each benchmark that were run on the benchmarking servers, i.e. an Intel Xeon E5-2680 v2 @ 3.60GHz and an arm Neoverse-N1 @ 3.00 GHz. To make sure that we get deterministic results for the benchmarks we issued various techniques for stabilizing our builds, they include pinning the processes to only one NUMA node and dropping the frequency of the processor.

After we gathered the first batch of results, we started analyzing them to understand what parts of LLVM caused the performance regressions and how can we recover the performance. Current results indicate losses of up to 15% that need careful investigation. At this moment we are investigating a performance loss in simdjson caused by the absence of the *align* attribute on a pointer argument from a performance critical function.

2 Finding new exploitations of undefined behavior

We explored various flags that deal with the exploitation of undefined behavior. Some of them were already integrated in LLVM, e.g. -fwrapv, and some of them were developed by us based on our previous experience and based on LangRef [8]. However, there is a third category of flags that we developed based on automatic exploration using Alive2.

Alive2 [14] is an automatic verification tool for LLVM optimizations. Roughly, after each optimization pass it checks whether the Intermediate Representation (IR) contains undefinedness. This helps us find places in the optimization pipeline that exploit undefined behavior.

We decided to patch Clang, the frontend for C/C++, whenever we found an undefined behavior exploitation in the optimization pipeline of LLVM. By making sure that we generate code free of undefined behavior directly from the source, i.e. Clang, we made sure that further optimization cannot take any advantage or further propagate the exploitation of undefined behavior. For example, instead of patching each place in LLVM that takes advantage by division by 0, we generated code from Clang that guarded each division so that LLVM cannot further optimize the division.

The following is an example of undefined behavior detected using Alive2. Listing 1 contains a C function from the aom-av1 [1] benchmark that returns a boolean value based on a pointer received as parameter. Listing 2 contains the corresponding Alive2 output for is_sb_aq_enabled. The output can be broken down in multiple parts:

1. The IR before the optimization and the IR before the optimization, they are delimited by =>
2. Information about the validity of the transformation, in this case we get an "Transformation doesn't verify! (unsound) ERROR: Source has guardable UB"
3. Detailed information about the cause of the error including values that triggered the error and the memory state

```c
static bool is_sb_aq_enabled(const AV1_COMP *const cpi) {
  return cpi->rc.sb64_target_rate >= 256;
}
```

Listing 1: C code for is_sb_aq_enabled

Listing 2: Alive2 output on is_sb_aq_enabled

```
1.
define i1 @is_sb_aq_enabled(ptr noundef %cpi) null_pointer_is_valid zeroext {
%entry:
  %cpi.addr = alloca i64 8, align 8
  store ptr noundef %cpi, ptr %cpi.addr, align 1
  %0 = load ptr, ptr %cpi.addr, align 1
  %rc = gep inbounds ptr %0, 677024 x i32 0, 1 x i64 427536
  %sb64_target_rate = gep inbounds ptr %rc, 248 x i32 0, 1 x i64 16
  %1 = load i32, ptr %sb64_target_rate, align 1
  %cmp = icmp sge i32 %1, 256
  ret i1 %cmp
}
=>
define i1 @is_sb_aq_enabled(ptr noundef %cpi) null_pointer_is_valid zeroext {
%entry:
  %rc = gep inbounds ptr noundef %cpi, 677024 x i32 0, 1 x i64 427536
  %sb64_target_rate = gep inbounds ptr %rc, 248 x i32 0, 1 x i64 16
  %0 = load i32, ptr %sb64_target_rate, align 1
  %cmp = icmp sge i32 %0, 256
  ret i1 %cmp
}

2.
Transformation doesn't verify! (unsound)
ERROR: Source has guardable UB

3.
Example:
ptr noundef %cpi = pointer(non-local, block_id=0, offset=-388623)

Source:
ptr %cpi.addr = null
ptr %0 = pointer(non-local, block_id=0, offset=-388623)
ptr %rc = poison
ptr %sb64_target_rate = poison
i32 %1 = UB triggered!

SOURCE MEMORY STATE
===================
NON-LOCAL BLOCKS:
Block 0 >       size: 601368    align: 9223372036854775808      alloc type: 0   address↩
    : 0
Block 1 >       size: 8968      align: 4        alloc type: 0   address: 601368

LOCAL BLOCKS:
Block 2 >       size: 601368    align: 9223372036854775808      alloc type: 0   address↩
    : 0

Target:
ptr %rc = poison
ptr %sb64_target_rate = poison
i32 %0 = UB triggered!
```

One can spot the poison values in %rc and %sb64_target_rate from the detailed information section of the output. They cause the subsequent "UB triggered!" because as per LangRef [12] when using a load instruction, "if pointer is not a well-defined value, the behavior is undefined.". To stop this exploitation of undefined behavior from taking place, we removed the *inbounds* keyboard from the GEP instruction directly from Clang. By removing *inbounds* we made sure that the GEP instructions where %rc and %sb64_target_rate are created will not generate undefinedness as none of the cases described in [11] is triggered, hence no undefined behavior is triggered.

We integrated the functionality of removing *inbounds* from GEP instructions in a flag called -fdrop-inbounds-from-gep. It only takes effect in Clang when different GEP instructions are generated. Using this method described above, we created 5 more flags:

- -fdrop-noalias-restrict-attr: Drop noalias attribute generated by the restrict keyword
- -fdrop-align-attr: Drop align attribute
- -fdrop-deref-attr: Drop dereferenceable attribute
- -fdrop-ub-builtins: Drops builtin_assume_aligned and replaces builtin_unreachable with builtin_trap
- -fdrop-pure-const-attr: Drop pure and const function attributes

We also had plans to drop the noundef attribute but fortunately there already existed a flag for that, callled "-Xclang -no-enable-noundef-analysis".

In the process of running Alive2 over the benchmarking suite, we found a couple of bugs in LLVM and bugs in Alive2. Below are two examples of such bugs.

When running Alive2 over aom-av1 we found one bug in LLVM related to loop vectorization [9], the LoopVectorizer was adding an misaligned store because it was not respecting the alignment of the original store instruction.

On BRL-CAD and OpenSSL we found a bug in Alive2 because it cannot analyse locally-allocated pointers on function calls. Listing 3 is a snippet from OpenSSL that showcases the problem. Because abuf and bbuf are pointers allocated inside xname_cmp, Alive2 cannot continue the analysis of the pointers inside i2d_X509_NAME. This is not a fundamental problem in Alive2, the functionality is not currently implemented.

```
1
2  static int xname_cmp(const X509_NAME *a, const X509_NAME *b)
3  {
4      unsigned char *abuf = NULL, *bbuf = NULL;
5
6      ...
7
8      // Alive2 cannot propagate the locally allocated pointers in ↩
       i2d_X509_NAME
9      alen = i2d_X509_NAME((X509_NAME *)a, &abuf);
10     blen = i2d_X509_NAME((X509_NAME *)b, &bbuf);
11
12     ...
13 }
```

Listing 3: OpenSSL code that was causing Alive2 bug

3 Running the benchmarks

After developing all the flags that control the behavior of the compiler with regards to undefined behavior exploitation, we started the performance experiments. There is a total of 20 configurations that we used:

- baseline, i.e. without any undefined behavior flag
- one undefined behavior flag at a time
- all undefined behavior flags combined

Considering that we have 25 benchmarks, each with a variable number of tests that can finish in a matter of minutes or in a matter of hours, the whole benchmarking process took us approximately two weeks for each of our servers, i.e. an Intel Xeon E5-2680 v2 @ 3.60GHz and an arm Neoverse-N1 @ 3.00 GHz.

Before analyzing the results we had to make sure that the benchmarking environment is stable so that our results do not show a high degree of variance. In the past semester we ran a couple of experiments on the Intel server and we have seen a satisfactory level of variance, e.g. 2%. However, this semester we also acquired an arm server on which we ran the experiments and which showed levels of variance way above the satisfactory threshold. This was a major problem as no significant conclusion could be drawn from benchmarks with this level of instability.

Figure 6 shows a case of variance in the aom-av1 benchmark. While configurations such as -fconstrain-shift-value, -fdrop-align-attr, -fdrop-deref-attr have an insignificant variance, configurations such as "-fdrop inbounds-from-gep -mllvm -disable-oob-analysis" or -fignore-pure-const-attrs show variances between the min value and the max value of 62% or 106%.

To solve the variance issues we found the following two methods to be effective:

- Use numactl to pin the benchmarking process to a specific NUMA node
- Reduce the processor frequency

Figure 1 shows the advantages of these two techniques on the baseline configuration. Running at 3.00GHz without numactl exposes outliers that are not found in the versions with reduced frequency and numactl enabled. Even though for this specific case the variance at 1.50GHz and at 2.00GHz seems acceptable, there are other tests in aom-av1 that do not show satisfactory variance for 1.50GHz and 2.00GHz. Because of this we chose to decrease the frequency to 1.00GHz.

The question on why the benchmarks are more unstable on arm than on x86 still persists. We have a couple of theories on why this might happen but we need to do more experiments in order to validate them. They include: CPU frequency instability caused by the thermal controllers or unstable memory access times caused by the memory access patterns displayed by the benchmarks.

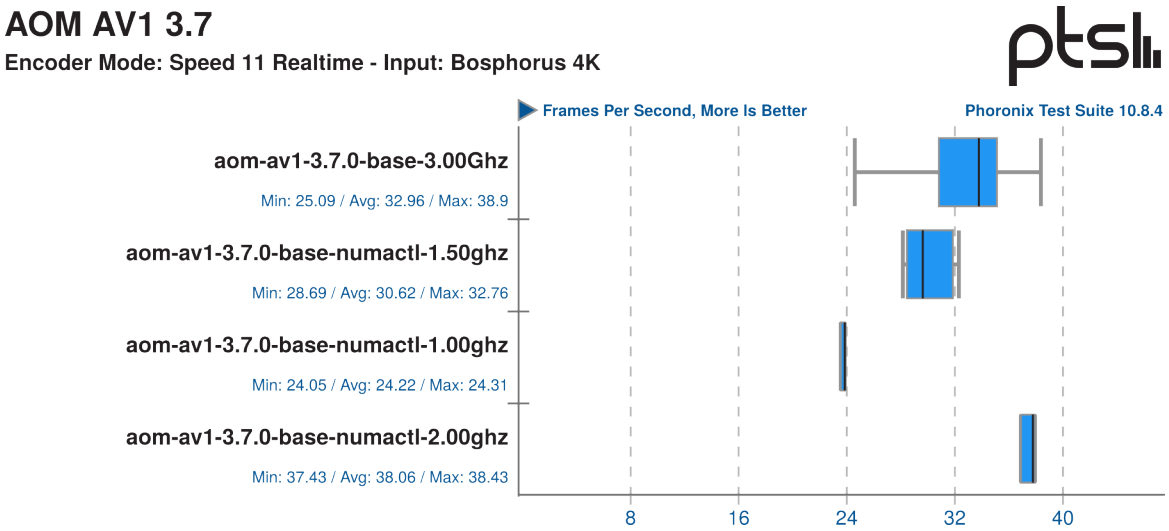

Figure 1: Improvement of stability using numactl and frequency dropping

After solving the instability problems, we continued to gather and analyse the results on both Intel and arm.

On one hand, Figure 2 and Figure 3 present the quantitative impact of each flag over the benchmark suite, i.e. how each flag affects the benchmarks. We define affected benchmarks as benchmarks for which there exists a performance improvement over 2% or a performance decrease over 2%.

In these figures we expand the definition of a benchmark. In the beginning of this section we stated that our suite consists of 25 benchmarks. However, these 25 benchmarks can be considered as 25 separate application that, in turn, contain a variable number of sub-benchmarks, or tests. This expansion results in over 140 benchmarks.

We expected the flags to affect differently the benchmarks on the two hardware architectures since they use different operating system configurations and have different hardware characteristics. The cumulative effect of the flags in bigger on arm than on x86. Plus, for each individual flag, arm has a bigger impact on the benchmark than x86. We plan to answer in the following semester what is the exact cause of these differences.

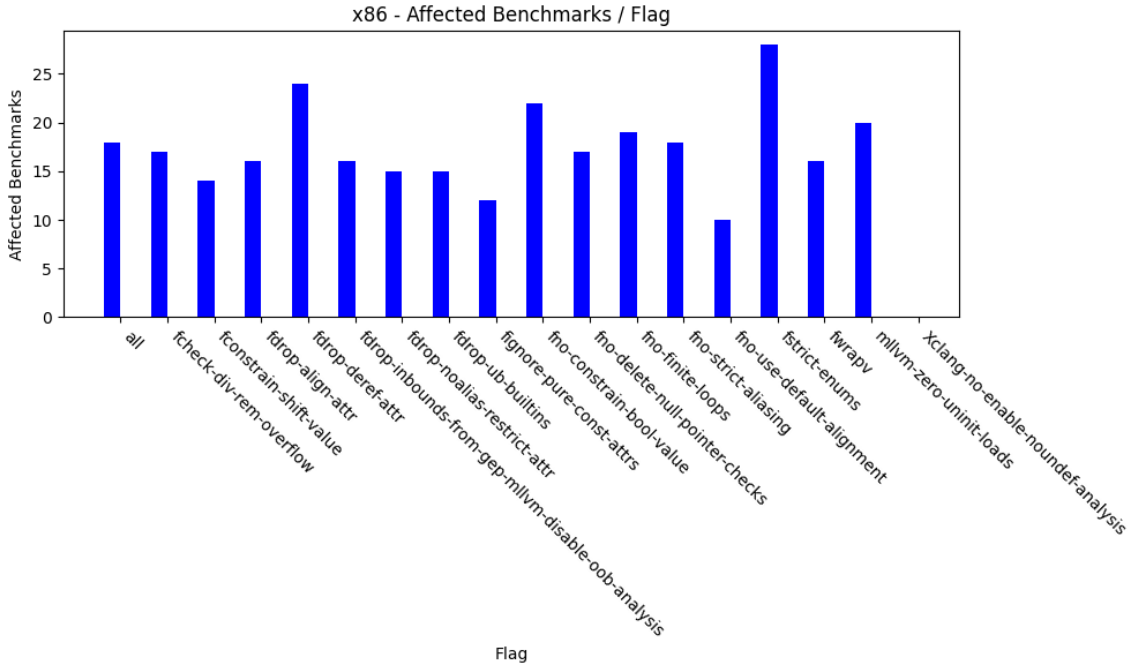

Figure 2: Number of affected benchmarks by each flag on x86 for -O2

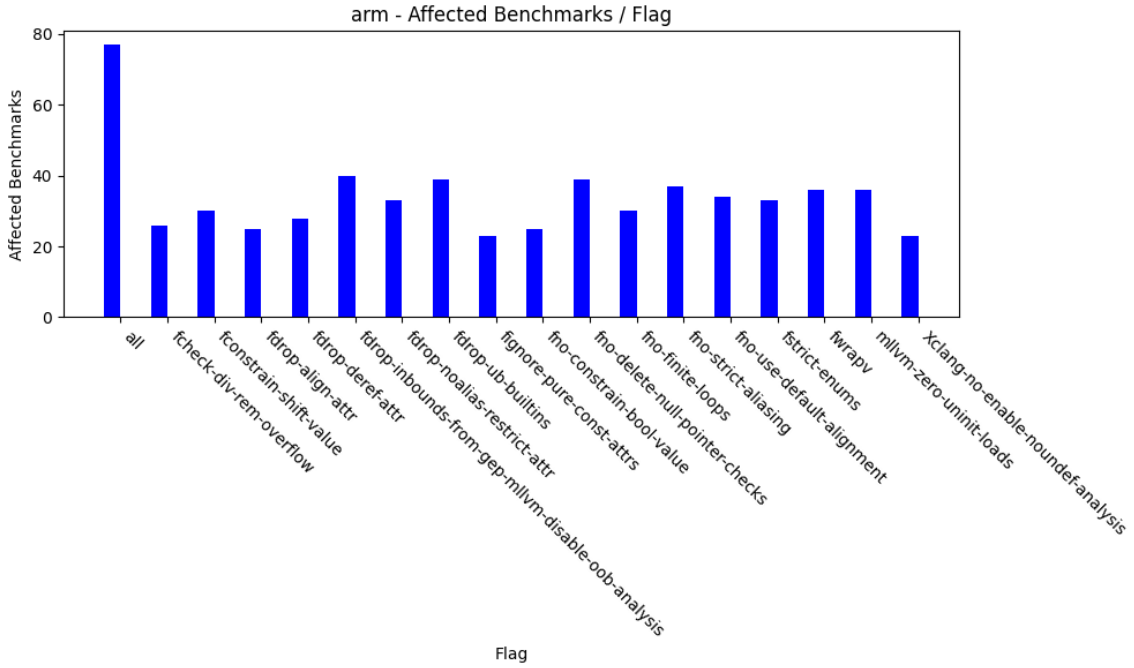

Figure 3: Number of affected benchmarks by each flag on arm of -O2

On the other hand, Figure 4 and Figure 5 present the qualitative impact of each flag over the benchmark suite. No flag exhibits overall negative impact or overall positive impact. Each benchmark is affected differently by the same flag. This happens because each benchmarked application contains different resource utilization patterns.

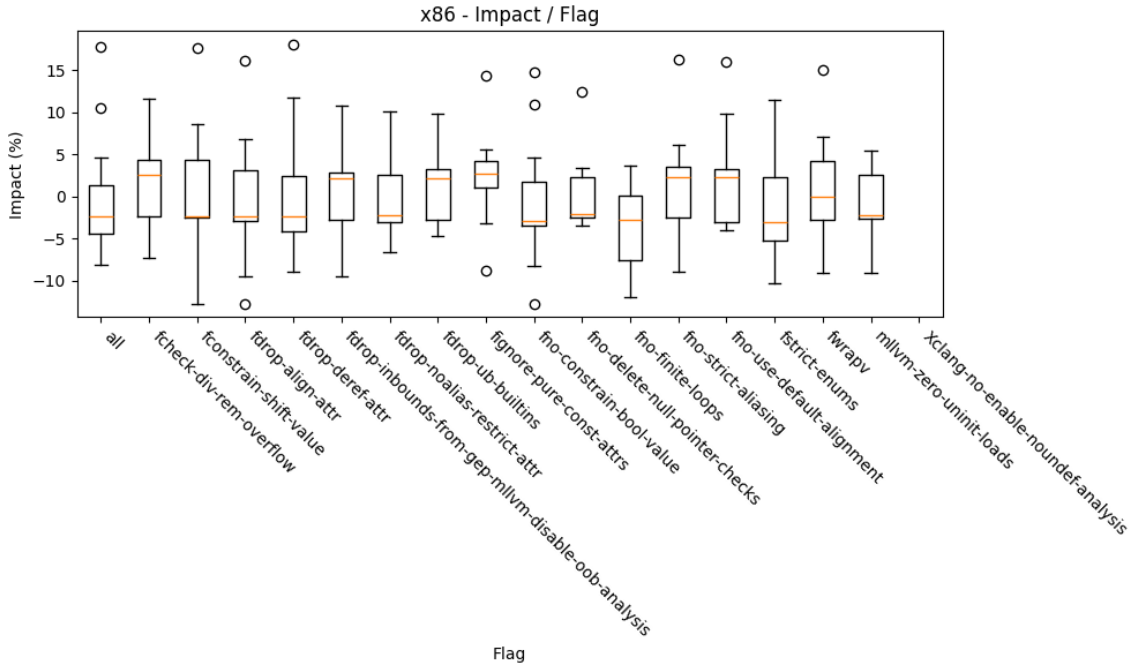

Figure 4: Impact of each flag on the suite on x86 for -O2

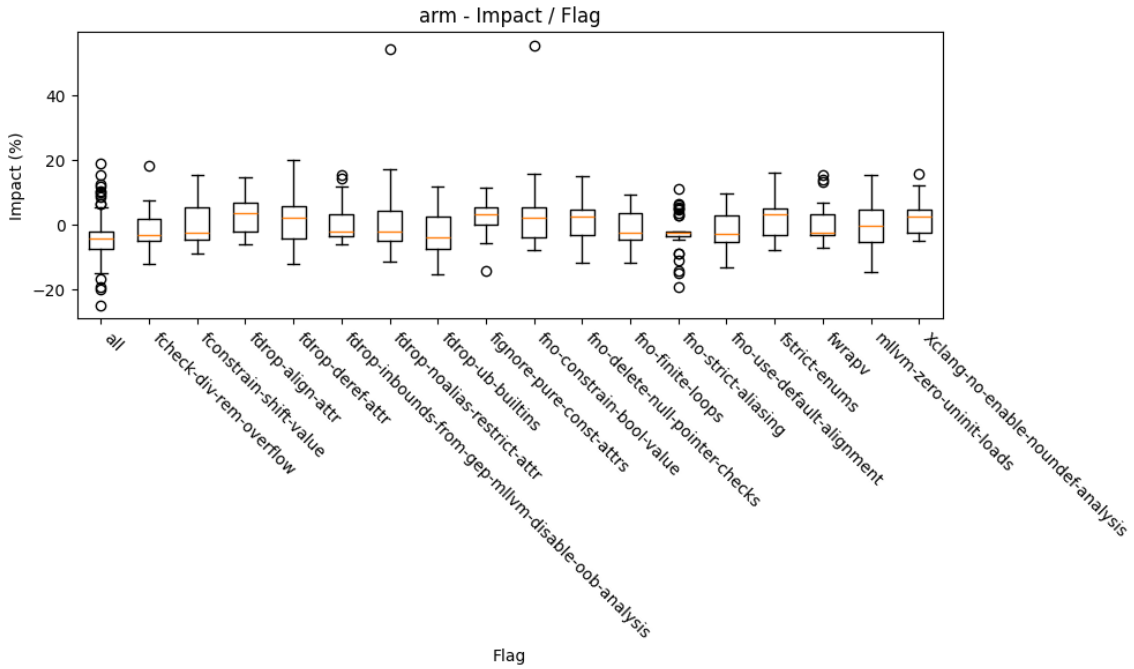

Figure 5: Impact of each flag on the suite on arm for -O2

Currently we have the -O2 numbers for both Intel and arm, we plan to continue to gather the numbers for -O3, -Os and -Oz.

While we are gathering the numbers for the rest of optimization levels, we started the analysis of the performance regressions for -O2 on x86. The next section talk about the analysis process and current findings.

4 Analysis of performance regressions

Currently we are analysing a performance regression in the simdjson-2.0.1 [13] benchmark on Intel. For the -fdrop-align-attr there is a loss of 12.3% performance compared to baseline. In terms of raw performance, this underlies a throughput drop from 2 GB/s to 1.77 GB/s.

The purpose of -fdrop-align-attr is to drop every *align* attribute for each argument of every function. By doing this we make sure that no undefined behavior related to alignment is generated [10]. However as seen in the above percentage, a good amount of performance is lost by using this approach.

To find the functions that triggered the regression we used elf_diff [2], a tool for comparing ELF binaries in a visual way as well as in a programmatic way. Using elf_diff we were able to get an idea of the changes caused by dropping the *align* attribute. Then we gathered all the affected functions and started a binary search to reduce to a minimum the set of affected functions that also caused the performance regression.

After finding the problematic function, Listing 4, we started to analyze how each argument of this function is affected by the *align* attribute, Listing 5

```
1
2 partial_tweets::simdjson_ondemand::run(simdjson::padded_string& this, std↩
    ::vector<partial_tweets::tweet<std::basic_string_view<char, std::↩
    char_traits<char>>> json, std::allocator<partial_tweets::tweet<std::↩
    basic_string_view<char, std::char_traits<char>>>>& result)
```

Listing 4: Function that caused performance regressions

```
1
2 ptr noundef nonnull align 8 dereferenceable(48) %this, ptr noundef nonnull↩
     align 8 dereferenceable(16) %json, ptr noundef nonnull align 8 ↩
    dereferenceable(24) %result
```

Listing 5: LLVM representation of arguments of partial_tweets::simdjson_ondemand::run

Even if every argument has an *align* attribute, we found that that only by dropping *align* from the *this* pointer, i.e. first parameter, and keeping the attributes of the other parameters intact, the performance drops.

To find what optimization pass was causing the regression, we used the -mllvm -print-after-all option with Clang and diffed the output with and without *align* enabled on the *this* pointer, Listing 6. We found that the Loop Invariant Code Motion (LICM) pass was generating suboptimal code that fed to subsequent passes. More specifically, because alignment information was missing, LICM was not able to hoist instructions dependent on the *this* pointer out of the main loop which caused the generated code to be more expensive in terms of instructions per loop iteration.

Listing 6: Diff between the code generated by LICM with and without the *align* attribute on the *this* pointer

```
*** IR Dump After LICMPass on while.body.i.i.i.i.i ***

 ; Preheader:
```

```
 while.body.i.lr.ph.i.i.i.i:                       ; preds = %if↩
    .end.i.i.i.i.i, %if.end.while.body.i.lr.ph.i_crit_edge.i.i.i
   %.ph.i.i.i.i = phi ptr [ %incdec.ptr.i.i.i.i.i.i.i.i.i, %if.end↩
      .i.i.i.i.i ], [ %add.ptr.i.i.i.i.i.i.i, %if.end.while.body.i↩
      .lr.ph.i_crit_edge.i.i.i ]
   %_depth.i.i78.i35.i.i.i.i = getelementptr inbounds %"class.↩
      simdjson::fallback::ondemand::json_iterator", ptr %doc, ↩
      i64 0, i32 4
   %parser.i.i.i132.i38.i.i.i.i = getelementptr inbounds %"class↩
      .simdjson::fallback::ondemand::json_iterator", ptr %doc, ↩
      i64 0, i32 1
   %error.i.i.i172.i39.i.i.i.i = getelementptr inbounds %"class.↩
      simdjson::fallback::ondemand::json_iterator", ptr %doc, ↩
      i64 0, i32 3
   %implementation.i.i.i133.i.i.i.i.i = getelementptr inbounds ↩
      %"class.simdjson::fallback::ondemand::parser", ptr %this, ↩
      i64 0, i32 1
-  %39 = load ptr, ptr %implementation.i.i.i133.i.i.i.i.i, align↩
    8
-  %n_structural_indexes2.i67.i.i134.i.i.i.i.i = getelementptr ↩
   inbounds %"class.simdjson::internal::↩
   dom_parser_implementation", ptr %39, i64 0, i32 1
-  %structural_indexes.i68.i.i135.i.i.i.i.i = getelementptr ↩
   inbounds %"class.simdjson::internal::↩
   dom_parser_implementation", ptr %39, i64 0, i32 2
   br label %while.body.i.i.i.i.i

 ; Loop:
 while.body.i.i.i.i.i:                             ; preds = %↩
    cleanup.cont.i.i.i9.i.i, %while.body.i.lr.ph.i.i.i.i

...

 sw.epilog.i.i139.i.i.i.i.i:                       ; preds = %if↩
    .then13.i.i127.i.i.i.i.i, %if.end.i.i118.i.i.i.i.i, %if.end.↩
    i.i118.i.i.i.i.i, %if.end.i.i118.i.i.i.i.i, %if.end.i.i118.i↩
    .i.i.i.i
+  %50 = load ptr, ptr %implementation.i.i.i133.i.i.i.i.i, align↩
    8, !tbaa !35, !noalias !77
+  %n_structural_indexes2.i67.i.i134.i.i.i.i.i = getelementptr ↩
   inbounds %"class.simdjson::internal::↩
   dom_parser_implementation", ptr %50, i64 0, i32 1
   %51 = load i32, ptr %n_structural_indexes2.i67.i.i134.i.i.i.i↩
      .i, align 8, !tbaa !67, !noalias !77
+  %structural_indexes.i68.i.i135.i.i.i.i.i = getelementptr ↩
   inbounds %"class.simdjson::internal::↩
   dom_parser_implementation", ptr %50, i64 0, i32 2
   %conv.i69.i.i136.i.i.i.i.i = zext i32 %51 to i64
   %52 = load ptr, ptr %structural_indexes.i68.i.i135.i.i.i.i.i,↩
       align 8, !tbaa !35, !noalias !77
   %arrayidx.i.i5570.i.i137.i.i.i.i.i = getelementptr inbounds ↩
```

```
      i32, ptr %52, i64 %conv.i69.i.i136.i.i.i.i.i
```

The instructions could not be hoisted because loads can only be hoisted if the pointer is known to be dereferenceable and the pointer is at least as aligned as the load. Since the *this* pointer has alignment 1 and the load has alignment 8, this condition is not met hence the load cannot be hoisted outside the loop. This requirement exists because not all hardware architectures support unaligned load, e.g. SPARC. This imposes a limitation on architectures such as x86 or arm that do support unaligned loads.

In LLVM, this is translated to the following sequence of calls. hoist [3] cannot be called because isSafeToExecuteUnconditionally [6] fails on the isSafeToSpeculativelyExecute [7] condition. Going further through the call chain isDereferenceableAndAlignedPointer [5] will fail because isAligned [4] returns false as BA, i.e. the alignment of the pointer, is 1 and Alignment, i.e. the alignment of the load instruction, is 8.

We continue to investigate methods of restoring the original performance by analysing how the *this* pointer was created, i.e. through a *new* call or through a memory read, so that we manage to save the alignment information and not depend on the argument attribute.

5 Conclusions and Further work

In this semester we made significant progress on finding the impact of undefined behavior optimizations. We continued the work from the past semester on developing and exploring new flags that control the behavior of the compiler with regards to exploitation of undefined behavior. We used Alive2 to find new flags. We managed to find six more flags, resulting in 18 final flags that we use in our benchmarking infrastructure.

We also started the performance experiments and solved issues regarding the stability of the results. Use used techniques such as NUMA pinning and CPU frequency lowering. This allowed us to have a deterministic and stable environment for our benchmarks.

Finally, we also started analysing the performance regressions and already found hints of where the problem might lie in LLVM. However, we need to further analyze the issues in order to provide fixes that can recover the initial performance.

In the next semester we plan to continue on analysing the performance regression with the data that is available now. We also plan to run the benchmarks with different optimization settings, such as -O3, -Os, -Oz. Finally, we plan to run the benchmarks on AMD to see how it compares to current data from Intel and arm.

References

[1] AOM-AV1 benchmark. `https://openbenchmarking.org/innhold/8fd7550f18276184e5b02576b69dff91e5ccda48`, last visited Friday 2nd February, 2024.

[2] elf_diff. `https://github.com/noseglasses/elf_diff`, last visited Friday 2nd February, 2024.

[3] hoist call in LLVM. https://github.com/llvm/llvm-project/blob/7cbf1a2591520c2491aa35339f227775f4d3adf6/llvm/lib/Transforms/Scalar/LICM.cpp#L919, last visited Friday 2nd February, 2024.

[4] isaligned call in LLVM. https://github.com/llvm/llvm-project/blob/7cbf1a2591520c2491aa35339f227775f4d3adf6/llvm/lib/Analysis/Loads.cpp#L109, last visited Friday 2nd February, 2024.

[5] isdereferenceableandalignedpointer call in LLVM. https://github.com/llvm/llvm-project/blob/72fd10adcbf8194a08141e38a95e11f4f1a8d7c2/llvm/lib/Analysis/ValueTracking.cpp#L6371, last visited Friday 2nd February, 2024.

[6] issafetoexecuteunconditionally call in LLVM. https://github.com/llvm/llvm-project/blob/7cbf1a2591520c2491aa35339f227775f4d3adf6/llvm/lib/Transforms/Scalar/LICM.cpp#L915, last visited Friday 2nd February, 2024.

[7] issafetospeculativelyexecute call in LLVM. https://github.com/llvm/llvm-project/blob/7cbf1a2591520c2491aa35339f227775f4d3adf6/llvm/lib/Transforms/Scalar/LICM.cpp#L1759, last visited Friday 2nd February, 2024.

[8] LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html, last visited Friday 2nd February, 2024.

[9] Loop vectorization bug found in LLVM using alive2. https://github.com/llvm/llvm-project/issues/65212, last visited Friday 2nd February, 2024.

[10] Semantics of align attribute in LangRef. https://llvm.org/docs/LangRef.html#parameter-attributes, last visited Friday 2nd February, 2024.

[11] Semantics of getelementptr in LangRef. https://llvm.org/docs/LangRef.html#id235, last visited Friday 2nd February, 2024.

[12] Semantics of load in LangRef. https://llvm.org/docs/LangRef.html#id209, last visited Friday 2nd February, 2024.

[13] simdjson benchmark. https://openbenchmarking.org/innhold/3cc784e9dea9f4b1638588a826339fb0d58b08f3, last visited Friday 2nd February, 2024.

[14] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

6 Appendix

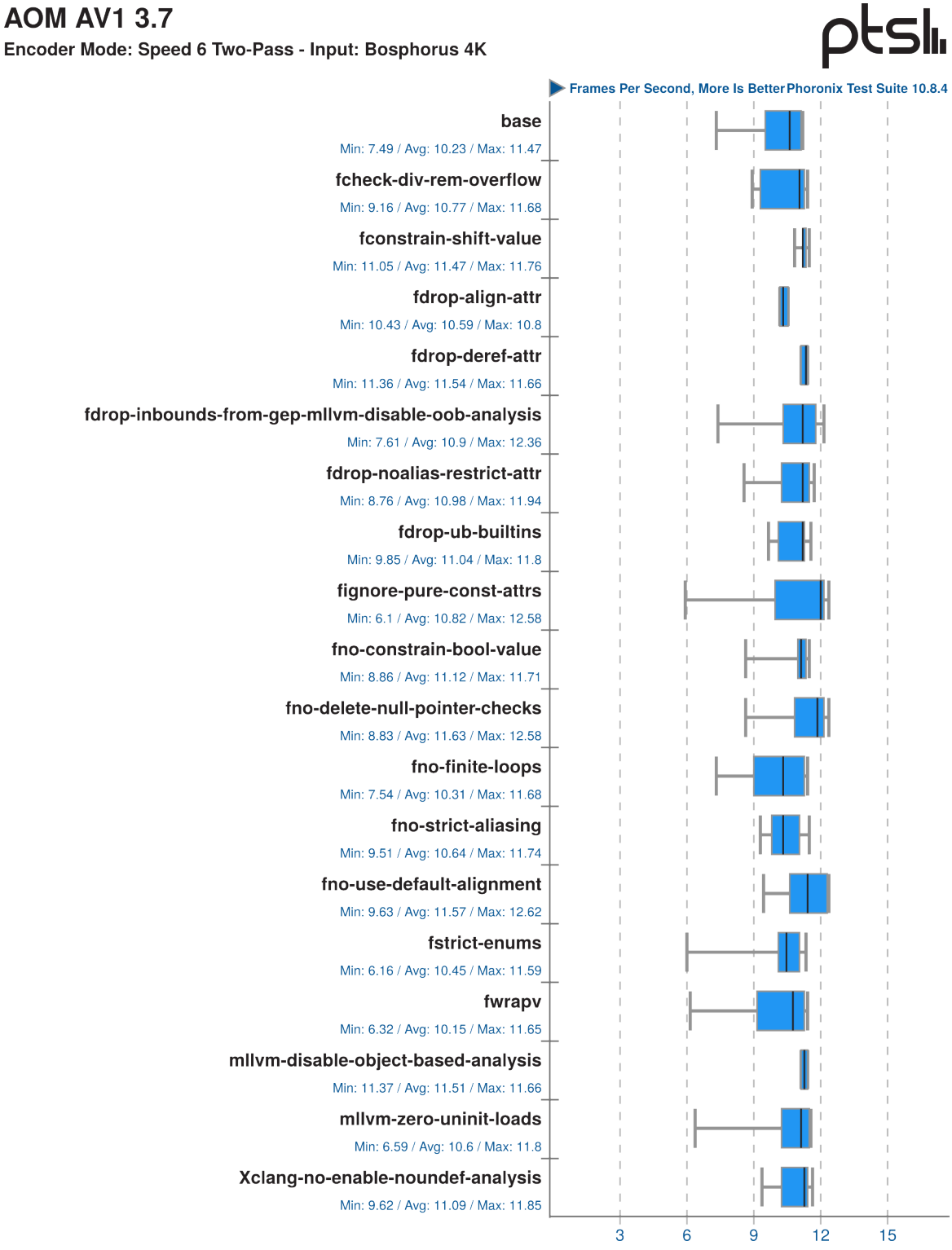

Figure 6: Variance of aom-av1 on different flags on arm

Benchmark Suite	
Application Category	Phoronix Application Identifier
LLVM Build Speed	build-llvm-1.4.0
HPC	fftw-1.2.0
Video encoding	aom-av1-3.7.0
JIT	luajit-1.1.0
Image Processsing	jpegxl-1.5.0 graphics-magick-2.1.0
Parallel Processing	tjbench-1.2.0 simdjson-2.0.1
Security	aircrack-ng-1.3.0 openssl-3.1.0
Password Cracking	john-the-ripper-1.8.0
Database	redis-1.4.0 sqlite-2.2.0
Audio Encoding	encode-flac-1.8.1
Texture Compression	basis-1.1.1 draco-1.6.0
Compression	compress-zstd-1.6.0 compress-pbzip2-1.6.0
Speech	espeak-1.7.0 rnnoise-1.0.2
Finance	quantlib-1.2.0
Circuit Simulator	ngspice-1.0.0
Webserver	apache-3.0.0 nginx-3.0.1
Theorem Prover	z3-1.0.0

Table 1: Benchmark suite that uses Phoronix applications. Second column contains Phoronix application identifiers, more information about them can be found at https://openbenchmarking.org/

Undefined behavior flags	
Flag	Description
-fwrapv	integer overflow does not generate poison
-fno-strict-aliasing	do not apply the strictest aliasing rules available
-fstrict-enums	(C++) poison if the program uses a cast (or an assignment) to convert an arbitrary integer value that does not fit in the enumerated type
-fno-delete-null-pointer-checks	asssume NULL pointer dereference is valid
-fno-finite-loops	assume no loop is finite
-fno-constrain-bool-value	"don't constrain bool values in 0,1"
-fconstrain-shift-value	mask shift amount to bit-witdh before executing the shift
-fno-use-default-alignment	pointers in loads/stores/memcpy's/etc are aligned to 1
-fdrop-inbounds-from-gep	drop poison generating 'inbounds' from GEP
-mllvm -zero-uninit-loads	all loads from uninitialized memory take the value 0
-mllvm -disable-oob-analysis	drop alias analysis that exploits out-of-bounds pointers
-mllvm -disable-object-based-analysis	drop alias analysis that compares the underlying objects of the pointers
-fcheck-div-rem-overflow	guard every div and rem operation to get rid of UB
-fdrop-noalias-restrict-attr	drops noalias attribute generated by 'restrict' pointers
-fdrop-align-attr	drop align attribute
-fdrop-deref-attr	drop dereferenceable and dereferenceable_or_null attributes
-Xclang -no-enable-noundef-analysis	drop noundef attribute
-fdrop-ub-builtins	ignores builtin_assume_aligned and replaces builtin_unreachable with builtin_trap
-fdrop-pure-const-attr	ignore pure and const function attributes

Table 2: Flags that control undefined behavior exploitation