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*

Thursday 26th 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 [10] 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. [12] 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.

```
1  void
2  srandomdev(void)
3  {
4    ...
5    struct timeval tv;
6    unsigned long junk;
7
8    gettimeofday(&tv, NULL);
9    srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
10   ...
11 }
```

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 [5, 6, 8].

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. [12] 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 starts with a short history of how compilers started to take advantage of undefined behavior optimizations. Then we present some of the most popular optimizations based on undefined behavior such as signed integer overflow or null access. Finally, we talk about the problem of benchmarking these optimizations on real-life software.

The first C compilers used for UNIX written by Ritchie et al. are operating on the idea that C is a high level assembler. Each C instruction has a direct mapping to its corresponding assembly code. No code transformations that increase the performance of the final binary are found in this type of compiler. This is proven by the small size of such compilers [3].

As C increased in popularity, it started being ported on different hardware with various requirements. Because of this, the code transformations performed by the compiler increased in complexity in order to be able to keep up with the performance requirements of the new hardware systems.

Based on this situation the first C standard [10] decided to give freedom to compiler implementations to take advantage of complex code transformations using the definition of undefined behavior. Instead of making strong assumptions about the behavior of the code, the Standard puts no responsability on the compiler and allows it to generate various optimizations based on the target hardware architecture.

At this moment in time C evolved from a language used only for Operating Systems development to a language with a broader rich based on its high hardware adoptance and on its performance advantages. Complex text editors, browsers, databases, compilers, version control systems, are just a few examples of programs that choose C as their main programming language.

As a result, people involved in this ecosystem started to pay more attention to the compiler generated code. Vallat [1] describes this state of things as a schism. On one hand we had GCC 2.8 people "conservative but trying to catch up on C++98" and on the other hand we had the Pentium GCC group "attempting to produce faster code by stretching the optimizer code beyond its limits".

"These projects eventually merged as gcc 2.95". From this moment on state-of-the-art compilers such as GCC and later Clang/LLVM focused their attention on generating the fastest possible code.

Next, we will cover some cases of undefined behavior optimizations illustrating on one hand their advantages and on the other hand their disadvantages.

The signed integer overflow optimization is the most famous undefined behavior optimization. C compilers are unable to generate fast code for architectures where the size of int is different from the register width. However, for backwards compatibility reasons, int is still 32 bit on all 64 bit major architectures. This creates problems when int is used for generating memory addresses and for generating memory accesses.

The issue is fairly common when generating code for loop variables. Given the following piece of code:

```
sum = 0;
for (int i = 0; i < count; ++i)
  sum += x[i];
```

the compiler can naively generate the following piece of assembly code:

```
                              ; ecx = count
                              ; rsi = points to x[]
    xor     eax, eax          ; clear sum
    xor     ebx, ebx          ; i
  lp:
    movsxd  rdx, ebx          ; sign-extend i to 64 bits
    add     eax, [rsi+rdx*4]  ; sum += x[i]
    inc     ebx               ; i++
    cmp     ebx, ecx          ; if i < count
    jl      lp                ;   goto lp
  done:
```

Because the loop counter is a 32 bit value it needs to be sign extended to 64 bits in order to be able to access the memory at x[i]. Things get worse when the compiler is free to loop unroll the computation inside the loop:

```
  lp:
    movsxd  rdi, ebx          ; sign-extend i to 64 bits
    add     eax, [rsi+rdi*4]  ; sum += x[i]
    lea     edi, [ebx+1]      ; i+1
    movsxd  rdi, edi          ; sign-extend i+1 to 64 bits
    add     eax, [rsi+rdi*4]  ; sum += x[i+1]
    lea     edi, [ebx+2]      ; i+2
    movsxd  rdi, edi          ; sign-extend i+2 to 64 bits
    add     eax, [rsi+rdi*4]  ; sum += x[i+2]
    lea     edi, [ebx+3]      ; i+3
    movsxd  rdi, edi          ; sign-extend i+3 to 64 bits
    add     eax, [rsi+rdi*4]  ; sum += x[i+3]
    add     ebx, 4            ; i=i+4
```

Here, the compiler is free to do loop unrolling but because the variable is declared on 32 bits it needs to extend it to 64 bits every time it wants to use it. An alternative would be to sign extend i only one time and them use the 64 bit value for the further computations but for this to be possible it is neccesary that the compiler proves that i will never overflow.

However we can use the fact that signed overflow is undefined and promote the int variable to a int64 type. The resulting C could will look as follows:

```
sum = 0;
int64 i64 = (int64) i;
for (i64 = 0; i64 < (int64) count; ++i64)
  sum += x[i64];
```

In this situation the unrolled code can be transformed into:

```
  lp:
    add     eax, [rsi+0]      ; sum += x[i+0]
    add     eax, [rsi+4]      ; sum += x[i+1]
```

```
4       add     eax, [rsi+8]      ; sum += x[i+2]
5       add     eax, [rsi+12]     ; sum += x[i+3]
6       add     rsi, 16           ; x += 4
```

In this situation the width of the loop counter is equal to the width of the pointer and the compiler is free to perform further optimizations that take advantage of pointer arithmetic.

Another class of undefined behavior optimizations is based on the idea that NULL accesses are not defined by the standard. GCC was the first to take advantage of this using -fdelete-null-pointer-check. By using global dataflow analysis the compiler can eliminate useless checks for null pointer. If a pointer is checked after it was dereferenced, it cannot be NULL.

However some environments depend on the assumption that dereferencing a NULL pointer does not cause a problem. This has caused a security vulnerability in the Linux kernel, more specifically in the tun_chr_poll function shown in Listing 2.

```
1  unsigned int
2  tun_chr_poll(struct file *file, poll_table * wait)
3  {
4    struct tun_file *tfile = file->private_data;
5    struct tun_struct *tun = __tun_get(tfile);
6    struct sock *sk = tun->sk;
7    if (!tun)
8      return POLLERR;
9    ...
10 }
```

Listing 2: tun_chr_poll in drivers/net/tun.c of the Linux kernel

The compilers notices that the tun pointer is dereferenced before it is NULL-checked and as an optimization it deletes the NULL check. If the memory page where NULL is present is not mapped then the kernel will generate a segmentaiton violation and it will halt. This happens regardless of the deleted NULL check. However if the NULL check is deleted and the page that contains NULL is mapped then the function does neither return a POLLERR nor generates a segmentation violation. In this point the system is vulnerable as an attacker could inject dangerous information in the page where NULL is present.

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. [12] and Ertl [9] 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 [10] 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 [12]. The second undefined behavior could discard security checks for NULL pointers [4] and the third undefined behavior could break manual optimizations on floating point numbers [11].

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 [2]. 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] Compilers in OpenBSD. `https://marc.info/?l=openbsd-misc&m=137530560232232&w=2`, last visited Thursday 26th January, 2023.

[2] OpenBSD platforms. `https://www.openbsd.org/plat.html`, last visited Thursday 26th January, 2023.

[3] Very early c compilers and language. `https://www.bell-labs.com/usr/dmr/www/primevalC.html`, last visited Thursday 26th January, 2023.

[4] 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 Thursday 26th January, 2023.

[5] FreeBSD solution to srandomdev vulnerability, Oct 2012. `https://github.com/freebsd/freebsd-src/commit/6a762eb23ea5f31e65cfa12602937f39a14e9b0c`, last visited Thursday 26th January, 2023.

[6] DragonFlyBSD solution to srandomdev vulnerability, Oct 2013. `https://github.com/DragonFlyBSD/DragonFlyBSD/commit/ebbb4b97bba5f71fea11be7f7df933ecaf76a6e5#diff-045a36943e1f0636c8c83adfc7bd60f403f4538e58a43a439621f8359ab4ccae`, last visited Thursday 26th 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 Thursday 26th January, 2023.

[8] OpenBSD solution to srandomdev vulnerability, Dec 2022. `https://github.com/openbsd/src/commit/99d815f892ce481695caf21f08f773f563820a66`, last visited Thursday 26th January, 2023.

[9] 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.

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

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

[12] 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.