Undefined Behavior and Compiler Optimizations

John Regehr, University of Utah

Based on joint work with: Nuno Lopes, Sanjoy Das, David Majnemer, Juneyoung Lee, Gil Hur, Ralf Jung, Zhengyang Liu, and others

Recent UB talks available online:

- “Garbage In, Garbage Out”

  —Chandler Carruth, CppCon 2016
- “Undefined Behavior is Magic”

  —Michael Spencer, CppCon 2016
- “Undefined Behavior is Awesome”

  —Piotr Padlewski, CppCon 2017
- “Undefined Behavior in 2017”

  —Me, CppCon 2017

30- Use after free B in C++

Signed integer overflow

- It's stuff you're not supposed
  - Some UBs are listed in the standard, some are

Modify a string literal

Strict aliasing violations

- And nothing actually stops yo
- Consequences range from absolutely nothing

e exe

Array OOB

Divide by zero

- This talk: How the compiler thinks about undefined behavior
  - “Compiler” means “heavily optimizing compiler”
- Everything you already understand about UB still applies
- My goal is to convince you of a handful of points

- Point 1: We want UB in a compiler IR

```
int fo
  int
  for
    if

  retu
}
```

In this talk:

- I'll be using pseudo-LLVM-IR to motivate the issues
  - IR is the language of the "middle end" of the compiler
- Similar issues come up in all optimizing C++ compilers

```
4 ]
%4 ]
     br i1 %11, label %4, label %3
3:   ret i32 %9
}
```

- Point 2: UB facilitates optimization by allowing safety checks to be decoupled from unsafe operations

```
while (…) {
  …
  z = safe_div(x,y)
  …
}
```

```
while (…) {
  …
  assert(y != 0)
  z = raw_div(x,y)
  …
}
```

```
assert(y != 0)
while (…) {
  …
  z = raw_div(x,y)
  …
}
```

- Safe languages like Rust and Swift benefit as much from UB in the compiler as C and C++ do

```
func add(a : Int, b : Int)->Int {
  return (a & 0xffff) + (b & 0xffff)
}
```

```
define i64 @_T04main3addS2i1a_Si1btF(i64, i64) {
entry:
  %2 = and i64 %0, 65535
  %3 = and i64 %1, 65535
  %4 = add nuw nsw i64 %3, %2
  ret i64 %4
}
```

```
7:   ret i64 %5

8:   call void @llvm.trap()
     unreachable
}
```

- Point 3: “All-or-nothing” semantics for UB is not appropriate in a compiler IR
  - UB in C++ is like setting off a bomb in the program
    - It blows up immediately
  - Compilers need speculation-friendly forms of UB
    - More like a time bomb: may go off later or may turn out to be a dud

```
for (int i = 0; i < n; ++i) {
  a[i] = x + 1;
}
```

Requires either
$n < 1$ or
$x \neq$ INT_MAX

```
init:
  br %head

head:
  %i = phi [ 0, %init ], [ %i1, %body ]
  %c = icmp slt %i, %n
  br %c, %body, %exit

body:
  %x1 = add nsw %x, 1
  %ptr = getelementptr %a, %i
  store %x1, %ptr
  %i1 = add nsw %i, 1
  br %head
```

```c
for (int i = 0; i < n; ++i) {
  a[i] = x + 1;
}
```

Requires either
n < 1 or
x ≠ INT_MAX

```llvm
init:

  br %head

head:
  %i = phi [ 0, %init ], [ %i1, %body ]
  %c = icmp slt %i, %n
  br %c, %body, %exit

body:
  %x1 = add nsw %x, 1
  %ptr = getelementptr %a, %i
  store %x1, %ptr
  %i1 = add nsw %i, 1
  br %head
```

```
(int i = 0; i < n; ++i) {
] = x + 1;
```

Is this a problem?

Requires either
n < 1 or
x ≠ INT_MAX

```
init:
   %x1 = add nsw %x, 1
   br %head

head:
   %i = phi [ 0, %init ], [ %i1, %body ]
   %c = icmp slt %i, %n
   br %c, %body, %exit

body:

   %ptr = getelementptr %a, %i
   store %x1, %ptr
   %i1 = add nsw %i, 1
   br %head
```

If x == INT_MAX and integer overflow is "immediate UB" then…

```
r (int i = 0; i < n; ++i) {
a[i] = x + 1;
```

Requires either n < 1 or x ≠ INT_MAX

```
init:
   %x1 = ad                  1
   br %head

head:
   %i = phi [ 0, %init ], [ %i1, %body ]
   %c = icmp slt %i, %n
   br %c, %body, %exit

body:

   %ptr = getelementptr %a, %i
   store %x1, %ptr
   %i1 = add nsw %i, 1
   br %head
```

```
for (int i = 0; i < n; ++i) {
  a[i] = x + 1;
}
```

Requires either
n < 1 or
AX

Compiler must never make code less defined

```
%ptr = getelementptr %a, %i
store %x1, %ptr
%i1 = add nsw %i, 1
br %head
```

- But if UB has a deferred effect…
- Case 1: x != INT_MAX

```
init:
  %x1 = add nsw %x, 1
  br %head
```

No problem

```
, [ %i1, %body ]
  br %c, %body, %exit

body:

  %ptr = getelementptr %a, %i
  store %x1, %ptr
  %i1 = add nsw %i, 1
  br %head
```

- But if UB has a deferred effect…
- Case 2: x == INT_MAX, n < 1

```
init:
  %x1 = undef
  br %head
```

No problem

```
, [ %i1, %body ]
  br %c, %body, %exit

body:

  %ptr = getelementptr %a, %i
  store %x1, %ptr
  %i1 = add nsw %i, 1
  br %head
```

- But if UB has a deferred effect…
- Case 3: x == INT_MAX, n ≥ 1

```
init:
  %x1 = undef
  br %head

No problem
, [ %i1, %body ]
  br %c, %body, %exit

body:

  %ptr      elementptr %a, %i
  sto       tr
  %i1       w %i, 1
  br %hea
```

- Deferred UB can work!
- What does “work” mean?
  - It means we have a tool we can use to justify desirable compiler optimizations
- But we must be really careful defining what it means
- E.g. what happens when?
  - undef is xor’ed with itself?
  - undef is used in a branch condition?
- A key issue is that UB appears on both sides of a compiler optimization

- Undef means “nondeterministic value choice”
- But here’s an optimization we’d like to do:

  –

- D

  –

  –

“Nondeterministic value choice” for undef does not justify optimizations such as

$(a + b) > a$

➔

$b > 0$

which is false, since no integer value is >INT_MAX

but 1 > 0 evaluates to true, OOOPS!

- LLVM adds a second kind of deferred UB
  - The **poison** value is more contagious than undef
    - undef & 0 ➔ 0
    - poison & 0 ➔ poison
    - Poison propagates unconditionally and does not have a value-based interpretation
- Good: Poison justifies (a + b) > a ➔ b > 0 and other nice things
- Bad: Reasoning about two kinds of UB can be really, really hard

- Point 4: Inside the optimizer you can make whatever rules you want
  - It's a self-contained world
  - There's no objective good or evil
    - Only engineering tradeoffs
- UB weakens the specifications for operations
  - The adds freedom on the left side (source) of an optimization
  - But it adds restrictions on the right side (target)!
  - You have to pay the piper
  - The tradeoffs are subtle

- Point 5: A compiler IR is a real computer language
  - Not OK to perform optimizations just because they feel right
- IR needs a precisely defined semantics
  - Every optimization must justified by math
  - So we’re stuck doing lots of proofs
- Few real IRs have received this sort of scrutiny

- Compiler optimizations are about “refinement”
  – Not equivalence!
- For every value of the inputs…
  – If the LHS is UB, the RHS can do anything
  – If the LHS is defined, the RHS
    - Is not UB
    - Computes the same output
- If every step in a compilation is a refinement, then the entire compilation is a refinement
  – And therefore correct

Which LLVM transformations are refinements?

- add nsw ➔ add
  - nsw: signed overflow is undefined
  - Unqualified “add” is two’s complement
- add nuw nsw ➔ add nsw
  - nuw: unsigned overflow is undefined
- add nsw ➔ add nuw

```
int x;
if (cond)
  x = a;
else
  x = b;
```

➔

```
  br %cond, %true, %false
true:
  br %merge
false:
  br %merge
merge:
  %x = phi [ %a, %true ], [ %b, %false ]
```

```
%x = select %cond, %a, %b
```

```
%y = select %cond2, true, %w
```

```
%y = or %cond2, %w
```

You can't always get what you want

```
$ cat foo.ll
define i1 @main(i1 %cond, i1 %b) {
  %a = select i1 %cond, i1 true, i1 %b
  ret i1 %a
}
$ clang-6.0 -O foo.ll -S -o - -emit-llvm
define i1 @main(i1 %cond, i1 %b)
local_unnamed_addr #0 {
  %a = or i1 %cond, %b
  ret i1 %a
}
$
```

```
while (…)
  if (cond
    foo
  } else {
    bar
  }
}
```

```
t = x + 1;
if (t == y
  w = x +
  foo(w);
}
```

You can't always get what you want

…itching)

…bal value

…nbering)

Takeaways so far…

- Deferred UB is useful but tricky
  - GCC and MSVC++ have concepts similar to undef and poison
  - … and similar problems
- Optimizations are math
  - Including UB!
  - Everyone writing optimizations needs to use the same math
  - And everyone needs to get the math right, or else we'll get miscompiles

- Let’s talk about pointers
  - In an assembly language there’s usually no difference between a pointer and a pointed-sized integer
  - In a safe high-level language like Java, pointers and integers are completely distinct types
- How about in C++?
  - C++ wants to optimize like a high-level language
  - But provide control like a low-level language
  - This isn’t so easy
  - We see similar concerns in LLVM IR

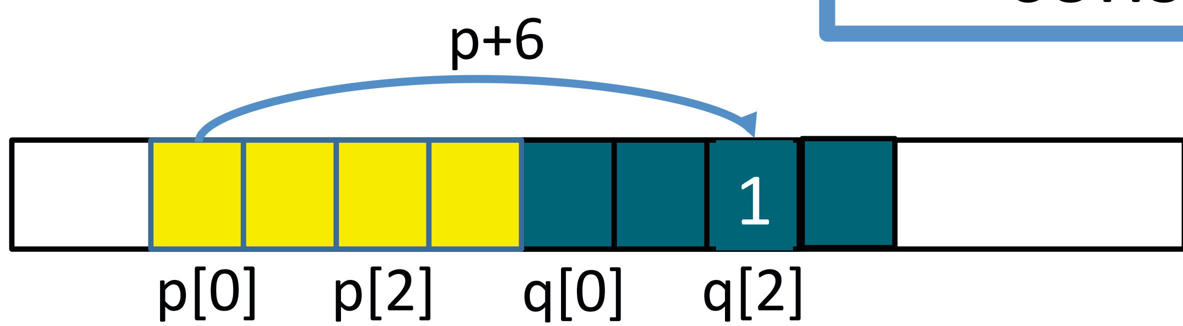
- We want a **memory model** that decides:
  - For each memory operation, is it UB?
  - For each load, what does it return?
  - (In this talk: Sequential code only)

```
char *p = malloc(4);
char *q = malloc(4);

q[2] = 0;       UB?
p[6] = 1;       UB?
print(q[2]);  prints what?
```

- The flat memory model

Pros and cons?




```
char *p = malloc(4);
char *q = malloc(4);

q[2] = 0;       not UB
p[6] = 1;       not UB
print(q[2]);    prints 1
```

- Dataflow-based provenance

Pros and cons?

p+6 is OOB

So far, so good!

asy to see
time ==
optimizable

```
q[2] = 0;        not
p[6] = 1;        UB
print(q[2]);     prints 0
```

- However, in C++ (and C, and LLVM) we can:
  – Convert a pointer to an integer
  – Do arbitrary integer math
  – Convert resulting integer to a pointer
  – Indirect through the pointer
- Memory model needs to account for this
  – Without breaking system-level codes
  – Without losing too many optimizations
    - Easy, unacceptable solution: flat memory model
  – We'll need two kinds of pointers…

- “Logical pointers” originate at allocations

```
char *p = malloc(4);
char *q = p + 2;
char *r = q - 1;
```

- Must remain within an allocated object or point one past the end
- Logical pointers use dataflow-based provenance
  - Logical pointers originating at different allocations never alias

- C doesn't get logical pointers quite right
  - From 6.5.9 p6 of C11:

Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[109]

- It's like a bit of the flat memory model leaked into the standard!
  - Compilers tend to ignore this when convenient
  - LLVM is more careful than GCC

- C++ gets this right

type (Clause 8). Comparing pointers is defined as follows:

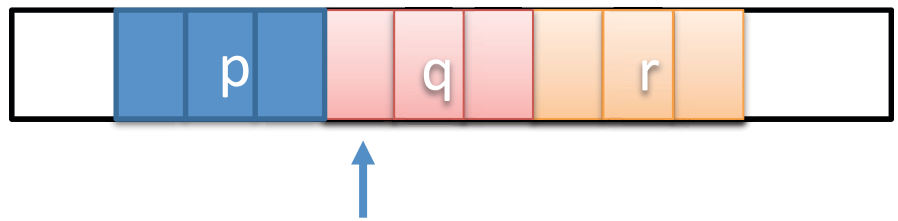
— If one pointer represents the address of a complete object, and another pointer represents the address one past the last element of a different complete object,[88] the result of the comparison is unspecified.

- This allows the compiler to stay in its logical pointer comfort zone

- “Physical pointers” originate at casts from integer to pointer

```
int x = ...;
char *p = (char *)x;
char *q = p + 2;
```

- Physical pointers use dataflow provenance and also control-flow-based provenance
  - Idea is that addresses cannot be predicted
  - They must be observed by the program at runtime




```c
char *p = malloc(3);
char *q = malloc(3);
char *r = malloc(3);
int x = (int)p + 3;
int y = (int)q;

if (x == y) {
   *(char*)x = 1; // OK
}


*(char*)x = 1; // UB
```

Observed address of p (data-flow)

Observed p+n == q (control-flow)

Can't access r, only p and q

Only p observed; p[3] is out-of-bounds

- Fun consequences of control-flow-based pointer provenance
  - Assume `p` is a logical pointer
  - `p` and `(int *)(int)p` are not necessarily the same!
    - LLVM wants to optimize `(int *)(int)p` to `p`
    - This is wrong! Provenance information can be lost
  - Our old friend GVN also causes trouble by replacing pointers that compare equal but have different provenance
  - Both LLVM and GCC can miscompile code like this

- It’s only valid to compare pointers with overlapping liveness
  - Potent … ges

```
char *p = mal…              malloc(4);
char *q = mal…

// valid                    malloc(4);
if (p == q) {

free(p);                    if (p == q) { ... }
```

But back to the big picture…

- Compiler engineers hate being wrong
- Users hate miscompiles

The fix:

1. Agree on undefined behavior semantics
2. Give compiler engineers tools that help them avoid being wrong
   - Clear documentation
   - UB-aware IR interpreter
   - Formal-methods-based tools

- “Alive” is a domain-specific language for writing LLVM peephole optimizations
- And a proof-based checking tool
  - Online: https://rise4fun.com/Alive

Is this optimization correct?

```
%x = select %cond, true, %b
  =>
%x = or %cond, %b
```

		Description
	1	Target is more poisonous than Source for i1 %x

```
----------------------------------------
Optimization: 1

ERROR: Target is more poisonous than Source for i1 %x

Example:
%cond i1 = 0x1 (1, -1)
%b i1 = poison
Source value: 0x1 (1, -1)
Target value: poison
```

Alive is opinionated!

Is this optimization correct?

```
%out = udiv i16 %in, 515
  =>
%tmp1 = zext %in to i32
%tmp2 = mul %tmp1, 65155
%tmp3 = lshr %tmp2, 25
%out = trunc %tmp3 to i16
```

```
--------------------------
Optimization: 1
Done: 1
Optimization is correc
```

Proved correct!
For all values of %in!
Using math!!

- LLVM is really good at doing these kinds of optimizations
  - But it takes >30,000 LOC
  - This code is very, very error-prone
- So why write this code by hand?
  - Isn't there a better way?
  - Well it's a small matter of research…

- Souper is a “superoptimizer” I work on
  – Takes some IR
  – Searches for a lower-cost refinement
  – Aggressively trips over problems in the UB model
- Optimizations derived by Souper are in
  – LLVM
  – Microsoft Mono and VC++
  – Binaryen (WebAssembly toolchain)
- Real goal is to replace big pieces of hand-written optimizers
  – Faster, more correct, and more effective compilers!
  – Several open problems to solve

Conclusions

- UB is fundamental in compiler IRs
  - UB is good for safe languages too!
- Understanding UB in the compiler is useful for non-compiler-hackers
- Dealing with UB is easier if…
  - Its meaning has been carefully thought out
  - We have tool support
- Future compilers will have a lot more automation

Thanks!