Improving DemandedBits Analysis for Shift Operations in LLVM

Panagiotis Karouzakis

karouzakispan@gmail.com

The DemandedBits Analysis

- It’s a backwards dataflow analysis
- A demanded bit contributes to a result bit.
- Other bits can be one or zero without affecting control or data flow.

Only the *lower* 16 bits of %1 are demanded; the rest are removed by the trunc.

```
%1 = add i32 %x, %y
%2 = trunc i32 %1 to i16
```

DemandedBits lacked support for non constant shift amounts

Before:

DemandedBits of **a**:
0xffffffff

All 32 bits considered alive

```
%b2 = and i32 %b, 3
%shl = lshr i32 %a, %b2
%shl.t = trunc i32 %shl to i8
%ret i8 %shl.t
```

After:
DemandedBits of **a**:
0x000007ff
Only the lower 11 bits considered alive

DemandedBits constant shift amount on LShr

%res = lshr i32 **%a**, %ShiftAmnt

```
case Instruction::LShr;
  if (isConstant(ShiftAmnt))) {
    // ShiftAmnt is APInt (like an int)
    AB = AOut.shl(ShiftAmnt);
  }
case Instruction::shl;
  …
```

AB = DemandedBits of **%a**

AOut = DemandedBits of **%res**

ShiftAmnt = 2
(bits 1 and 3 are demanded)
AOut = 0b0000 1010
So, AB becomes
0b0010 1000

Enabling DemandedBits propagation through shift ops

What if the ShiftAmnt is not a constant?

Then the ShiftAmnt can take a range of values

ShiftAmnt $\in$ [min, max]

How can we use a shl(logical left shift) on a range?

range = [min, max]

Enabling DemandedBits propagation through shift ops

%res = lshr i32 **%a**, %ShiftAmnt

range = [1, 3]
AOut = 0b0000 1001

We need the OR of all possible shift amounts

AOut = DemandedBits of **%res**
AB = DemandedBits of **%a**

To get the ShiftedRange we

Shift by 1 to get 0b0001 0010
Shift by 2 to get 0b0010 0100
Shift by 3 to get 0b0100 1000

We then take the OR for every shift to cover all the positions.

AB = 0b0111 1110

Enabling DemandedBits propagation through shift ops

```
case Instruction::LShr{
  if(isConstant(ShiftAmnt){ …}
  else { // ShiftAmnt is not a constant.
       computeKnownBits(ShiftAmnt, Known);
       auto min = Known.getMinValue();
       auto max = Known.getMaxValue();
       // get the OR of all possible shift amounts
       // between min and max.
       GetShiftedRange(min, max, /*shiftLeft*/ true)
       // fshl exact case
  }
}
```

Simplified Alive Proof

```
define i8 %src(i8 %or, i8 %x, …, i8
%demanded)

%sh = lshr i8 %or, %x
%AOut = and i8 %sh, %demanded
ret i8 %AOut.
```

```
define i8 %tgt(i8 %or, i8 %x, ..., i8
%demanded)

; Build mask of all demanded bits

; mask = OR (demanded << s ),
                    s ∈ [min, max]

; keep only the demanded bits,
; dead bits do not matter
; or2 = (or & mask)

; res = lshr i8 %or2, %x
; result remains unchanged, thus safe
```

Evaluation : Some simplifications

Before

```
1% = and i8 %x, 127
2% = zext i8 %1 to i32
```

After

```
1% = zext i8 %x to i32
```

Before

```
%1 = sext i8 %x to i32
%2 = ashr i32 %1, C
```

After

```
%1 = zext i8 %x to i32
%2 = lshr i32 %1, C
```

Evaluation

Postgres benchmark

```
function ts_stat_sql(..):
loop:
        %10 = load i32, ptr %1

        ; sign-extend low 8 bits.
        %sext = shl i32 %10, 24
        %47 = ashr exact i32 %sext, 24
        %67 = lshr i16 %66, 14 ; low 2 bits active
        %68 = sext i16 %67 to i32
        %69 = lshr i32 %47, %68
        %70 = and i32 %69, 1
exit loop
```

```
function ts_stat_sql(..):
loop:
        %10 = load i32, ptr %1

        %67 = lshr i16 %66, 14
        %68 = sext i16 %67 to i32
        %69 = lshr i32 %10, %68
        %70 = and i32 %69, 1
        …
exit loop
```

Only the lower 4 bits of %10 can be used, sign extension is unnecessary

Appendix

Shl proof: https://alive2.llvm.org/ce/z/dpvbkq

LShr proof: https://alive2.llvm.org/ce/z/eeGzyB

Ashr proof: https://alive2.llvm.org/ce/z/EN-siK

commit link: https://github.com/llvm/llvm-project/pull/148880

Github: https://github.com/karouzakisp

If you implement similar optimizations, please add me as a reviewer.

If you are looking for a compiler engineer, contact me.