Exploiting Undefined Behavior in C/C++ Programs for Optimization: A Study on the Performance Impact

LUCIAN POPESCU, Politehnica University of Bucharest, Romania and INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal
NUNO P. LOPES, INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal

The C and C++ languages define hundreds of cases as having undefined behavior (UB). These include, for example, corner cases where different CPU architectures disagree on the semantics of an instruction and the language does not want to force a specific implementation (e.g., shift by a value larger than the bitwidth). Another class of UB involves errors that the language chooses not to detect because it would be too expensive or impractical, such as dereferencing out-of-bounds pointers.

Although there is a common belief within the compiler community that UB enables certain optimizations that would not be possible otherwise, no rigorous large-scale studies have been conducted on this subject. At the same time, there is growing interest in eliminating UB from programming languages to improve security.

In this paper, we present the first comprehensive study that examines the performance impact of exploiting UB in C and C++ applications across multiple CPU architectures. Using LLVM, a compiler known for its extensive use of UB for optimizations, we demonstrate that, for the benchmarks and UB categories that we evaluated, the end-to-end performance gains are minimal. Moreover, when performance regresses, it can often be recovered through small improvements to optimization algorithms or by using link-time optimizations.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Software performance**; **Semantics**.

Additional Key Words and Phrases: Undefined Behavior, Compiler Optimizations, C, C++, LLVM

1 Introduction

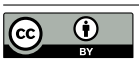
The C programming language, now over 50 years old, carries with it a significant amount of historical legacy. One of the key design choices made by its creators was to ensure that C programs could run efficiently across all hardware platforms available at the time [9]. This required that each language construct, such as an integer addition, be translatable into a single assembly instruction in order to maximize performance on the relatively simple and slow CPUs of that era.

As a consequence of this design goal, the language was standardized to reflect only the least common denominator semantics of several CPU architectures. For instance, although two's complement arithmetic is now universally adopted, it was not at the time. This led both C and C++ to define signed integer overflow as undefined behavior (UB), granting compilers the flexibility to map arithmetic operations to a single assembly instruction across most CPU architectures.

While all modern CPUs implement two's complement arithmetic, theoretically allowing the C/C++ standards to define all cases of integer overflow, certain disparities between CPU architectures persist. For example, the result of a shift by an amount equal to or greater than the bitwidth varies between ARM and x86. Standardizing one behavior over the other would necessitate additional instructions for some architectures, undermining the original goal of efficiency.

In addition to handling CPU architecture differences, the C and C++ standards use UB to avoid mandating the detection of certain errors that would be too costly or impractical to detect. For

Authors' Contact Information: Lucian Popescu, Politehnica University of Bucharest, Romania and INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal, lucian.popescu187@inesc-id.pt; Nuno P. Lopes, INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal, nuno.lopes@tecnico.ulisboa.pt.



example, detecting out-of-bounds memory accesses would require emitting additional code, which would likely reduce performance, contrary to the languages' design goals.

Over time, compiler developers realized that UB could be leveraged for more than just eliminating a few assembly instructions. For instance, the UB in integer overflows allows compilers to optimize `a + b > a` into `b > 0`. While this transformation is correct for pure integers, it does not hold in two's complement arithmetic. Ironically, such expressions are often incorrectly used to check if `a + b` overflows, leading to unexpected results.

Another example where compilers take advantage of UB in integer overflows for optimization is demonstrated in the following program (shown on the left):

```
for (int i=0; i <= n; ++i) {
  a[i] = 42;
}
```

```
for (int i=0; i <= n; ++i) {
  *(a + (long)i) = 42;
}
```

```
long n2 = (long)n;
for (long i=0; i <= n2; ++i) {
  *(a + i) = 42;
}
```

When compiling the left program for a 64-bit CPU, compilers must generate assembly code that resembles the program in the middle because pointers have 64 bits while the `int` type is usually defined to have just 32 bits. This results in an implicit cast in the original program.[1] Changing the induction variable `i` to have 64 bits and removing the sign-extend cast from the loop body (program on the right) improves the performance of this loop by up to 40% in some micro-architectures. However, this transformation is only legal because integer overflow is UB, and therefore the compiler is allowed to assume there is no overflow.

Over time, compilers have evolved to exploit UB for optimization, operating under the assumption that programs are well-defined. However, real-world programs often contain bugs that can trigger UB. A notable example (below) from the Linux kernel highlights the dangers of such optimizations [10]. Since dereferencing a null pointer triggers UB, the compiler can assume that after line 4, the pointer `tun` is non-null. Consequently, the compiler optimizes away the `if` statement, creating a security vulnerability.

```
1 unsigned tun_chr_poll(struct file *file) {
2   struct tun_file *tfile = file->private_data;
3   struct tun_struct *tun = __tun_get(tfile);
4   struct sock *sk = tun->sk; // dereferences tun; implies tun != NULL
5   if (!tun) // always false
6     return POLLERR;
7   ...
8 }
```

In response to such issues, some compilers introduced flags to disable the removal of null-pointer checks, even when the compiler can infer that the check is unnecessary. While this flag could have prevented the vulnerability in the example above, such mechanisms are not a panacea. If a program triggers UB, the resulting behavior remains unpredictable, and the effect of such flags may not align with the developer's expectations.

More recently, a study by Xu et al. [60] found over 4,000 bugs related with UB in open-source software. This and other studies have led to the development of several flags in the compilers to disable or detect certain classes of UB. For example, there is a proposal to initialize local (stack) variables in C++ [5]. Reports on the performance impact range from a 4% slowdown in the Linux kernel [42] to a 13% slowdown in a benchmark suite [62].

Despite the growing interest in mitigating UB in both programming languages and compilers, there has been no comprehensive study on the impact of UB in full-scale applications using modern

[1] Historically, developers learned to declare loop induction variables as `int`. Unfortunately, that results in a mismatch between the size of pointers and common array indexing types. C++ tries to avoid this situation with iterators.

hardware. Given that modern CPUs execute instructions out of order and have wide pipelines, the overhead of executing a few additional assembly instructions is often negligible.

In this paper, we present an exhaustive analysis of the various classes of UB exploited by the LLVM compiler, which is known for its extensive use of UB for optimization. We modified LLVM to selectively disable exploitation of each UB class, allowing us to measure the impact of each class on performance and code size across a variety of real-world applications.

In summary, the contributions of this paper are as follows:

(1) A comprehensive study of the classes of UB exploited for optimization by LLVM, which is widely recognized for its extensive use of UB.
(2) An extension to the Alive2 translation validation tool [34] to detect optimizations that rely on UB. This ensures we have thorough coverage of all UB classes exploited by LLVM.
(3) A detailed analysis of the run-time performance and code size impact associated with exploiting each class of UB for optimization.

2 Undefined Behavior

The C and C++ standards leave many aspects of program behavior undefined. However, most of these cases are benign in practice, as compilers typically provide reasonable, consistent behavior that developers can rely on. In some situations, compilers go beyond the standard's requirements by offering multiple levels of undefined behavior (UB) with stronger guarantees.

We focus on how UB is handled by LLVM, since it is particularly well-specified and because LLVM is widely known for leveraging UB for optimization. LLVM has two main classes of UB:

- Deferred UB, which is used to define corner cases for which LLVM does not wish to specify a concrete value, while allowing the operation to be executed anywhere. For example, a signed addition can be executed speculatively, even though the result in cases of overflow is not concretely defined.
- Immediate UB, which is reserved for operations that trigger hardware exceptions and thus cannot be executed speculatively, e.g., division by zero and dereferencing a null pointer.

The rationale for having these two classes of UB is to enable optimizations that would not be possible if all UB was immediate. For example, deferred UB allows LLVM to hoist arithmetic operations out of loops easily. Furthermore, LLVM has two values for representing deferred UB: `undef` and `poison`, with the former being weaker than the latter. Passing these values to certain operations triggers immediate UB.

The way that Clang (LLVM's C/C++ frontend) maps input programs into LLVM's intermediate representation (IR) is a refinement of the semantics given in the C and C++ standards. Clang offers stronger guarantees and leaves fewer behaviors as undefined. There are three reasons for this: (1) cases that offer no potential for optimization, (2) cases where LLVM tries to offer "nicer" semantics (e.g., not forcing the input pointers to `memcpy` to be non-null even if the size argument is zero), and (3) exploiting some UB leads to many programs misbehaving due to a common coding pattern triggering UB frequently in practice (e.g., forcing a value range for enums is opt-in via `-fstrict-enums`). The difference between the last two cases is that for the last case the compiler offers a set of flags to exploit more UB.

In this study, we wanted to understand the impact of each class of UB that LLVM exploits in practice for optimization. In order to do this, we modified LLVM to allow us to disable each class of UB individually. Note that we do not attempt to detect UB either statically or at run time; we merely want to disable the exploitation of UB. This contrasts with countermeasures, such as initializing all stack values to zero, which have a higher cost.

Table 1. Flags that disable UB exploitation in Clang/LLVM. The 'New?' column indicates the flags that we implemented. The last column indicates whether the LLVM IR is expressive enough to implement the flag.

Category	Flags	Acronym	New?	Frontend?
Arithmetic Operations	-fcheck-div-rem-overflow	AO1	✓	✓
	-fconstrain-shift-value	AO2	✓	✓
	-fwrapv	AO3		✓
Type Ranges	-fno-constrain-bool-value	TR1	✓	✓
	-fno-strict-enums	TR2		✓
Function Domains	-fignore-pure-const-attrs	FD1	✓	✓
	-fdrop-ub-builtins	FD2	✓	✓
	-fno-finite-loops	FD3		✓
Pointers and Memory	-fdrop-align-attr	PM1	✓	✓
	-fdrop-deref-attr	PM2	✓	✓
	-fno-delete-null-pointer-checks	PM3		✓
	-fdrop-noalias-restrict-attr	PM4	✓	✓
	-fno-strict-aliasing	PM5		✓
	-fno-use-default-alignment	PM6	✓	✓
	-Xclang -no-enable-noundef-analysis	PM7		✓
	-mllvm -zero-uninit-loads	PM8	✓	
Alias Analysis	-fdrop-inbounds-from-gep -mllvm -disable-oob-analysis	AA1	✓	
	-mllvm -disable-object-based-analysis	AA2	✓	

2.1 Disabling Exploitation of Undefined Behavior

Table 1 lists the five categories of UB that Clang/LLVM currently exploit for optimization. We further subdivide each category, for a total of 18 individual aspects that are exploited. We implemented 12 new flags in Clang/LLVM and use 6 existing ones to selectively disable UB.

We attempted to implement as many flags on the frontend side (Clang) as possible by changing the generated IR to be free from UB. The main reasons why we chose this approach are as follows:

(1) It is easier to ensure the implementation does not miss some case.
(2) LLVM is used by many languages. Any change to the IR or the optimizers can negatively impact the performance of these other languages.
(3) It allows us to benchmark the optimizer that is common across multiple languages and thus some of the results we obtain for C/C++ may transfer to other languages.

Unfortunately, LLVM's IR is not sufficiently expressive to implement all flags. We had to implement 3 flags directly in LLVM by changing the code of multiple static analyses and optimizations. We now describe the changes implemented by each flag. In the appendix, we show example programs affected by each flag, as well as the IR differences.

2.1.1 Arithmetic Operations. Flag AO1 instruments the IR to trap (a well-defined exit) in the cases where division and remainder would trigger UB (i.e., the divisor is zero, or the operation overflows).

Flag AO2 ensures that shift operations are well-defined by masking the shift amount so it is always smaller than the bitwidth. This matches the semantics of x86.

Flag AO3 makes signed overflow in arithmetic operations defined in term of two's complement arithmetic. In practice, this amounts to not adding the `nsw` attribute to arithmetic operations.

2.1.2 Type Ranges. Flag TR1 prevents the compiler from assuming that **`bool`** variables only have the integer value 0 or 1. This amounts to not emitting LLVM's `!range` metadata. Flag TR2 is similar, but for **`enum`**-typed variables.

2.1.3 Function Domains. Flag FD1 ignores the `__attribute__((const/pure))` annotations in the input programs. While these are not part of the C/C++ standards, many compilers support them.

Flag FD2 converts calls to `__builtin_unreachable()` into well-defined traps, and ignores calls to assume builtins. Again, these functions are not part of the C/C++ standards.

Flag FD3 prevents the compiler from assuming that all loops without side-effects terminate. This amounts to removing LLVM's `mustprogress` attribute from functions.

2.1.4 Pointers and Memory. Flags PM1 and PM2 prevent Clang from emitting, respectively, `align` and `dereferenceable` function argument attributes. For example, these are added to the "this" argument of C++ methods, so the optimizer can assume that the whole object is dereferenceable.

Flag PM3 prevents the compiler from assuming that null pointers are not dereferenceable. In particular, it prevents certain null pointer checks from being optimized away. The attribute `null_pointer_is_valid` is added to functions, the `dereferenceable` argument attributes are changed to `dereferenceable_or_null`, and the `nonnull` argument attribute is not emitted.

Flag PM4 makes Clang ignore the `restrict` keyword on pointers, i.e., it makes Clang no longer emit LLVM's `noalias` function argument attribute.

Flag PM5 prevents the compiler from using type-based alias analysis by having Clang skipping the emission of TBAA metadata.

Flag PM6 prevents the compiler from making assumptions about the alignment of data, i.e., all memory operations are emitted with alignment of 1 (i.e., they are possibly completely unaligned).

Flag PM7 prevents the compiler from assuming that function arguments and return values are well-defined values (i.e., they are not `undef` nor `poison`). In practice, this makes Clang skip emitting the `noundef` attribute.

Flag PM8 changes the value of uninitialized loads from `undef` to zero. Note that this is not the same as initializing all variables to zero! Instead, LLVM's optimizer is changed so that when it replaces such a load, it replaces it with zero instead of `undef`. This is to prevent subsequent optimizations that would take advantage of `undef` to prove that the code triggers UB, at a much smaller cost than explicitly initializing all variables. While this flag disables exploitation of UB, the semantics offered is mostly unchanged: loading uninitialized data yields a non-deterministic value, but it never triggers immediate UB.

2.1.5 Alias Analysis. These flags change the behavior of the alias analysis (AA) algorithms.

Flag AA1 prevents the compiler from assuming that the result of pointer arithmetic operations are always within bounds of the input object, as well as prevents AA from concluding no-alias for out-of-bounds accesses. This flag consists of two modifications: change Clang to emit LLVM's pointer arithmetic operations (`getelementptr`) without the `inbounds` attribute, and change the AA algorithm itself.

Flag AA2 prevents AA from using object provenance information for inference (e.g., `p + i` and `q + j` cannot alias if `p` and `q` are initialized with distinct calls to `malloc`).

Together, these flags offer a flat memory model.