Analysis of the Performance Gains of Undefined Behavior Optimizations

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

[*]Department of Computer Science, Univesity POLITEHNICA of Bucharest
[**]Instituto Superior Tecnico, Universidade de Lisaboa
[*]*lucian_ioan.popescu@stud.acs.upb.ro, razvan.deaconescu@cs.pub.ro*
[**]*nuno.lopes@tecnico.ulisboa.pt*

Wednesday 25th January, 2023

Abstract

State-of-the-art compilers, such as GCC and Clang/LLVM, use undefined behavior to issue optimizations. These optimizations may boost the performance of specific classes of programs, such as High Performance Computing. In contrast, other classes of programs, such as Operating Systems, might not benefit from the performance advantages of undefined behavior optimizations. In the latter case, disadvantages emerge in the form of security vulnerabilities. In this study we analyze the advantages and disadvantages of undefined behavior optimizations for various classes of programs. To achieve this we take Clang/LLVM and disable all undefined behavior optimizations. Then we analyze the preformance impact for each class of application using suitable benchmarks.

1 Context and motivation

The ISO C Standard [18] provides a definition of undefined behavior that gives absolute freedom to compiler implementation when erroneous program constructs, erroneous data or indeterminately-valued objects are encountered. This allows various compilers to treat undefined behavior in different ways while still being standard conformant. For example, signed overflow is undefined behavior as there are multiple ways of treating it based on the target architecture. On MIPS and DEC Alpha the ADD and ADDV instructions trap while on IA-32 the ADD instruction performs second's complement wrapping.

The freedom that undefined behavior supplies also gives birth to compiler opitmizations. The main philosophy here is that a program that triggers undefined behavior for some input is incorrect. The compiler can assume that the program it is gives is always correct and so, it takes advantage of undefined behavior by assuming it never happens. The following piece of code: $if(a + c < a + b)$ can be transformed, in this case, into the following piece of code: $if(c < b)$, assuming that the addition is signed. This is possible because signed oveflow is undefined, as discussed, and the compiler is free to infer the addition property of inequality.

Certain classes of programs, such as High Performance Computing take great advantage of undefined behavior for issuing compiler optimizations. Programs in this class contain tight loops that perform a a significant amount of operations per second. As a consqeuence programers are interested in generating code that runs as fast as possible. Not needing to perform bounds checks, not considering the case where the signed iterator of the loop oveflows or converting the loop iterator from signed to unsigned are just a few scenarios that can boost the performance of the code.

Wang et al. [22] experimented with the consequences of disabling undefined behavior optimizations on the SPECint 2006 benchmark. Out of 12 programs in the benchmark they noticed slowdowns for 2 of them. 456.hmmer slows down 7.2% with GCC 4.7 and 9.0% with Clang/LLVM 3.1 while 462.libquantum slows down 6.3% with the same version of GCC and 11.8% with the same version of Clang/LLVM.

In Operating Systems things behave differently than in the case of High Performance Computing. Operating Systems are part of a class of programs that are not CPU bound hence might not benefit from the performance advantages of undefined behavior. Instead they make part of the class of programs where security and robustness play an important role. Because of this, undefined behavior optimizations must be carefully analyzed so that they do not introduce security issues, as Linus Torvalds noted [7]:

> Performance doesn't come from occasional small and odd micro-optimizations. I care about performance a lot, and I actually look at generated code and do profiling etc. None of those three options [-fno-strict-overflow, -fno-strict-aliasing, -fno-deletel-null-pointer-checks] have *ever* shown up as issues. But the incorrect code they generate? It has.

The problems Torvalds complains about are often encountered in the field of Operating Systems. One of the most well known security issue found due to undefined behavior optimizations was found in BSD systems. Listing 1 presents a historical snippet of code from srandomdev(3), a function that initializes the random number generator of the system.

```c
void
srandomdev(void)
{
  ...
  struct timeval tv;
  unsigned long junk;

  gettimeofday(&tv, NULL);
  srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
  ...
}
```

Listing 1: srandom function in lib/libc/stdlib/random.c on BSD systems

It makes use of unitialized memory, through junk, for generating randomness. This is a typical case of undefined behavior and its consequences are best seen in the difference of the generated code by two different compilers.

By inspecting `/usr/lib/libSystem.B.dylib` in Mac OS X 10.6 we see the following generated code for Listing 1:

```
leaq     0xe0(%rbp),%rdi
xorl     %esi,%esi
callq    0x001422ca        ; symbol stub for: _gettimeofday
callq    0x00142270        ; symbol stub for: _getpid
movq     0xe0(%rbp),%rdx
movl     0xe8(%rbp),%edi
xorl     %edx,%edi
shll     $0x10,%eax
xorl     %eax,%edi
xorl     %ebx,%edi
callq    0x00142d68        ; symbol stub for: _srandom
```

The compiler allocates a slot on the stack for junk and interprets that slot as the value of the junk variable that will be used in the srandom argument computation.

In the next version of the same operating system, i.e. Mac OS X 10.7, the same file has the following generated code:

```
leaq     0xd8(%rbp),%rdi
xorl     %esi,%esi
callq    0x000a427e        ; symbol stub for: _gettimeofday
callq    0x000a3882        ; symbol stub for: _getpid
callq    0x000a4752        ; symbol stub for: _srandom
```

The newer version discards the seed computation as an optimization, generating code that calls gettimeofday and getpid but not using their values. Furthermore srandom is called with some garbage value that does not depend on the expression declared in the corresponding C code.

This change in the generated source code happened at approximately the same time at which Apple decided to not use GCC anymore due to license problems and switched to Clang/LLVM. Meanwhile, BSD systems solved this problem and srandomdev(3) uses defined behavior for generating random numbers [3, 5, 10].

In this context, it is important to analyze for each class of programs what are the advantages and the disadvantages of undefined behavior optimizations issued by the compiler based on the requirements of the class. Wang et al. [22] started the work in this field by analyzing the preformance of GCC and Clang with a limited set of disabled undefined behavior optimizations on SPECint 2006 benchmark.

We take their work one step further and analyze the advantages and disadvantages of undefined behavior optimizations for real-life software.

To achieve our goals, we take Clang/LLVM and disable all undefined behavior optimizations, e.g signed overflow optimizations, null access optimizaitions, oversized shift optimizations, etc. We do the modifications either through the already accessible compiler flags or by changing the internals of the compiler. The end result will be a modified compiler free of undefined behavior optimizations that will be used to test the advantages and disadvantages for each class of programs.

The metric we use for assessing the advantages and disadvantages is performance. The performance may include, but it is not limited to, code size and code speed. We take examples from each class of programs and create benchmarks that show how the performance has changed based on the modifications done in the compiler.

This study is structured as follows. [needs reformulaiton]

2 Background

This section presents UB optimizations in real-life software projects such as Linux and OpenBSD. After we presents such examples, we provide an analysis of the risks they introduce and current solutions that try to tackle the risks, but are however incomplete. The final part of this section present the situation of UB optimizations as seen by various acaedmics and C programmers in the field.

Wang et al. [22] compiled a list of UB optimizations that show the dangerous effects of using the UB definition when issuing compiler optimizations. They created case studies for the following classes of undefined behaviors: division by zero, oversized shift, signed integer overflow, out-of-bounds pointer, null pointer dereference, type-punned pointer dereference and uninitialized read. The consequences of these optimizations range from unexpected code generation [6, 13] to real-life vulnerabilities [9].

Code snippets with a high risk of triggering UB optimizations are provided in Listings 2, 3 and 4.

```
if (!msize)
  msize = 1 / msize; /* provoke a signal */
```

Listing 2: Compiler assumes that dividing a number by zero makes no sense and the whole block is deleted (lib/mpi/mpi-pow.c in the Linux kernel)

```
ifa = &in6ifa_ifpwithaddr(ifp,
  &satosin6(rt_key(rt))->sin6_addr)->ia_ifa;
if (ifa) {
  ...
}
```

Listing 3: Compiler assumes that in6ifa_ifpwithaddr returns NULL then makes ipa6 NULL and deletes the if check (sys/netinet6/nd6.c in the OpenBSD kernel)

```
static __inline int
hibe_cmp(struct hiballoc_entry *l, struct hiballoc_entry *r)
{
  return l < r ? -1 : (l > r);
}
```

Listing 4: Comparing pointers that do not point to the same aggregate or union is undefined behavior so the compiler is free to return anything from this function (sys/kern/subr_hibernate.c in the OpenBSD kernel)

The code shown in these examples was fixed up to this day [4,8,11] but the risk of existing code triggering uncatched UB optimizations still persists.

To address these issues the research community created solutions that tackle the problem from different angles. One approach was to introduce new compiler improvements that would catch undefined behaviors either at compile-time or at run-time. However such endeavours could not provide the expected results.

On one hand, generating reports for all undefined behaviors at compile-time is undecidable [17]. Moreover, generating such reports is unuseful in specific cases. Listing 5, for example, could generate reports such as:

- pointer a may originate from a non-integral or non-void pointer
- pointer a may be NULL
- variable b may be uninitialized

```c
void foo(int *a, int b) {
  *a = b;
}
```

Listing 5: Code that may report false undefined behavior

This is the case because the internal representation of the compiler may not have enough context to report only the useful information about undefined behaviors and because the compiler cannot understand the intention of the programmer when issuing an UB optimization. In this context, to issue UB optimizations is paradoxical. The compiler does not have the context to find and report undefined behaviors, but it uses undefined behaviors in order to generate code transformations [19].

On the other hand, catching undefined behavior at run-time proves to be an incomplete approach. The run-time checker would need to visit all the states of the program in order to ensure that no undefined behavior is triggered. To catch all states that may contain undefined behavior we need to run the checker for as long as it requires, which may not be desirable in most cases because it may take too much time. Checkers for this task are IOC [15], UBsan [12] and various compiler flags such as GCC's -ftrapv and Clang's -fcatch-undefined-behavior.

Another approach for run-time checking is to compare the unoptimized code with the optimized code generated by the compiler. However program equivalence is undecidable [21]. Also, decompilation might be used to compute the semantic distance between the original C code and the decompiled optimized assembly code. Doing so we could spot the introduced UB optimizations and delete them later. However decompilation is a hard problem in general [14] because of type erasure.

Besides the introduction of compiler improvements, another solution would be to issue additions to the standard that would provide more robustness to the definition of undefined behavior. At the moment, state-of-the-art compilers, such as GCC and Clang/LLVM, take a liberal view of the standard and interpret it in a way that allows them to push various dangerous optimizations. The opposite view is the constructivist one, where the compiler implementations construct a robust definition of undefined behavior, even if the standard imposes no strong requirements. Until the standard makes it clear what approach it would take in the future, implementations and developers need to decide their approach based on the ambiguous definition provided in the standard.

Various academics and C programmers have complained throughout the years about this situation. Linus Torvalds [7] said in 2016 on the GCC mailing list (paranthesis mine):

> The fact is, undefined compiler behavior is never a good idea. Not for serious projects.
>
> Performance doesn't come from occasional small and odd micro-optimizations. I care about performance a lot, and I actually look at generated code and do profiling etc. None of those three options (-fno-strict-overflow, -fno-strict-aliasing and -fno-deletel-null-pointer-checks) have *ever* shown up as issues.

> But the incorrect code they generate? It has.

John Regehr wrote in one of his blog posts:

> One suspects that the C standard body simply got used to throwing behaviors into the “undefined” bucket and got a little carried away. Actually, since the C99 standard lists 191 different kinds of undefined behavior, it’s fair to say they got a lot carried away.

DJ Bernstein, one of the most important voice in writing cryptography code, wrote:

> Pretty much every real-world C program is ”undefined” according to the C ”standard”, and new compiler ”optimizations” often produce new security holes in the resulting object code, as illustrated by
>
> https://lwn.net/Articles/342330/ https://kb.isc.org/article/AA-01167
>
> and many other examples. Crypto code isn’t magically immune to this

Bernstein’s complaint was heard by GCC developers and they tried to start an initiative for creating a boringcc dialect in GCC. However the efforts is stopped because human resources are missing.

Finally, Dennis Ritchie also commented about the dangerous effect of noalias, an early undefined behavior that was in the end dropped by the ANSI C standard:

> ‘Noalias’ is much more dangerous; the committee is planting timebombs that are sure to explode in people’s faces. Assigning an ordinary pointer to a pointer to a ‘noalias’ object is a license for the compiler to undertake aggressive optimizations that are completely legal by the committee’s rules, but make hash of apparently safe programs. Again, the problem is most visible in the library; parameters declared ‘noalias type *’ are especially problematical.

While noalias does not exist in current standards, the ”timebomb” effects that Ritchie describes can be found in many other undefined behaviors as described in the examples provided in this section.

3 Related Work

Little work has been done in the area of detecting the performance speedup of UB optimizations in real-life software projects. Wang et al. [22] and Ertl [16] provide metrics for this class of optimizations based on SPECint benchmark.

Wang et al. state that they observed a decrease in performance of 7.2% with GCC and 9.0% with Clang for 456.hammer and 6.3% with GCC and 11.8% with Clang for 462.libquantum. The experiments were conducted with UB optimizations turned off.

Ertl states that with Clang-3.1 and UB optimizations turned on the speedup factor is 1.017 for SPECint 2006. Furthermore, for a specific class of programs, i.e. Jon Bentley’s traveling salesman problem, the speedup factor can reach values greater than 2.7 if the programmer issues source-level optimizations by hand, surpassing the UB optimizations issued by the compiler.

4 Research Plan

Given the little research done in the field of analysing the performance of UB optimizations, this study aims to provide insights of the performance of these optimizations on a specific class of software applications, i.e. operating systems.

The first step of our work is to filter out all undefined behavior instances presented in the standard and focus on the undefined behaviors that present a potential for being used in compiler optimizations. Our filtering strategy is based on the assumption that all undefined behaviors that conflict with the intentionality of the programmer shall not be used to issue code optimizations.

Then we either modify the compiler implementation or use compiler-specific flags to turn off these optimizations. A preliminary list of such undefined behaviors extracted from the standard [18] is:

- An arithmetic operation is invalid (such as division or modulus by 0) or produces a result that cannot be represented in the space provided (such as overflow or underflow) ($3.3).
- An invalid array reference, null pointer reference, or reference to an object declared with automatic storage duration in a terminated block occurs ($3.3.3.2).
- A pointer is converted to other than an integral or pointer type ($3.3.4).

The first undefined behavior could lead to code being eliminated if the compiler detects that the arithmetic operation is incompatible with the standard [22]. The second undefined behavior could discard security checks for NULL pointers [2] and the third undefined behavior could break manual optimizations on floating point numbers [20].

To analyze the role of these optimizations in real-life software, we take a self-contained operating system with focus on robustness and security, i.e. OpenBSD, and compile it on one hand with UB optimizations turned on and on the other hand with UB optimizations turned off. After this stage, the result will be two comparison candidates which will be tested against various benchmarks that will highlight the advantages and disadvantages of the UB optimizations.

Furthermore, we analyze the role of UB optimizations in the various hardware architectures that OpenBSD supports [1]. We suspect that there are hardware setups on which undefined behaviors play a bigger role in compiler optimizations. At the same time, we want to see how the compiler treats robustness and security for each hardware architecture.

At this moment we do not know which components of the systems will be modified when UB optimizations are turned off so we cannot provide the benchmarks that we intend to use. However after we get this information, we plan to create a modification map that will help us visualize the components with the highest rate of modification. After this step, what we will do is to provide benchmarks that will focus on those specific components.

The final result will be a fine grained comparison between the two test candidates that will focus on one hand of speed and performance and on the other hand on robustness and security.

5 Conclusions

The definition of undefined behavior is used by compiler implementations to issue aggressive optimizations. We argue that this class of optimization is very dangerous as it conflicts with programmer intentionality and with a robust definition of code semantics. In this study analyze the performance impact of undefined behavior optimizations in real-life software projects. By doing so we evaluate if the advantages of issuing UB optimizations surpass the security risks that they introduce. In order to do this we take a robust and secure implementation of operating system, i.e. OpenBSD, and compare the system that contains UB optimizations with the same system without UB optimizations. The comparison is done on multiple hardware architectures to inspect what role UB optimizations have for various hardware setups.

References

[1] OpenBSD platforms. `https://www.openbsd.org/plat.html`, last visited Wednesday 25th January, 2023.

[2] Add -fno-delete-null-pointer-checks to GCC CFLAGS, July 2009. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a3ca86aea507904148870946d599e07a340b39bf`, last visited Wednesday 25th January, 2023.

[3] FreeBSD solution to srandomdev vulnerability, Oct 2012. `https://github.com/freebsd/freebsd-src/commit/6a762eb23ea5f31e65cfa12602937f39a14e9b0c`, last visited Wednesday 25th January, 2023.

[4] Solution for UB from listing 2, Feb 2012. `https://github.com/torvalds/linux/commit/e87c5e35a92e045de75fb6ae9846a38bdd0f92bd`, last visited Wednesday 25th January, 2023.

[5] DragonFlyBSD solution to srandomdev vulnerability, Oct 2013. `https://github.com/DragonFlyBSD/DragonFlyBSD/commit/ebbb4b97bba5f71fea11be7f7df933ecaf76a6e5#diff-045a36943e1f0636c8c83adfc7bd60f403f4538e58a43a439621f8359ab4ccae`, last visited Wednesday 25th January, 2023.

[6] Undefined behavior and fermat's last theorem, March 2015. `https://web.archive.org/web/20201108094235/https://kukuruku.co/post/undefined-behavior-and-fermats-last-theorem/`, last visited Wednesday 25th January, 2023.

[7] Re:

$$isocpp - parallel$$

proposal for new memory_order_consume definition, 2016. `https://gcc.gnu.org/legacy-ml/gcc/2016-02/msg00381.html`, last visited Wednesday 25th January, 2023.

[8] Solution for UB from listing 4, Aug 2016. `https://marc.info/?l=openbsd-tech&m=147263379303192&w=2`, last visited Wednesday 25th January, 2023.

[9] Cve records on undefined behavior vulnerabilities, 2022. `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=undefined+behavior`, last visited Wednesday 25th January, 2023.

[10] OpenBSD solution to srandomdev vulnerability, Dec 2022. `https://github.com/openbsd/src/commit/99d815f892ce481695caf21f08f773f563820a66`, last visited Wednesday 25th January, 2023.

[11] Solution for UB from listing 3, Jan 2022. `https://marc.info/?l=openbsd-tech&m=164332561831177&w=2`, last visited Wednesday 25th January, 2023.

[12] Undefined behavior sanitizer, 2022. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, last visited Wednesday 25th January, 2023.

[13] R. Chen. Undefined behavior can result in time travel, June 2014. `https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633`, last visited Wednesday 25th January, 2023.

[14] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[15] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–29, 2015.

[16] M. A. Ertl. What every compiler writer should know about programmers or optimization based on undefined behaviour hurts performance. In *Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*, 2015.

[17] C. Hathhorn, C. Ellison, and G. Roşu. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 336–345, 2015.

[18] Programming languages — C. Standard, International Organization for Standardization, Dec 1990.

[19] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in llvm. *ACM SIGPLAN Notices*, 52(6):633–647, 2017.

[20] C. Lomont. Fast inverse square root. *Tech-315 nical Report*, 32, 2003.

[21] M. Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

[22] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, pages 1–7, 2012.