A Formal C Memory Model
Supporting Integer-Pointer Casts

Jeehoon Kang (Seoul National University)
Chung-Kil Hur (Seoul National University)
William Mansky (UPenn)
Dmitri Garbuzov (UPenn)
Steve Zdancewic (UPenn)
Viktor Vafeiadis (MPI-SWS)

@INRIA Paris
Motivation

• Integer-pointer cast is an important feature of C.
  + used in Linux kernel, Java HotSpot VM

• Pointers being integers invalidates optimizations.
  + e.g. constant propagation

• Want to support integer-pointer casts & optimizations
Integer-Pointer Casts:
Importance in Practice

• Example 1: Pointers as hash keys

  void hash_put(void* key, Data value);
  Data hash_get(void* key);

• Example 2: Pointer compression in Java HotSpot VM

  int32_t compress(void*);  // 64bit -> 32bit
  void* decompress(int32_t); // 32bit -> 64bit
Identifying Pointers with Integers: Invalidates Constant Propagation

- Anyone can access any address.

```c
extern void g();

char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
```
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

define f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
Identifying Pointers with Integers: Invalidates Constant Propagation

- Anyone can access any address.

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
```
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
g();
    return a; // -> return '0'
}
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
g();
    return a; // -> return '0'
}
Identifying Pointers with Integers: Invalidates Constant Propagation

- Anyone can access any address.

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
```
Goal of Memory Model

- To validate common optimizations by disallowing problematic memory accesses
- To allow integer-pointer casts
Validating Optimizations
Supporting Int-Ptr Casts

Outline

Validating Optimizations
Supporting Int-Ptr Casts
Outline

Supporting Int-Ptr Casts

Invalidates Most Opt.

Naive

Validating Optimizations
Outline

Validating Optimizations
Supporting Int-Ptr Casts

Invalidates Most Opt.

Naive → C11

Validating Optimizations

Supporting Int-Ptr Casts

Outline

C11
Naive

Invalidates Most Opt.

Supporting Int-Ptr Casts

Validating Optimizations
Validating Optimizations
Supporting Int-Ptr Casts

Outline

Invalidates Most Opt.

Supporting Int-Ptr Casts

Naive → C11

CompCert

Validating Optimizations
Invalidates Most Opt.

Supporting Int-Ptr Casts

Naive → C11

Ours

CompCert

Validating Optimizations

Outline
Validating Optimizations
Supporting Int-Ptr Casts

Outline
- C11
- Ours
- CompCert
- Naive

Invalidates Most Opt.
C11 Model: High-Level Idea

- Integers & pointers are tagged with permission.

<table>
<thead>
<tr>
<th>Integers</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>0x120</td>
</tr>
<tr>
<td>0x100</td>
<td>0x100</td>
</tr>
</tbody>
</table>
• Integers & pointers are tagged with permission.
Integers & pointers are tagged with permission.
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a;  // -> '0'
}

<table>
<thead>
<tr>
<th>expr</th>
<th>value {perm.}</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>0x120</td>
</tr>
<tr>
<td>&amp;b</td>
<td>0x100</td>
</tr>
<tr>
<td>&amp;b+0x20</td>
<td>0x120</td>
</tr>
</tbody>
</table>
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
C11 Model: Protection by Permission

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value {perm.}</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>0x120 {&amp;a}</td>
</tr>
<tr>
<td>&amp;b</td>
<td>0x100 {&amp;b}</td>
</tr>
<tr>
<td>&amp;b+0x20</td>
<td>0x120</td>
</tr>
</tbody>
</table>
C11 Model: Protection by Permission

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```

```
expr | value {perm.}
--- | ---
&a   | 0x120 {&a}
&b   | 0x100 {&b}
&b+0x20 | 0x120 {&b}
```
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}

expr | value {perm.}
--- | ---
&a | 0x120 {&a}
&b | 0x100 {&b}
&b+0x20 | 0x120 {&b}

Cannot Access a
Integers also need to carry permission.
Since integer-pointer casts should preserve permission.

Operations need to properly calculate permission.

```
int y = x - x; // -> int y = 0;
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x100 {&amp;a}</td>
</tr>
<tr>
<td>x - x</td>
<td>0x000 {?}</td>
</tr>
</tbody>
</table>

-> 0x000 {}
Integers also need to carry permission.
Since integer-pointer casts should preserve permission.

Operations need to properly calculate permission.

\[
\text{int } y = x - x; \quad // \rightarrow \text{int } y = 0;
\]

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x100 {&amp;a}</td>
</tr>
<tr>
<td>x - x</td>
<td>0x000 {&amp;a}</td>
</tr>
</tbody>
</table>

\[
\rightarrow 0x000 {} \quad \text{🤔}
\]
Integers also need to carry permission. Since integer-pointer casts should preserve permission.

Operations need to properly calculate permission.

```plaintext
int y = x - x; // -> int y = 0;
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x100 {&amp;a}</td>
</tr>
<tr>
<td>x - x</td>
<td>0x000 {}</td>
</tr>
<tr>
<td></td>
<td>-&gt; 0x000 {}</td>
</tr>
</tbody>
</table>
• **Integers also need to carry permission.**
  Since integer-pointer casts should preserve permission.

• **Operations need to properly calculate permission.**

```c
int y = x - x; // -> int y = 0;
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x100 {&amp;a}</td>
</tr>
<tr>
<td>x - x</td>
<td>0x000 {}</td>
</tr>
<tr>
<td>2 * x</td>
<td>0x200 {?}</td>
</tr>
<tr>
<td>x XOR x</td>
<td>0x000 {?}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
C11 Model’s Problems (2/2):
Invalidates Some Optimizations

• A useful code motion is not allowed.

```c
int a, b;
...
if (a != b) {
    a = b;
}
```

...
C11 Model’s Problems (2/2): Invalidates Some Optimizations

• A useful code motion is not allowed.

```
int a, b;
...
if (a != b) {
  a = b;
}
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x100 {&amp;x} -&gt; 0x100 {}</td>
</tr>
<tr>
<td>b</td>
<td>0x100 {}</td>
</tr>
</tbody>
</table>
We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```

Invalidates Some Optimizations

Integer type for pointers
We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

C11 Model’s Problems (2/2):

Invalidates Some Optimizations

Integer type for pointers
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- We found a real GCC bug.
  ```c
  void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
  ```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
We found a real GCC bug.

```c
void main() {
  int x = 0;
  intptr_t xi = (intptr_t) &x;
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1
```

• We found a real GCC bug.

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- We found a real GCC bug.

```c
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}  // DEAD CODE
  if (xi != i) {
    printf("unreachable\n");
    xi = i;
  }
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x);  // prints 0
}
```

[https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752)
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- We found a real GCC bug.

```c
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}  // valid
  if (xi != i) {
    printf("unreachable\n");
  }
  xi = i;    // code motion
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x);  // prints 0
}
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
We found a real GCC bug.

```c
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
  }
  xi = i; // code motion
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); }  // prints 0
```

• We found a real GCC bug.

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {}
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", 0); } // constant propagation x -> 0
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
Invalidates Most Opt.

Supporting Int-Ptr casts

Naive → C11

Complex Semantics

Invalidates Some Opt.

Validating Optimizations

Ours

CompCert

C11
Invalidates Most Opt.
Validating Optimizations
Supporting Int-Ptr Casts

Invalidates Most Opt.
Complex Semantics Invalidates Some Opt.

Supporting Int-Ptr Casts

Naive  C11  Ours

CompCert

Validating Optimizations
CompCert Model: High-Level Idea

- Pointers are different from integers.

<table>
<thead>
<tr>
<th>Integers</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}

CompCert Model:
Protection by Logical Blocks
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}

void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```c
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```

**Naive CompCert Model:**

<table>
<thead>
<tr>
<th>&amp;a</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0x100</td>
</tr>
<tr>
<td>b+0x20</td>
<td>0x120</td>
</tr>
</tbody>
</table>

**CompCert Model:**

<table>
<thead>
<tr>
<th>&amp;a</th>
<th>(l1,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>(l2,0)</td>
</tr>
<tr>
<td>b+0x20</td>
<td>(l2,0x20)</td>
</tr>
</tbody>
</table>

**Protection by Logical Blocks**

*Cannot Access a*
Invalidates Most Opt.
No Int-Ptr Casts
Validating Optimizations
Supporting Int-Ptr Casts

Naive → C11 → Ours → No Int-Ptr Casts

CompCert
Naive
Complex Semantics
Invalidates Some Opt.
Invalidates Most Opt.
No Int-Ptr Casts
Validating Optimizations
Supporting Int-Ptr Casts
Ours
CompCert
C11
No Int-Ptr Casts
Validating Optimizations
• Pointers *become* integers *only when casted*.
• Pointers become integers only when casted.
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;

Our Model:
Realizes at Casting to Integer
Our Model:
Realizes at Casting to Integer

```c
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```
Our Model: Realizes at Casting to Integer

char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;

\[
\begin{array}{c}
\text{a} \\
\text{b}
\end{array}
\begin{array}{c}
(l_1, 0) \\
(l_2, 0)
\end{array}
\]
char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;

Our Model:
Realizes at Casting to Integer
Our Model:
Realizes at Casting to Integer

char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;

Our Model: Realizes at Casting to Integer
(char*) 0x101 (l2,1)
Our Model:
Realizes at Casting to Integer

char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;

FAIL
Our Model:
Realizes at Casting to Integer
(char*) 0x101 (l2,1)
(char*) 0x120 FAIL
Our Model:

- Realizes blocks when casting to integer
- Casts back to corresponding blocks

```c
char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;

Our Model:
- Realizes blocks when casting to integer
- Casts back to corresponding blocks
```
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    //char* p = b + 0x20;
    bi = (uintptr_t) b;
    p = (char*) (bi+0x20);
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    //char* p = b + 0x20;
    bi = (uintptr_t) b;
    p = (char*) (bi+0x20);
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
Benefits of Our Model (1/6):
Still Validates Optimizations

```c
void g() {
    char b[2] = {'2', '3'};
    //char* p = b + 0x20;
    bi = (uintptr_t) b;
    p = (char*) (bi + 0x20);
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```

```
<table>
<thead>
<tr>
<th>&amp;a</th>
<th>(l1, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>(l2, 0)</td>
</tr>
<tr>
<td>bi</td>
<td>0x100</td>
</tr>
<tr>
<td>bi + 0x20</td>
<td>0x120</td>
</tr>
</tbody>
</table>
```

Cannot access a
Benefits of Our Model (2/6):
Fully Supports Integer-Pointer Casts

- Pointer-to-integer casts always succeed.
- Integer operations on casted pointers always succeed.
Benefits of Our Model (3/6): Simple Semantics

- Integer values are **just integers w/o permission**.
- Integer operations are **just** integer operations.

• Integer optimizations are allowed.
  
  ```
  int a = x - x; // -> int a = 0;
  ```

• The useful code motion is allowed.
  
  ```
  int a, b;
  ...
  if (a != b) {
      a = b;
  }
  ```
Benefits of Our Model (5/6): Easily Applicable to Compilers

- Just treat “casted pointers as escaped”.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {}
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```
Benefits of Our Model (5/6): Easily Applicable to Compilers

• Just treat “casted pointers as escaped”.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {} 
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```
Benefits of Our Model (5/6):
Easily Applicable to Compilers

• Just treat “casted pointers as escaped”.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {} // treated as escaped
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```

DEAD CODE

no constant propagation
Insignificant: performance degradation due to “casted pointers as escaped”

+ In practice, addresses casted to integers are global addresses.

+ Compilers already treat global addresses as escaped.
Invalidates Most Opt.
No Int-Ptr Casts
Validates Most Opt.
Simple Semantics
Easily Applicable to Compilers
Little Performance Penalty
C11

Fully Supports Int-Ptr Casts
Validates Most Opt.
Simple Semantics
Easily Applicable to Compilers
Little Performance Penalty
CompCert

Validating Optimizations

Supporting Int-Ptr Casts

Naive

C11

Ours

No Int-Ptr Casts
What Else is in the Paper?

• Formal definition of our memory model
• Reasoning principles for compiler verification
• Verification of other optimization examples
  + Dead code elim., dead allocation elim., arithmetic optimizations, alias analysis, etc.
• Comparison with other possible models

Fully formalized in Coq
Application to CompCert

➢ Problem
  • Non-determinism at Ptr-to-Int Casting

➢ Solution
  • Mixed-Simulation Relation
    – Forward-Simulation at deterministic steps
    – Backward-Simulation at non-deterministic steps (only ptr-to-int casting)
    – An idea from my previous work:

Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language.
Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, Viktor Vafeiadis.
ICFP 2015
http://sf.snu.ac.kr/intptrcast

Thanks!