Analysis of the Performance Gains of Undefined Behavior Optimizations

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

[*]Department of Computer Science, Univesity POLITEHNICA of Bucharest
[**]Instituto Superior T[Pleaseinsert"PrerenderUnicode–"intopreamble]cnico, Universidade de Lisboa
[*]*lucian_ioan.popescu@stud.acs.upb.ro, razvan.deaconescu@cs.pub.ro*
[**]*nuno.lopes@tecnico.ulisboa.pt*

Tuesday 31st January, 2023

Abstract

State-of-the-art compilers, such as Clang/LLVM, use undefined behavior to issue optimizations. In this study we explore the impact of undefined behavior optimizations for a diverse set of application categories. The results can be used by application developers who may choose to disable certain such optimizations becuause they buy low performance benefits or they are considered to dangerous. Furthermore we present an extensive catalogue of all undefined behavior optimizations that are used today.

1 Context and motivation

The ISO C Standard [5] 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.

Wang et al. [7] 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 addition to the above mentioned performance results, undefined behavior optimizations can give birth to unexpected code generation. Listing 1 presents a historical snippet of code from srandomdev(3), a function that initializes the random number generator of the system.

```
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 [1–3].

In this context, it is important to analyze for each application category what are the advantages and the disadvantages of undefined behavior optimizations issued by the compiler based on the requirements of the category. Wang et al. [7] started the work in this field by analyzing the preformance of GCC and Clang/LLVM 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 a diverse set of application categories. Furthermore we present an extensive catalogue of all undefined behavior optimizations that are used today.

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 application categories.

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 application categories and create benchmarks that show how the performance has changed based on the modifications done in the compiler.

This study is structured as follows. Section 2 provides examples of undefined behavior optimizations. Section 3 presents the previous work in the field of assessing the performance impact of this class of optimizations. In Section 4 we present our research plan. Finally, Section 5 summarizes this work providing an overview of our goals.

2 Background

This section presents two examples of undefined behavior optimizations, i.e. signed overflow optimizations and null access optimizations.

2.1 Signed Overflow Optimizations

The signed integer overflow optimization is highly used in compilers nowdays. 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 complicate 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 count64 = (int64) count;
for (int64 i = 0; i < count64; ++i)
  sum += x[i];
```

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]
    add     eax, [rsi+8]      ; sum += x[i+2]
    add     eax, [rsi+12]     ; sum += x[i+3]
    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.

2.2 NULL Access Optimizations

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

Wang et al. [7] and Ertl [4] provide metrics for undefined based optimizations based on the SPECint benchmark.

Wang et al. 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.

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

4 Research Plan

In the second semester we plan to disable all undefined behavior optimizations from the frontend of the compiler, i.e. Clang. Sources of undefined behavior include:

- signed integer overflow
- pointer overflow
- NULL pointer dereference
- shift overflow
- uninitialized load
- use-after-free
- out-of-bounds accesses
- infinite loop

In parallel, we plan to create a benchmarking infrastructure. We use this infrastructure for detecting the performance changes for various program categories. To achieve this, we compile each category with different sets of undefined behavior optimizations. Doing this we understand the impact of the frontend optimizations on each program category.

In the third semester we tackle the optimizations present in LLVM and the backend of the compiler. Compared to Clang, LLVM makes more aggressive use of undefined behavior optimizations. In the first phase we will disable the optimizations that we know about. Later, in the second phase, we might use Alive2 [6] to find new sources of undefined behavior that are used by LLVM.

For benchmarking we will use the same approach as in the first semester. The plan is to make the modifications in LLVM easily configurable, i.e. create flags that enable and disable the corresponding modifications.

We plan to continue this work in the fourth semester if we discover that there is more modification work than we expected. Otherwise we will focus on providing insights on the security side of undefined behavior optimizations, as opposed to the performance side.

5 Conclusions

State-of-the-art compilers, such as Clang/LLVM, use undefined behavior to issue optimizations. However there is no study that presents exactly what optimizations are used in Clang/LLVM and what their effects are on various application categories. In this work we fill this gap by making the neccessary changes in the compiler to disable these optimizations. Then we benchmark a diverse set of application categories compiled with our compiler in order to assess the performance when the optimizations are disabled. For the performance metrics, we focus on code speed and code size.

References

[1] FreeBSD solution to srandomdev vulnerability, Oct 2012. https://github.com/freebsd/freebsd-src/commit/6a762eb23ea5f31e65cfa12602937f39a14e9b0c, last visited Tuesday 31st January, 2023.

[2] DragonFlyBSD solution to srandomdev vulnerability, Oct 2013. https://github.com/DragonFlyBSD/DragonFlyBSD/commit/ebbb4b97bba5f71fea11be7f7df933ecaf76a6e5#diff-045a36943e1f0636c8c83adfc7bd60f403f4538e58a43a439621f8359ab4ccae, last visited Tuesday 31st January, 2023.

[3] OpenBSD solution to srandomdev vulnerability, Dec 2022. https://github.com/openbsd/src/commit/99d815f892ce481695caf21f08f773f563820a66, last visited Tuesday 31st January, 2023.

[4] 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.

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

[6] 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.

[7] 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.