Undefined Behavior: Long Live Poison!

Nuno Lopes (Microsoft Research)
Gil Hur, Juneyoung Lee, Yoonseung Kim, Youngju Song (Seoul N. Univ.)
Sanjoy Das (Azul Systems), David Majnemer (Google), John Regehr (Univ. Utah)

Outline

1. Motivation for undef & poison
2. Why they are broken
3. Proposal to fix problems
4. Deployment scenario
5. Evaluation

Undef for SSA Construction

```
int x;
if (c)
  x = f();

if (c2)
  g(x);
```

```
entry:
  %x = alloca i32
  br %c, %ctrue, %cont

ctrue:
  %v = call @f()
  store %v, %x
  br %cont

cont:
  br %c2, %c2true, %exit

c2true:
  %v2 = load %x
  call @g(%v2)
```

```
entry:
  br %c, %ctrue, %cont

ctrue:
  %xf = call @f()
  br %cont

cont:
  %x = phi [ %xf, %ctrue ],
           [ undef, %entry ]
  br %c2, %c2true, %exit

c2true:
  call @g(%x)
```

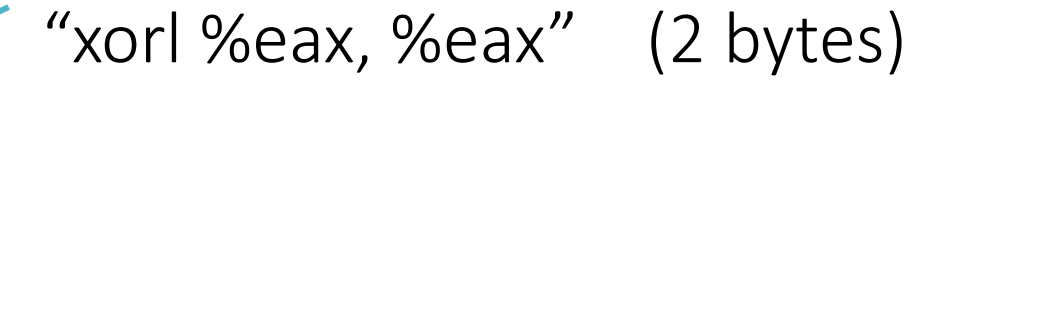
Undef for SSA Construction

```
entry:
  br %c, %ctrue, %cont

ctrue:
  %xf = call @f()
  br %cont

cont:
  %x = phi [ %xf, %ctrue ],
           [ undef, %entry]
  br %c2, %c2true, %exit

c2true:
  call @g(%x)
```



If 0 instead, LLVM produces extra
“xorl %eax, %eax” (2 bytes)

Undef is not enough!

```
a + b > a          =>          b > 0
```

```
%add = add nsw %a, %b
%cmp = icmp sgt %add, %a
```

=>

```
%cmp = icmp sgt %b, 0
```

```
%a = INT_MAX
%b = 1
```

```
%add = add nsw INT_MAX, 1          == undef
%cmp = icmp sgt undef, INT_MAX == false
```

```
%cmp = icmp sgt 1, 0
     == true
```

Different result: invalid optimization!

Undef is not enough #2

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

Mismatch between pointer and index types on x86-64

```
entry:
  br %head

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

body:
  %iext = sext %i to i64
  %ptr = getelementptr %a, %iext
  store 42, %ptr
  %i1 = add nsw %i, 1
  br %head
```

Hoisting sext gives 39% speedup on my desktop!

Undef is not enough #2

```
entry:
  br %head

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

body:
  %iext = sext %i to i64
  %ptr = getelementptr %a, %iext
  store 42, %ptr
  %i1 = add nsw %i, 1
  br %head
```

i + 1 + … + 1 <= n

On overflow:

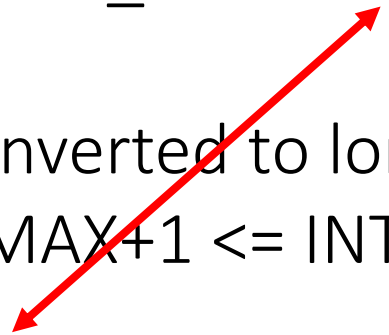
undef <= n

If n = INT_MAX: **true**

If i converted to long:

INT_MAX+1 <= INT_MAX:

false!



Different result: invalid optimization!

Nsw cannot be UB!

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

We want to hoist x + 1

```
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
```

Motivation: Summary

Undef: SSA construction, padding, …

Poison: algebraic simplifications, widening of induction variables, …

UB: instructions that trap the CPU (division by zero, load from null ptr, …)

Problems with Undef & Poison

Duplicate SSA uses

Rewrite expression to remove multiplication:
2 * x -> x + x

If x = undef:

2 * undef -> undef + undef **== undef**

Before: even number
After: any number

Transformation is not valid!

Hoist past Control-Flow

```
if (k != 0) {
  while (...) {
    use(1 / k);
  }
}
```

k != 0, so safe to hoist division?

```
if (k != 0) {
  int tmp = 1 / k;
  while (...) {
    use(tmp);
  }
}
```

If k = undef

"k != 0" may be true **and**

"1 / k" trigger UB

Mixing Poison & Undef

```
%v = select %c, %x, undef
  =>
%v = %x
```

Wrong if %x is poison!

GVN vs Loop Unswitching

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```

```
if (c2) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

Loop unswitch

Branch on poison/undef **cannot** be UB

Otherwise, wrong if loop never executed

GVN vs Loop Unswitching

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

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

Contradiction with loop unswitching!

GVN

Branch on poison/undef **must** be UB

Otherwise, wrong if **y** poison but not **x**

LLVM IR: Summary

Current definition of undef (different value per use) breaks many things

There’s no way to use both GVN and loop unswitching!

Poison and undef don’t play well together

Proposal

Proposal

Remove undef

Replace uses of undef with poison (and introduce poison value in IR)

New instruction: “`%y = freeze %x`” (stops propagation of poison)

All instructions over poison return poison (except phi, freeze, select)

`br poison` -> UB

Poison

```
and %x, poison   ->  poison  ; just like before
and 0, poison    ->  poison  ; just like before
```

```
%y = freeze poison
%z = and %y, 1      ; 000..0x (like old undef)
%w = xor %y, %y     ; 0 -- not undef: all uses of %y get same val
```

Fixing Loop Unswitch

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```

```
if (freeze(c2)) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

GVN doesn’t need any change!

Freeze: avoid UB

```
%0 = udiv %a, %x
%1 = udiv %a, %y
%s = select %c, %0, %1
```

```
%c2 = freeze %c
%d  = select %c2, %x, %y
%s  = udiv %a, %d
```

Bit fields

```
a.x = foo;


%val  = load %a
%val2 = freeze %val  ; %val could be uninitialized (poison)
%foo2 = freeze %foo
%val3 = ... combine %val2 and %foo2 ...
store %val3, %a
```

Bit fields #2

```
a.x = foo;
```

```
%val  = load <32 x i1>, %a
%val2 = insertelement %foo, %val, ...
store %val2, %a
```

+ No freeze
+ Perfect store-forwarding
- Many insertelements

  Back to lower bit fields with structs?

Load Widening

```
%v = load i16, %ptr
```

Cannot widen to “`load i32, %ptr`”
If following bits may be uninitialized/poison

Safe:

```
%tmp = load <2 x i16>, %ptr
%v = extractelement %tmp, 0
```

Deployment

Deployment Plan

1) Add freeze instruction + CodeGen support
2) Change clang to start emitting freeze for bit-field stores
3) Add auto-upgrade
4) Fix InstCombine, Loop unswitching, etc to use freeze
5) Replace references to undef in the code with poison or “freeze poison”
7) Kill undef
8) Investigate remaining perf regressions
9) Run LLVM IR fuzzer with Alive to find leftover bugs

Auto Upgrade IR

```
%x = add %y, undef
  =>
%u = freeze poison
%x = add %y, %u
```

(undef is equivalent to freeze with 1 use)

```
%x = load i32, %ptr
  =>
%ptr2 = bitcast %ptr to <32 x i1>*
%t    = load <32 x i1>, %ptr2
%t2   = freeze %t
%x    = bitcast %t2 to i32
```

CodeGen

Do we want poison at SDAG/MI levels?

How to better lower “freeze poison”?

Evaluation

Evaluation

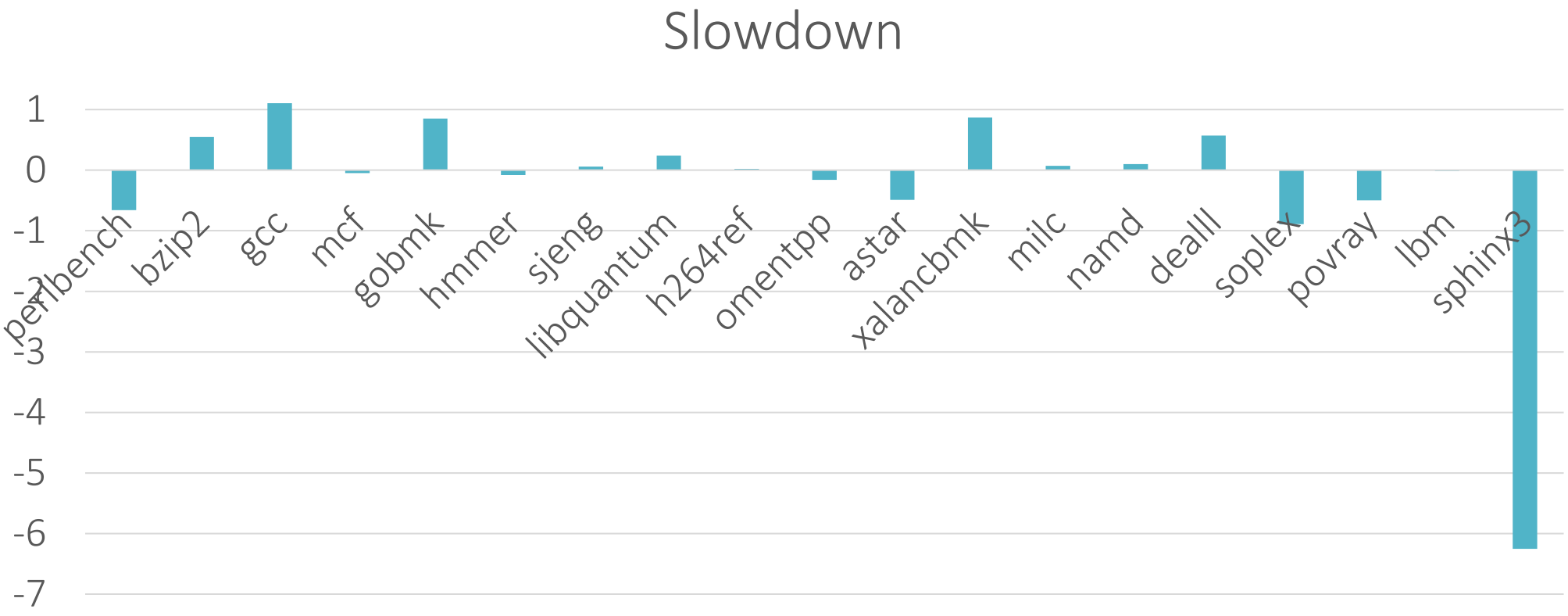
Prototype implementation:

- Add freeze in loop unswitch
- Make clang emit freeze for bitfields
- A few InstCombine fixes
- SelDag: “freeze poison” -> CopyFromReg + CopyToReg

Compare:

- -O3 vs -O3 w/ freeze
- SPEC 2k6, LNT, single-file programs
- Compile time, running time, memory consumption, IR size

SPEC 2k6 running time



Lower is better

Range: -6.2% to 1.1%

LNT

Running time: overall 0.18% slowdown

A few big regressions (Dhrystone, SPASS, Shootout) due to unrolling

Compile time: unchanged

Single-file programs

(bzip2, gcc, gzip, oggenc, sqlite3)

Compile time: 0.9% regression on gcc

Memory consumption: 1% increase on gcc

IR: gcc increase 5% in # instructions (2.4% freeze in total)

Others: 0.1-0.2% freeze

Static Analyses

`IsNonZero(d)`: safe to hoist division?

```
while(c) {
  x = 1 / d;
}
```

What if `d` is poison?

Should analyses take poison into account or return list of values that must be non-poison?

(only relevant for optimizations that hoist instructions past control-flow)

Conclusion

LLVM IR needs improvement to fix miscompilations

We propose killing undef and empower poison

Early results from prototype show few regressions

Call for Action:

- Comment on the ML; Vote!
- Review design for CodeGen, SelDag, MI, big endian, …
- Volunteer to review patches, fix regressions, …

Select

1) Select should be equivalent to arithmetic:
   "`select %c, true, %x`" -> "`or %c, %x`"
   arithmetic -> select
2) `br` + `phi` -> `select` should be allowed (SimplifyCFG)
3) `select` -> `br` + `phi` should be allowed (when cmov is expensive)

We propose to make "`select %c, %a, %b`" poison if any of the following holds:

- `%c` is poison
- `%c = true` and `%a` is poison
- `%c = false` and `%b` is poison

Poison: bitcasts

```
%x = bitcast <3 x i2>  <2, poison, 2> to <2 x i3>
  =>
%x = <poison, poison>
```

```
%x = bitcast <6 x i2>  <2, poison, 2, 2, 2, 2> to <4 x i3>
  =>
%x = <poison, poison, 5, 2>
```