Safely Optimizing Casts between Pointers and Integers

Seoul National Univ.

MPI-SWS

University of Utah

Microsoft Research

Juneyoung Lee
Chung-Kil Hur

Ralf Jung
Zhengyang Liu
John Regehr

Nuno P. Lopes
## Overview

<table>
<thead>
<tr>
<th>Assembly (x86-64, ARM, ..)</th>
<th>LLVM IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer [0, 2^{64})</td>
<td>[0, 2^{64}) + provenance</td>
</tr>
<tr>
<td>Integer [0, 2^{64})</td>
<td>[0, 2^{64}) + ?</td>
</tr>
</tbody>
</table>
Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code for readability

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

* [https://godbolt.org/z/9eNt6w](https://godbolt.org/z/9eNt6w)
Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code for readability

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

* [https://godbolt.org/z/9eNt6w](https://godbolt.org/z/9eNt6w)
Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code for readability

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

* https://godbolt.org/z/9eNt6w
**Problem: Pointer as a Pure Integer**

We use C syntax for LLVM IR code for readability

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

* [https://godbolt.org/z/9eNt6w](https://godbolt.org/z/9eNt6w)*
Problem: Pointer as a Pure Integer

We use C syntax for LLVM IR code for readability

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

* https://godbolt.org/z/9eNt6w
Problem: Pointer as a Pure Integer

Memory: 0x0

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

constant prop.

* [https://godbolt.org/z/9eNt6w](https://godbolt.org/z/9eNt6w)
Problem: Pointer as a Pure Integer

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```

[Diagram showing memory allocation and pointer comparison]

* https://godbolt.org/z/9eNt6w
Problem: Pointer as a Pure Integer

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

Memory:
```
0x0 0x100 0x101
```

* [https://godbolt.org/z/9eNt6w](https://godbolt.org/z/9eNt6w)
Problem: Pointer as a Pure Integer

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

---

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```

---

* https://godbolt.org/z/9eNt6w*
Problem: Pointer as a Pure Integer

constant prop.

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

Memory:

0x0 0x100 0x101
-----------------
0x100 0x101

0x100 0x101

0x100 0x101

* https://godbolt.org/z/9eNt6w
Problem: Pointer as a Pure Integer

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

Memory:

```
0x0  0x100  0x101
```

* https://godbolt.org/z/9eNt6w
Problem: Pointer as a Pure Integer

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

Memory:

```
0x0 0x100 - 10 0x101
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```

* [https://godbolt.org/z/9eNt6w](https://godbolt.org/z/9eNt6w)
Problem: Pointer as a Pure Integer

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```

* https://godbolt.org/z/9eNt6w
Problem: Pointer as a Pure Integer

Problem with “pointer as a pure integer”

Cannot protect accesses from different blocks

* https://godbolt.org/z/9eNt6w
LLVM’s Solution: Pointers have Provenance

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

Constant propagation:

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

* [https://godbolt.org/z/9eNt6w](https://godbolt.org/z/9eNt6w)
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

LLVM’s Solution:
Pointers have Provenance

Constant propagation:

* https://godbolt.org/z/9eNt6w
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
LLVM’s Solution: Pointers have Provenance

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

Provenance

```
(constant prop.) p: - (q, 0x101)
0x0 0x100 0x101
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```
LLVM’s Solution: Pointers have Provenance

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

constant prop.

https://godbolt.org/z/9eNt6w
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

LLVM’s Solution:
Pointers have Provenance

Memory:
(p,0x100) (q,0x101)

Provenance

undefined behavior because p ≠ q

* https://godbolt.org/z/9eNt6w
What about Integers?

<table>
<thead>
<tr>
<th>Assembly (x86-64, ARM, ..)</th>
<th>LLVM IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer</td>
<td>[0, $2^{64}$)</td>
</tr>
<tr>
<td>Integer</td>
<td>[0, $2^{64}$)</td>
</tr>
</tbody>
</table>
Miscompilation with PtrIntCast

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```
Miscompilation with PtrToIntCast

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```
Miscompilation with PtrIntCast

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```

**Cast elimination.**

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

**Constant propagation.**

```
int eq. prop.
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

**Integer equivalence property.**

```
const prop.
```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*) (int)(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*) iq = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
Miscompilation with PtrIntCast

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```
Miscompilation with PtrIntCast

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
Miscompilation with PtrIntCast

char p[1],q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)ip = 10;
    print(q[0]);
}

cast
elem.

constant
prop.

int. eq.
prop.
Miscompilation with PtrIntCast

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

int eq.
prop.

cast elim.

constant prop.

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

Miscompilation with PtrIntCast
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
Miscompilation with PtrIntCast

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

\[\text{Miscompilation with PtrIntCast}\]
We found this miscompilation bug in both LLVM & GCC
Which pass is responsible for it?
Problem depends on the model

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```

```
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)ip = 10;
    print(q[0]);
}
```
char p[1],q[1]={'\0'};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)ip = 10;
    print(q[0]);
}
```

```
char p[1],q[1]={'\0'};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)ip = 10;
    print(q[0]);
}
```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*) (int)(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*) (int)(p+1) = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)ip = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)ip = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

Integer Without Provenance Model

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
Integer-With-Provenance is Unnatural

• Hard to explain integer equality propagation
• Hard to explain many other transformations as well

\[
\begin{align*}
r &= (i + j) - k \\
r &= \text{(int)}(\text{float})j \\
r &= i + (j - k) \\
r &= j
\end{align*}
\]
Integer-With-Provenance is Unnatural

- Hard to explain integer equality propagation
- Hard to explain many other transformations as well

\[
\begin{align*}
\text{r} &= (i + j) - k \\
\text{r} &= \text{(int)}(\text{float})j \\
\text{r} &= i + (j - k) \\
\text{r} &= \text{proven} + \text{proven}?
\end{align*}
\]
Integer-With-Provenance is Unnatural

- Hard to explain integer equality propagation
- Hard to explain many other transformations as well

\[
\begin{align*} 
  r &= (i + j) - k \\
  r &= (\text{int})(\text{float})j \\
  r &= j \\
\end{align*}
\]
Our Suggestion [OOPSLA’18]: Integer-Without-Provenance Model

<table>
<thead>
<tr>
<th>Assembly (x86-64, ARM, ..)</th>
<th>LLVM IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer [0, 2^{64})</td>
<td>[0, 2^{64}) + provenance</td>
</tr>
<tr>
<td>Integer [0, 2^{64})</td>
<td>[0, 2^{64})</td>
</tr>
</tbody>
</table>
Integer-Without-Provenance Model

- Semantics of Casts
- Problematic Optimizations
- How to Recover Performance?
Semantics of Casts [OOPSLA’18]

1. Pointer-to-integer casts remove provenance
2. Integer-to-pointer casts gain full provenance

How to regain protection from unknown accesses?
By exploiting nondeterministic allocation

How to perform in-bounds checking on full-provenance pointers?
By recording in-bounds offsets at the pointer & checking when dereferenced
Optimizations Unsound in Our Model

1. Cast Elimination

\[
p_2 = (\text{char}*)(\text{int})p \quad \Rightarrow \quad p_2 = p
\]

2. Integer Comparison to Pointer Comparison

\[
c = \text{icmp\ eq\ (int)}p,\ (\text{int})q \quad \Rightarrow \quad c = \text{icmp\ eq\ p,\ q}
\]
Optimizations Unsound in Our Model

1. Cast Elimination

\[ p_2 = (\text{char}*)(\text{int})p \quad \Rightarrow \quad p_2 = p \]

Full provenance

2. Integer Comparison to Pointer Comparison

\[ c = \text{icmp \ eq \ (int)p, (int)q} \quad \Rightarrow \quad c = \text{icmp \ eq \ p, q} \]
Optimizations Unsound in Our Model

1. Cast Elimination

\[
p2 = \text{char}^*(\text{int})p \quad \rightarrow \quad p2 = p
\]

- Full provenance
- Provenance \( p \)

2. Integer Comparison to Pointer Comparison

\[
c = \text{icmp} \text{ eq } \text{(int)}p, \text{(int)}q \quad \rightarrow \quad c = \text{icmp} \text{ eq } p, q
\]

- Comparison of integers
- Comparison of pointers
Performance Issue

• Cast elimination removes significant portion of casts
  • 13% of ptrtoints, 40% of inttoptrs from C/C++ benchmarks *

• Disabling cast elimination hinders other optimizations
  • ptrtoint makes variables escaped
  • inttoptr is regarded as pointing to an unknown object

• Disabling cast elimination causes slowdown
  • 1% slowdown in perlbench_r, blender_r

* SPEC2017rate + LLVM test-suite, -O3
Our Solution

1. Do not generate \text{Ptr} \leftrightarrow \text{Int} casts in the first place
   • 86\% of \text{Ptr} \leftrightarrow \text{Int} casts are introduced by LLVM, not by programmers
     - \text{Ptr} \rightarrow \text{Int} casts are generated from pointer subtractions
     - \text{Int} \rightarrow \text{Ptr} casts are from canonicalizing loads/stores as int types
   • How: by introducing new features

2. Allow the previous optimizations conditionally
   • How: by developing an analyzer to check such conditions
To reduce $\text{Ptr} \rightarrow \text{Int}$ Casts: Introduce Pointer Subtraction Operation

Before Fix (Uses `ptrtoint`)

\[
\begin{align*}
\text{ip} &= \text{ptrtoint } p \\
\text{iq} &= \text{ptrtoint } q \\
\text{i} &= \text{ip} - \text{iq}
\end{align*}
\]

After Fix (Uses `psub`)

\[
\text{i} = \text{psub } p, q
\]

\[
\text{psub } p, q \overset{\text{def}}{=} \begin{cases} 
 p - q & \text{if } prov(p) = prov(q) \lor \text{prov}(p) = \text{full} \lor \text{prov}(q) = \text{full} \\
\text{poison} & \text{Otherwise}
\end{cases}
\]
To reduce Int→Ptr Casts:
Stop Canonicalizing Loads/Stores as Ints

\[
\begin{align*}
v &= \text{load} \ i64^* \ p \\
v2 &= \text{load} \ i8^{**} \ p
\end{align*}
\]

* https://godbolt.org/z/y48Mkt
To reduce Int→Ptr Casts:
Stop Canonicalizing Loads/Stores as Ints

\[
\begin{align*}
  v &= \text{load } \text{i64}* \ p \\
  v2 &= \text{load } \text{i8}** \ p \\
\end{align*}
\]

\[
\begin{align*}
  v &= \text{load } \text{i64}* \ p \\
  v2 &= \text{inttoptr } v \\
\end{align*}
\]

* [https://godbolt.org/z/y48Mkt](https://godbolt.org/z/y48Mkt)
To reduce Int→Ptr Casts:
Stop Canonicalizing Loads/Stores as Ints

```c
v = load i8** p
v2= load i8** p

v = load i64* p
v2= load i8** p

v = load i64* p
v2= inttoptr v
```

* https://godbolt.org/z/y48Mkt*
To reduce Int → Ptr Casts: Stop Canonicalizing Loads/Stores as Ints

\[
\begin{align*}
  v &= \text{load } \text{i8}** p \\
  v_2 &= \text{load } \text{i8}** p
\end{align*}
\]

Use ‘d64’ (data type) instead

\[
\begin{align*}
  v &= \text{load } \text{i64}** p \\
  v_2 &= \text{load } \text{i8}** p
\end{align*}
\]

\[
\begin{align*}
  v &= \text{load } \text{i64}** p \\
  v_2 &= \text{inttoptr } v
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>Has Provenance</th>
<th>Supports Integer operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>d64</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>i64</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Unlike cast between int ↔ ptr, d64 ↔ ptr preserves provenance.

* https://godbolt.org/z/y48Mkt
To reduce Int$\rightarrow$Ptr Casts:
Stop Canonicalizing Loads/Stores as Ints

\[
\begin{align*}
  v &= \text{load } i8** \ p \\
  v2 &= \text{load } i8** \ p
\end{align*}
\]

Use ‘d64’ (data type) instead

\[
\begin{align*}
  v &= \text{load } i64* \ p \\
  v2 &= \text{load } i8** \ p
\end{align*}
\]

\[
\begin{align*}
  v &= \text{load } i64* \ p \\
  v2 &= \text{inttoptr } v
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>Has Provenance</th>
<th>Supports Integer operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>d64</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>i64</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Unlike cast between int$\leftrightarrow$ptr, d64$\leftrightarrow$ptr preserves provenance.

* https://godbolt.org/z/y48Mkt
Conditionally Allowing Cast Elimination

// p and q have same underlying object
p2 = inttoptr(ptrtoaint p)
c  = icmp eq/ne p2, q

\[ c = \text{icmp eq/ne } p, q \]

p2 = inttoptr(ptrtoaint p)
c  = psub p2, q
\[ c = \text{psub } p, q \]

• More examples & descriptions are listed at https://github.com/aqjune/eurollvm19
Evaluation: the # of Casts

<table>
<thead>
<tr>
<th></th>
<th>Baseline (LLVM 8.0)</th>
<th>No Cast Fold</th>
<th>Reduce Cast Introduction</th>
<th>Conditionally Fold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before O3 # of ptrtoints</td>
<td>44K</td>
<td>44K</td>
<td>14K</td>
<td>14K</td>
</tr>
<tr>
<td># of inttoptrs</td>
<td>1.5K</td>
<td>1.5K</td>
<td>1.5K</td>
<td>1.5K</td>
</tr>
<tr>
<td>After O3 # of ptrtoints</td>
<td>57K</td>
<td>66K</td>
<td>11K</td>
<td>11K</td>
</tr>
<tr>
<td># of inttoptrs</td>
<td>29K</td>
<td>45K</td>
<td>5K</td>
<td>4.8K</td>
</tr>
</tbody>
</table>

- C/C++ benchmarks of SPEC2017rate + LLVM Nightly Tests used
- 81% of ptrtoints / 83% of inttoptrs removed (compared to baseline)
Evaluation: Performance Impact

- LLVM Nightly Tests (C/C++): ~0.1% avg. slowdown (-1% ~ 3.6%)
Conclusion

• Provenance helps compiler do more optimizations on pointers

• Integer with provenance works badly with integer optimizations

• We suggest separating pointers/integers conceptually

• We show how to regain performance after removing invalid optimizations

https://github.com/aqjune/eurollvm19
Conclusion

• Provenance helps compiler do more optimizations on pointers

We're updating Alive to support pointer-integer casts! 😊

**PROGRAM:** Name: pprintfload3
ENTRY:

```
v16 = ptrtoint i8* p1 to i16
p2 = inttoptr i16 v16 to i8*
v2 = load i8* p2
v1 = load i8* p1
```

**PRECONDS:**
Instruction "v2 = load i8* p2" has no UB.

**CHECK:**
Instruction "v1 = load i8* p1" has no UB?

```
v1 === v2?
```

**Result:** INCORRECT

https://github.com/aqjune/eurollvm19
supplementary slides
Constant Propagation and Readonly function

```c
char p[1], q[1] = {0};
if (foo(p, q)) {
    //readonly
    *(p+i) = 10;
    print(q[0]);
}
```
char p[1], q[1] = {0};
return (int)(p+1) == (int)q?
if (foo(p, q)) {  //readonly
  *(p+i) = 10;
  print(q[0]);
}

Constant Propagation and Readonly function

char p[1], q[1] = {0};
if (foo(p, q)) {  //readonly
  *(p+i) = 10;
  print(0);
}
Integer Equality Propagation and Performance

- Performed by many optimizations
  - CVP, Instruction Simplify, GVN, Loop Exit Value Rewrite, …

- Reduces code size
  - -10% in minisat, -6% in smg2000, -4% in simple_types_constant_folding, …

- Boosts performance in small benchmarks
  - x2000 speedup in nestedloop
Sound Optimizations that are already in LLVM

\[
\text{gep}(p, -(\text{int})q) \quad \rightarrow \quad (\text{void}*)(\text{int}p-(\text{int})q)
\]

\[
\text{select (p==null), p, null} \quad \rightarrow \quad \text{null} // \text{null}=(\text{void}*)\emptyset
\]

**Rationale**

It is safe to replace \( p \) with \((\text{void}*)(\text{int})p\).
Delayed Inbounds Checking

\[ p = (\text{char}*)0x100 \quad // \quad p=(0x100,*) \]
\[ p2 = \text{gep} \ p, 1 \quad // \quad p=(0x101,*) \]
\[ p3 = \text{gep inbounds} \ p, 1 \]
\[ \quad // \quad p = (0x101,*,\{0x100,0x101\}) \]
\[ \text{load} \ p3 \quad // \quad 0x100, \ 0x101 \ \text{should be} \]
\[ \quad // \ \text{in-bounds addr of the} \]
\[ \quad // \ \text{object at 0x101} \]