Taming Undefined Behavior in LLVM

Seoul National Univ.

Azul Systems

Google

University of Utah

Microsoft Research

Juneyoung Lee
Yoonseung Kim
Youngju Song
Chung-Kil Hur

Sanjoy Das

David Majnemer

John Regehr

Nuno P. Lopes
What this talk is about

• A compiler IR (Intermediate Representation) can be designed to allow more optimizations by supporting “undefined behaviors (UBs)”

• LLVM IR’s UB model
  - Complicated
  - Invalidates some textbook optimizations

• Our new UB model
  - Simpler
  - Can validate textbook optimizations (and more)
Undefined Behavior (UB) & Problems
Motivation for UB

Peepholes Optimization

\begin{align*}
\text{int}^* & \; p \\
\text{int} & \; a \\
\text{int} & \; b \\
\end{align*}

\textbf{IR} \quad \text{output}(p + a > p + b) \quad \rightarrow \quad \textbf{IR} \quad \text{output}(a > b)
Peephole Optimization

Motivation for UB

```
int* p
int a
int b
```
Motivation for UB

Peephole Optimization

int* p
int a
int b

```
output(p + a > p + b)
```

IR: 0x0

0x0 (Overflow!)

```
output(a > b)
```

IR: 0x100 0

0x100 0

0xFFFFFF00 4
Motivation for UB

Peephole Optimization

```
int* p
int a
int b
```

```
output(p + a > p + b)
```

```
output(a > b)
```

(Overflow!)
Motivation for UB

Peephole Optimization

```
int* p
int a
int b
```

**Output:**

\[
p + a > p + b
\]

**IR:**

```
output(p + a > p + b)
```

**Output:**

\[
a > b
\]

**IR:**

```
output(a > b)
```

**0x0**

(Overflow!)

**0x100**

**false**

```
output(0x0)
```

**0x100**

**true**

```
output(0x100)
```
Simple UB Model:
Pointer Arithmetic Overflow is Undefined Behavior

Motivation for UB
Loop Invariant Code Motion

Simple UB Model:
Pointer Arithmetic Overflow is Undefined Behavior

IR

... for(i=0; i<n; ++i) {
    a[i] = p + 0x100
}

q = p + 0x100
for(i=0; i<n; ++i) {
    a[i] = q
}
Loop Invariant Code Motion

Simple UB Model:
Pointer Arithmetic Overflow is Undefined Behavior
Loop Invariant Code Motion

Simple UB Model:
Pointer Arithmetic Overflow is Undefined Behavior

Problems with UB

IR
... for(i=0; i<n; ++i)
{
    a[i] = p + 0x100
}
0xFFFFF00

IR
q = p + 0x100
for(i=0; i<n; ++i)
{
    a[i] = q
}

0xFFFFF00
Overflow!
Loop Invariant Code Motion

Simple UB Model:
Pointer Arithmetic Overflow is Undefined Behavior

Problems with UB

IR

... for(i=0; i<n; ++i) {
  a[i] = p + 0x100
}

0xxFFFFFFFF00

UB

Overflow!

IR

q = p + 0x100

for(i=0; i<n; ++i) {
  a[i] = q
}

0xxFFFFFFFF00

0
Existing Approaches
Poison Value: A Deferred UB

Simple UB Model:
Pointer Arithmetic Overflow is Undefined Behavior

```
... 
for (i=0; i<n; ++i)
{
    a[i] = p + 0x100
}
```

```
q = p + 0x100
for (i=0; i<n; ++i)
{
    a[i] = q
}
```
Poison Value: A Deferred UB

LLVM’s UB Model:
Pointer Arithmetic Overflow is
A Poison “Value”
Poison Value: A Deferred UB

LLVM’s UB Model:
Pointer Arithmetic Overflow is
A Poison “Value”

... for(i=0; i<n; ++i)
{
    a[i] = p + 0x100
}

q = p + 0x100
for(i=0; i<n; ++i)
{
    a[i] = q
}
Poison Value: A Deferred UB

LLVM’s UB Model:
Pointer Arithmetic Overflow is A Poison “Value”

IR
output(p + a > p + b)

0x100
0x0
(Overflow!)

UB
0xFFFFFFF00

IR
output(a > b)

0x100
0
Poison Value: A Deferred UB

LLVM’s UB Model:
Pointer Arithmetic Overflow is
A Poison “Value”

IR
output(p + a > p + b)

Poison
0x100
0

IR
output(a > b)

0x100
0
Poison Value: A Deferred UB

LLVM’s UB Model:
Pointer Arithmetic Overflow is
A Poison “Value”
Poison Value: A Deferred UB

LLVM’s UB Model:
Pointer Arithmetic Overflow is
A Poison “Value”
Summary of Poison

\[
\begin{align*}
\text{p} & \rightarrow + \rightarrow > \rightarrow \text{output} \\
0xFF000 & \rightarrow + \\
0x100 & \rightarrow + \\
\text{a} & \rightarrow + \\
\text{b} & \rightarrow +
\end{align*}
\]
Summary of Poison

\[
\begin{align*}
0x\text{FFFFFFF00} & \quad 0x100 \\
& \quad p \\
& \quad a \\
& \quad p \\
& \quad b \\
\text{poison} & \quad + \\
& \quad > \\
& \quad \text{output}
\end{align*}
\]
Summary of Poison

\[ \text{poison} \rightarrow + \rightarrow > \rightarrow \text{output} \]

\[ 0x\text{FFFFFFF00} \quad 0x100 \]

\[ p \quad a \quad p \quad b \]

Propagate
Summary of Poison

Propagate

Raise UB

UB

output

poison

poison

0xFFFFFFF00

0x100

p

a

p

b

>
Summary of Poison

Propagate

Raise UB

"Poison is Sometimes Too Poisonous"
Problems with LLVM’s UB

Global Value Numbering (GVN)

LLVM’s UB Model:
Branching on poison is
???

if (x == y) {
    .. use x..
}

if (x == y) {
    .. use y..
}
Problems with LLVM’s UB

Global Value Numbering (GVN)

LLVM’s UB Model:
Branching on poison is ???

if (x == y) {
    .. use x ..
}

if (x == y) {
    .. use y ..
}
Problems with LLVM’s UB

Global Value Numbering (GVN)

LLVM’s UB Model:
Branching on poison is

```c
if (x == y) {
  .. use x ..
}
```

```c
if (x == y) {
  .. use y ..
}
```
Problems with LLVM’s UB

Global Value Numbering (GVN)

LLVM’s UB Model:
Branching on poison is ???

if (x == y) {
  .. use x ..
}

if (x == y) {
  .. use y ..
}
Problems with LLVM’s UB

Global Value Numbering (GVN)

LLVM’s UB Model:
Branching on poison is
???

if (x == y) {
.. use x ..
}

if (x == y) {
.. use y ..
}
Problems with LLVM’s UB

Global Value Numbering (GVN)

LLVM’s UB Model:
Branching on poison is
???
Problems with LLVM’s UB

Global Value Numbering (GVN)

LLVM’s UB Model:
Branching on poison is
Undefined Behavior

```
if (x == y) {
  .. use x ..
}
```

```
if (x == y) {
  .. use y ..
}
```
Problems with LLVM’s UB

Global Value Numbering (GVN)

LLVM’s UB Model:
Branching on poison is Undefined Behavior

if (x == y) {
  .. use x ..
}
Problems with LLVM’s UB

Loop Unswitching (LU)

LLVM’s UB Model:
Branching on poison is Undefined Behavior

while (n > 0) {
    if (cond)
        A
    else
        B
}

if (cond)
    while (n > 0)
        { A }
else
    while (n > 0)
        { B }
Problems with LLVM’s UB

Loop Unswitching (LU)

LLVM’s UB Model:
Branching on poison is Undefined Behavior

while (n > 0) {
  if (cond)
    A
  else
    poison
}

if (cond)
  while (n > 0)
    { A }
else
  while (n > 0)
    { B }

poison
Problems with LLVM’s UB
Loop Unswitching (LU)

LLVM’s UB Model:
Branching on poison is Undefined Behavior

while (n > 0) {
  if (cond) 
    A
  else
    poison
}

if (cond)
  while (n > 0)
    { A }
else
  while (n > 0)
    { B }

UB
poison

0
Inconsistency in LLVM

• GVN + LU is inconsistent.

• We found a miscompilation bug in LLVM due to the inconsistency (LLVM Bugzilla 31652).
  - It is being discussed in the community
  - No solution has been found yet
Our Approach
Overview

Existing Approaches

- Complex
- Inconsistent
- Poison values
  - Undef. values
  - Defined values
- Can't Control Poison
- More Defined
- GVN + LU
Overview

Existing Approaches
- Complex
- Inconsistent
- GVN + LU

Our Approach
- Simpler

More Defined

UB
- Poison values
- Undef. values
- Defined values

Can’t Control Poison

UB
- Poison values
- Defined values

freeze
Overview

Existing Approaches
- Complex
- Inconsistent
- GVN + LU

Our Approach
- Simpler
- UB

More Defined
- Poison values
- Undef. values
- Defined values

Can’t Control Poison

Can Control Poison

freeze
Overview

Existing Approaches
- Complex
- Inconsistent

Our Approach
- Simpler
- Consistent

- GVN + LU

Defined values
- Poison values
- Undef. values
- Defined values

Poison values
- Can’t Control Poison
- Can Control Poison

Defined values
- freeze

More Defined

Defined
Key Idea: “Freeze”

- Introduce a new instruction

\[ y = \text{freeze} \ x \]

- Semantics:

  When \( x \) is a \text{defined} value:
  \[ \text{freeze} \ x \rightarrow x \]

  When \( x \) is a \text{poison} value:
  \[ \text{freeze} \ x \rightarrow 0, 1, 2, \ldots \]

  \text{Nondet. Choice of A Defined Value}
Our Solution

Loop Unswtiching

Our UB Model:
Branching on poison is
Undefined Behavior
Our Solution

Loop Unswitching

Our UB Model:
Branching on poison is
Undefined Behavior

while (n > 0) {
    if (cond)
        A
    else
        B
}

if (freeze(cond))
    while (n > 0)
        { A }
else
    while (n > 0)
        { B }
Our Solution
Loop Unswitching

Our UB Model:
Branching on poison is
Undefined Behavior

Our Solution

Loop Unswitching

```
while (n > 0) {
    if (cond)
        A
    else
        B
}
```

```
if (freeze(cond))
    while (n > 0)
        { A }
else
    while (n > 0)
        { B }
```
Our UB Model:
Branching on poison is Undefined Behavior

Our Solution
Loop Unswitching

while (n > 0) {
    if (cond)
        A
    else
        B
}

if (freeze(cond))
    while (n > 0)
        { A }
else
    while (n > 0)
        { B }
Summary of Freeze

- Branching on `freeze(poison)` => Nondet.
  - Used for Loop Unswitching
- Branching on `poison` => UB
  - Used for Global Value Numbering

Compilers can control poison!
Summary of Freeze

Compilers can control poison!

- Branching on `freeze(poison)` => Nondet.
  - Used for Loop Unswitching

- Branching on `poison` => UB
  - Used for Global Value Numbering

Freeze can also fix many other UB-related problems.
Further Example

Hoisting Division

// bitwise-or
k = x | 0x1

while (n > 0)
    use(100 / k)

// bitwise-or
k = x | 0x1
t = 100 / k
while (n > 0)
    use(t)
Further Example

Hoisting Division

\[
k = x | 0x1
\]

while \((n > 0)\)

\[
\text{use}(100 / k)
\]

\[
k = x | 0x1
\]

\[
t = 100 / k
\]

while \((n > 0)\)

\[
\text{use}(t)
\]
Further Example

Hoisting Division

\[ k = x \mid 0x1 \]
\[ \text{while } (n > 0) \]
\[ \text{use(}100 \div k\text{)} \]

\[ k = x \mid 0x1 \]
\[ t = 100 \div k \]
\[ \text{while } (n > 0) \]
\[ \text{use}(t) \]
Further Example

Hoisting Division

```plaintext
k = x | 0x1
while (n > 0)
  use(100 / k)
```

```plaintext
k = x | 0x1
while (n > 0)
  t = 100 / k
  use(t)
```
Further Example

Hoisting Division

\[ k = x \mid 0x1 \]

while \((n > 0)\)

use\((100 / k)\)

\[ t = 100 / k \]

while \((n > 0)\)

use\((t)\)
LLVM does not currently support it.

Further Example

Hoisting Division

poison poison

\[
k = x \mid 0x1
\]

while (n > 0)

use(100 / k)

poison poison

\[
k = x \mid 0x1
\]

\[
t = 100 / k
\]

while (n > 0)

use(t)

UB
Further Example

Hoisting Division

LLVM does not currently support it.

```
k = x | 0x1
while (n > 0)
  use(100 / k)
```
LLVM does not currently support it.

Further Example
Hoisting Division

\[ k = x | 0x1 \]
\[
\text{while (n > 0)}
\]
\[
\text{use}(100 / k)
\]

\[ k = \text{freeze}(x) | 0x1 \]
\[
\text{t} = 100 / k
\]
\[
\text{while (n > 0)}
\]
\[
\text{use}(t)
\]
LLVM does not currently support it.

Further Example

Hoisting Division

```
k = x | 0x1
while (n > 0)
    use(100 / k)
```

```
k = freeze(x) | 0x1
while (n > 0)
    t = 100 / k
    use(t)
```
LLVM does not currently support it.

Further Example

Hoisting Division

k = x | 0x1
while (n > 0)
use(100 / k)

k = freeze(x) | 0x1
while (n > 0)
use(t)
Further Example

Hoisting Division

Freeze can make LLVM support it!

\[
k = x | 0x1
\]

\[
\text{while } (n > 0)
\]

\[
\text{use}(100 / k)
\]

A defined value

\[
k = \text{freeze}(x) | 0x1
\]

\[
t = 100 / k
\]

\[
\text{while } (n > 0)
\]

\[
\text{use}(t)
\]
Implementation

• Target: LLVM 4.0 RC 4 (Mar. 2017)
• Add Freeze instruction to LLVM IR
• Bug Fixes Using Freeze
  - Loop Unswitching Optimization
  - C Bitfield Translation to LLVM IR
  - InstCombine Optimizations

* More details are given in the paper
Experiment Results

• Benchmarks (4.6M LOC):
  - SPEC CPU2006
  - LLVM Nightly Test
  - Large Single File Benchmarks

• Compilation Time: ± 1%

• Compilation Memory Usage: Max + 2%

• Generated Code Size: ± 0.5%

• Execution Time: ± 3%

* More details are given in the paper
“Freeze” Can Fix UB Semantics Without Significant Performance Penalty

- Benchmarks (4.6M LOC):
  - SPEC CPU2006
  - LLVM Nightly Test
  - Large Single File Benchmarks

- Compilation Time: ± 1%
- Compilation Memory Usage: Max + 2%
- Generated Code Size: ± 0.5%
- Execution Time: ± 3%

* More details are given in the paper
Conclusion

• Modern compilers’ UB models cannot support some textbook optimizations.

• We propose “freeze” to fix such problems.

• Freeze has little impact on performance.