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
int x;
if (c)
  x = f();
if (c2)
  g(x);

entry:
  %x = alloca i32
  br %c, %ctru, %cont

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

cont:
  br %c2, %c2true, %exit

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

entry:
  br %c, %ctru, %cont

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

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

c2true:
  call @g(%x)
Undef for SSA Construction

```assembly
entry:
  br %c, %ct, %cont

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

cont:
  %x = phi [ %xf, %ct ],
      [ undef, %entry]
  br %c2, %ct2, %exit

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

If 0 instead, LLVM produces extra “xorl %eax, %eax” (2 bytes)
Undef is not enough!

\[ a + b > a \implies b > 0 \]

\[
\%\text{add} = \text{add nsw} \ %a, %b \\
\%\text{cmp} = \text{icmp sgt} \ %\text{add}, %a
\]

\[
\%a = \text{INT\_MAX} \\
%b = 1
\]

\[
\%\text{add} = \text{add nsw} \ \text{INT\_MAX}, 1 \implies \text{undef} \\
\%\text{cmp} = \text{icmp sgt} \text{ 1, 0} \implies \text{false}
\]

Different result: invalid optimization!
for (int i = 0; i <= n; ++i) {
    a[i] = 42;
}

Mismatch between pointer and index types on x86-64

entry:
    br %head

code:
    %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!

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

We want to hoist $x + 1$
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!
if (k != 0) {
    while (...) {
        use(1 / k);
    }
}

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

k != 0, so safe to hoist division?

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);
}

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: \( \texttt{\%y} = \texttt{freeze} \%x \) (stops propagation of poison)
All instructions over poison return poison (except phi, freeze, select)
\texttt{br poison} \rightarrow \text{UB}
Poison

\[
\text{and } \%x, \text{poison} \rightarrow \text{poison} \quad ; \text{just like before}
\]
\[
\text{and } 0, \text{poison} \rightarrow \text{poison} \quad ; \text{just like before}
\]

\[
\%y = \text{freeze} \quad \text{poison}
\]
\[
\%z = \text{and} \quad \%y, 1 \quad ; \text{000..0x (like old undef)}
\]
\[
\%w = \text{xor} \quad \%y, \%y \quad ; 0 \quad \text{-- not undef: all uses of } \%y \text{ get same val}
\]
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
%c2 = freeze %c
%1 = udiv %a, %y
%d = select %c2, %x, %y
%s = select %c, %0, %1
%s = udiv %a, %d
Bit fields

\[ a.x = \text{foo}; \]

\[
\begin{align*}
\%\text{val} &= \text{load} \ %a \\
\%\text{val2} &= \text{freeze} \ %\text{val} \ ; \ %\text{val} \ \text{could be uninitialized (poison)} \\
\%\text{foo2} &= \text{freeze} \ %\text{foo} \\
\%\text{val3} &= \ldots \ \text{combine} \ %\text{val2} \ \text{and} \ %\text{foo2} \ldots \\
\text{store} \ %\text{val3}, \ %a
\end{align*}
\]
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

\[ %v = \text{load } \text{i16}, \text{ptr} \]

Cannot widen to \text{“load } \text{i32}, \text{ptr}”
If following bits may be uninitialized/poison

Safe:
\[ %\text{tmp} = \text{load } \text{<2 x i16>}, \text{ptr} \]
\[ %v = \text{extractelement } %\text{tmp}, 0 \]
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”
6) Kill undef
7) Investigate remaining perf regressions
8) Run LLVM IR fuzzer with Alive to find leftover bugs
Auto Upgrade IR

\%x = \text{add} \ \%y, \ \text{undef} \\
=> \\
\%u = \text{freeze poison} \\
\%x = \text{add} \ \%y, \ \%u

(undef is equivalent to freeze with 1 use)

\%x = \text{load i32, } \%\text{ptr} \\
=> \\
\%\text{ptr2} = \text{bitcast } \%\text{ptr to } <32 \times \text{i1}>* \\
\%t = \text{load } <32 \times \text{i1}>, \%\text{ptr2} \\
\%t2 = \text{freeze } \%t \\
\%x = \text{bitcast } \%t2 \text{ 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

Slowdown

Lower is better
Range: -6.2% to 1.1%
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

\texttt{IsNonZero}(d): safe to hoist division?

\begin{verbatim}
while(c) {
    x = 1 / d;
}
\end{verbatim}

What if \texttt{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) \texttt{br + phi} -> \texttt{select} should be allowed (SimplifyCFG)
3) \texttt{select} -> \texttt{br + phi} should be allowed (when cmov is expensive)

We propose to make "\texttt{select %c, %a, %b}" poison if any of the following holds:
   - %c is poison
   - %c = \texttt{true} and %a is poison
   - %c = \texttt{false} and %b is poison
Poison: bitcasts

```c
%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>
```