Alive2
Verifying existing optimizations

Nuno Lopes
Microsoft Research

John Regehr
University of Utah

Alive

- Found dozens of bugs in LLVM
- Avoided many other bugs due to use before commit

Verifying peephole optimizations

```
$ alive nsw.opt

----------------------------------------
Pre: WillNotOverflowSignedAdd(%x, %y)
  %r = add i4 %x, %y
=>
  %r = add i4 nsw %x, %y

Done: 1
Optimization is correct!
```

Type inference

```
$ alive shl.opt

----------------------------
  %signed = sext %y
  %r = shl %x, %signed
=>
  %unsigned = zext %y
  %r = shl %x, %unsigned

Done: 2016
Optimization is correct!
```

- Verifies all type combinations
- Integers: up to 64 bits
- Vectors: up to 4 elements
- Can take several minutes

- Be careful when fixing bit-width:
  - correct for i32 doesn't imply correct for i1!

Alive2

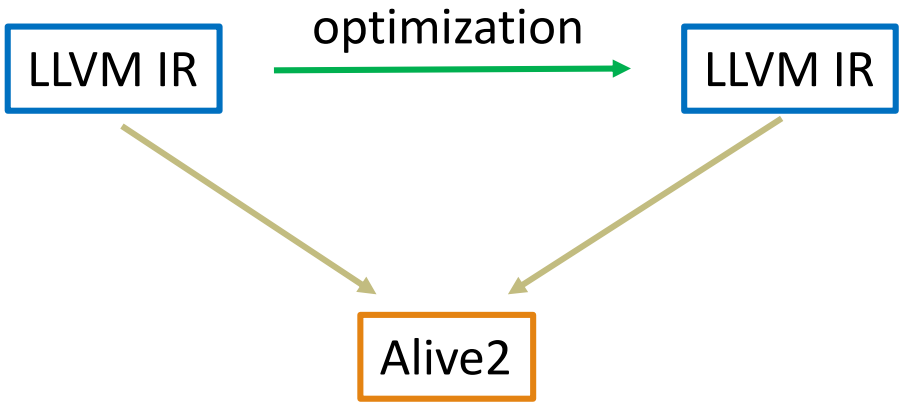
- Production-quality reimplementation of Alive in C++
- Zero false positives by design
- Goal: support all LLVM instructions plus most used intrinsics/features
- More tools: alive, alive-tv, opt plugin, clang plugin (planned)

Translation Validation

New tricks of Alive2

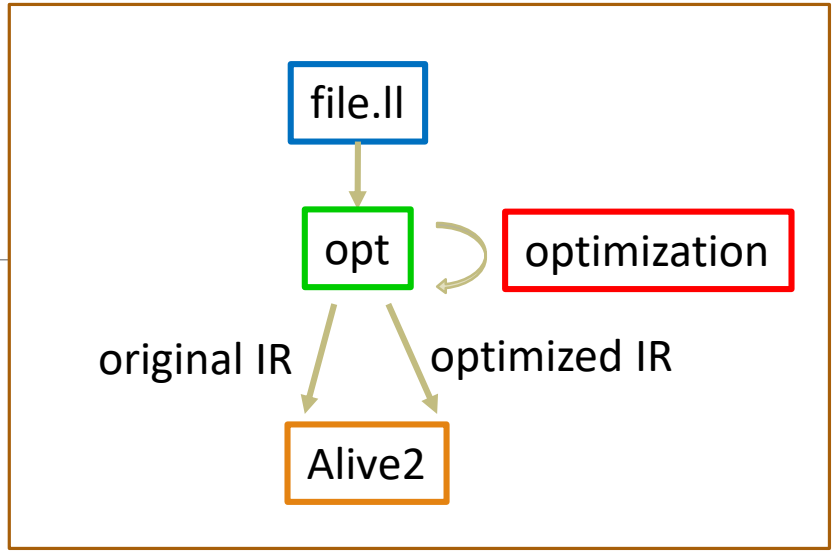
Translation Validation

- *Was* the optimization correct?



- Correct
- Not correct + example
- Timeout

opt plugin



- Alive within LLVM
- Checks if an optimization done by opt was correct

- Examples:

```
opt -load=tv.so -tv -instcombine -tv file.ll
opt -load=tv.so -tv -instcombine -tv -simplifycfg -tv file.ll
```

- TODO: `opt -verify-each …`

Finding bugs in LLVM test suite

- Experiment: run LLVM's own test suite with Alive's opt plugin
- `llvm-lit -vv -Dopt=opt-alive.sh llvm/test/Transforms`

- Script adds "-tv" around opt's arguments
- Script skips unsupported passes, like -inline, -ipconstprop, -deadargelim, etc

- About 40 minutes with 8 cores (vs 2 mins without Alive)

Bugs found in LLVM test suite so far

13 bugs reported (6 fixed)

- 6: InstCombine
- 1: InstSimplify
- 1: SimplifyCFG
- 1: ConstProp
- 1: CVP
- 1: DivRemPairs
- 1: GlobalOpt
- 1: IR utils

- Many more related with undef (not reported) & others not analyzed yet
- This is just due to Alive2, not counting things we found using original Alive

PR43665

```
; llvm/test/Transforms/InstCombine/vector-xor.ll

define <2 x i8> @test(<2 x i8> %a) {
; CHECK-LABEL: @test(
; CHECK-NEXT:    %1 = lshr <2 x i32> <i8 4, i8 undef>, %a
; CHECK-NEXT:    ret <2 x i8> %1
;
  %1 = ashr <2 x i8> <i8 -5, i8 undef>, %a
  %2 = xor  <2 x i8> <i8 -1, i8 -1>, %1
  ret <2 x i8> %2
}
```

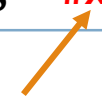
Spot the bug?

Me neither!

PR43665

```
$ opt -load=tv.so –tv –instcombine –tv xor.ll

define <2 x i8> @test(<2 x i8> %a) {
  %1 = ashr <2 x i8> { -5, undef }, %a
  %2 = xor <2 x i8> { -1, -1 }, %1
  ret <2 x i8> %2
}
=>
define <2 x i8> @test(<2 x i8> %a) {
  %1 = lshr <2 x i8> { 4, undef }, %a
  ret <2 x i8> %1
}

Transformation doesn't verify!
ERROR: Value mismatch
```

```
Example:
<2 x i8> %a = < #x00 (0),  #x04 (4)  >

Source:
<2 x i8> %1 = < #xfb (-5), #x00 (0)  >
<2 x i8> %2 = < #x04 (4),  #xff (-1) >

Target:
<2 x i8> %1 = < #x04 (4),  #x08 (8)  >

Source value: < #x04 (4),  #xff (-1) >
Target value: < #x04 (4),  #x08 (8)  >
```



Mismatch in 2nd element of returned vector

PR43665

```
$ opt -load=tv.so –tv –instcombine –tv xor.ll

define <2 x i8> @test(<2 x i8> %a) {
  %1 = ashr <2 x i8> { -5, undef }, %a
  %2 = xor <2 x i8> { -1, -1 }, %1
  ret <2 x i8> %2
}
=>
define <2 x i8> @test(<2 x i8> %a) {
  %1 = lshr <2 x i8> { 4, undef }, %a
  ret <2 x i8> %1
}

Transformation doesn't verify!
ERROR: Value mismatch
```

Bits:

```
%x = sxyz.abcd

ashr %x, 4 == ssss.sxyz

lshr %x, 4 == 0000.sxyz
```

If s = 1:

```
ashr %x, 4 == 1111.1xyz
lshr %x, 4 == 0000.1xyz
```

There’s no value %x can take that makes these values equal!
(xor result with -1 doesn’t help)

Alive-tv

- Takes 2 LLVM IR files and checks if the transformation is correct
- Very useful to try an optimization before implementing it

```
; src.ll
define i1 @test(i32 %idx) {
  %ptr0 = call i8* @malloc(i64 4)
  %ptr = getelementptr i8, i8* %ptr0, i32 %idx
  call void @free(i8* %ptr)
  %cmp = icmp eq i32 %idx, 0
  ret i1 %cmp
}
```

```
; tgt.ll
define i1 @test(i32 %idx) {
  %ptr0 = call i8* @malloc(i64 4)
  %ptr = getelementptr i8, i8* %ptr0, i32 0
  call void @free(i8* %ptr)
  ret i1 true
}
```

```
$ alive-tv src.ll tgt.ll
…
Transformation seems to be correct!
```

Order of arguments matters!

What's verified?

- Refinement of the return value of functions
- Refinement of final memory isn't checked ATM (coming next month)

- -disable-undef-input: What if undef didn't exist? Assume function arguments are not undef

```
define i8 @test(i8 %x) {
  %add = add i8 %x, undef
  ret i8 %add
}
=>
define i8 @test(i8 %x) {
  ret i8 0
}
```

```
define void @test(i8* %p) {
  store i8 3, i8* %p
  ret void
}
=>
define void @test(i8* %p) {
  store i8 42, i8* %p
  ret void
}
```

Features

- Most integer instructions
- Vectors (partial)
- Floats (no fast math)
- Some intrinsics
- Memory (partial)
- Loops (very limited)

Limitations (medium-term)

- Intra-procedural only
- No inttoptr
- Final memory not checked

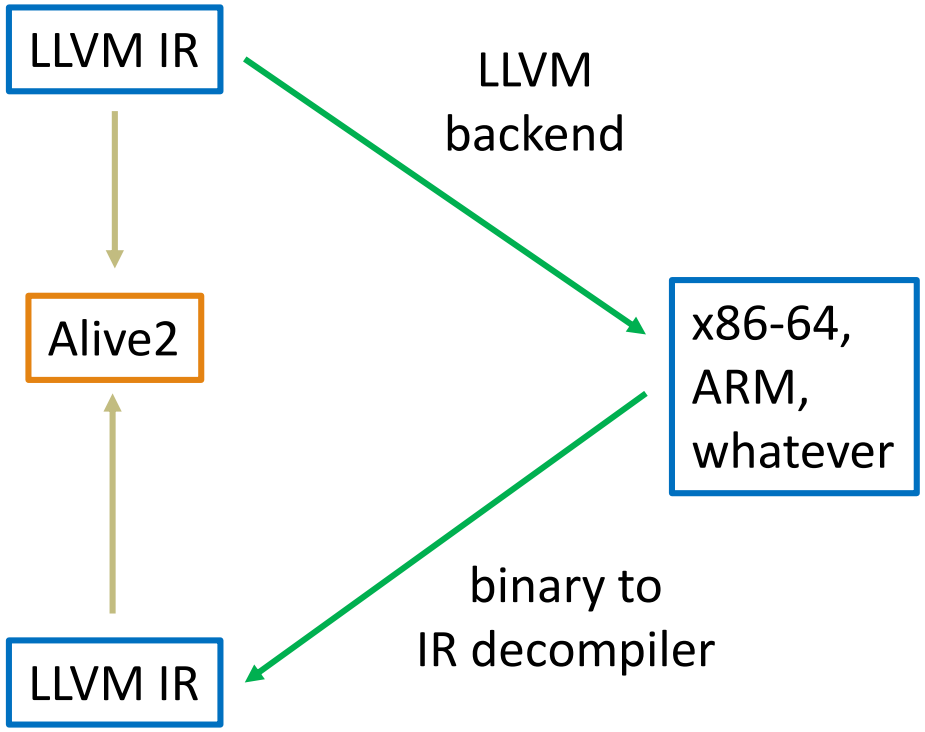
- Trusted:
  - TLI data (list of known functions, alloc size, alignments, etc)

Beyond Optimizations

Finding bugs in backends

- LLVM backends contain a lot of complexity, we'd like to make sure they don't have latent crash or miscompile bugs
- Of course, Alive2 only understands IR!
- Can we exploit decompilers that lift object code to IR?

Finding bugs in backends



Generating IR for Backend Testing

- opt-fuzz is a bounded exhaustive generator of IR functions
  - 1 insn == 5,876 functions
    - Quick smoke test
  - 2 insns == 2,524,808 functions
    - Testing these takes a while
  - 3 insns == fills 2 TB disk, oops
    - Better use a cluster!
  - 4 insns == ??
    - Probably infeasible without cutting corners somewhere
- Great for exploring corner cases not emitted by Clang
- Gives us fine-grained control of code properties
  - Operation widths, use of intrinsics, UB flags, etc.
- Small functions are less likely to trigger solver timeouts

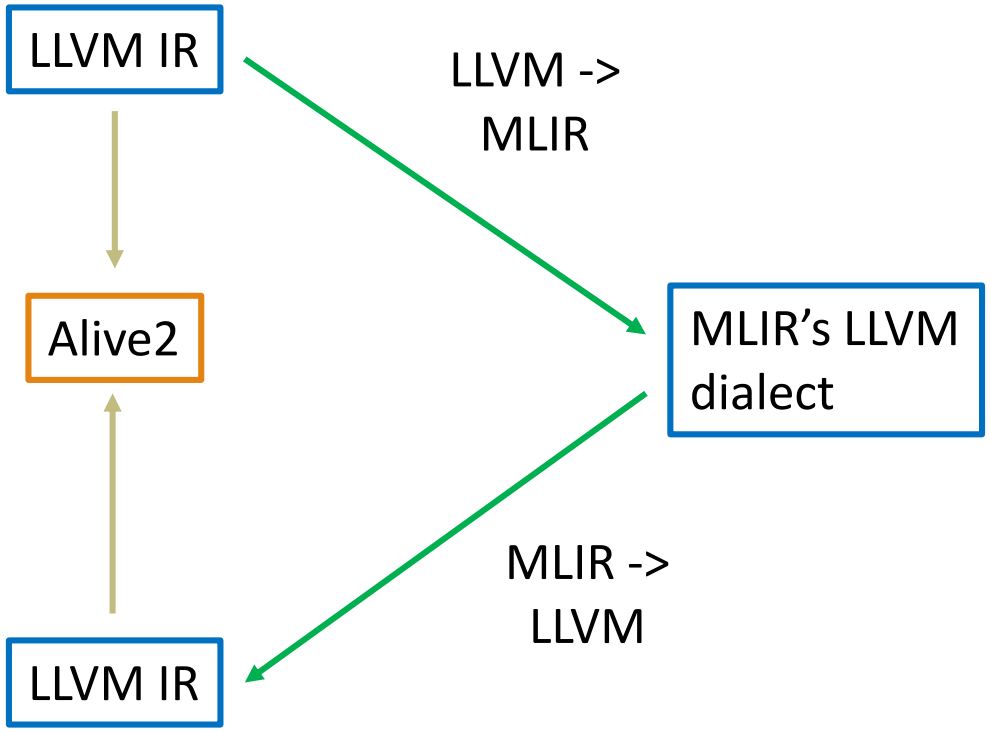
Finding bugs in backends

- So far we've found only decompiler bugs 😂 😂
  - But, only started working on this a couple weeks ago
- UB in LLVM is an interesting complication for correct decompilation
  - For example, this is defined for all values of cl:

```
shl %eax, %cl
```

  - So a decompiler must mask off high bits of %cl

Beyond backends



Conclusion

- Alive2: fully automatic verification of LLVM optimizations
  - Once and for all for peephole optimizations (alive)
  - When running an optimization: translation validation (alive-tv, opt -tv, clang -tv)

- Please help LLVM & thank you for the help so far:
  - Fixing bugs
  - Reporting bugs found in LLVM test suite using Alive2
  - Using Alive2 when you make a change to LLVM (no more new bugs!)

https://github.com/AliveToolkit/alive2

Alive -root-only

```
$ alive reorder.opt

--------------------------------
  %a = mul %x, %y
  %r = mul %a, %z
  ret %r            ; %x * %y * %z
=>
  %a = mul %x, %z
  %r = mul %a, %y
  ret %r            ; %x * %z * %y

ERROR: Value mismatch for i8 %a

Example:
i8 %x = #x01 (1)
i8 %y = #x01 (1)
i8 %z = #x00 (0)
Source value: #x01 (1)
Target value: #x00 (0)
```

```
$ alive -root-only reorder.opt

--------------------------------
  %a = mul %x, %y
  %r = mul %a, %z
  ret %r
=>
  %a = mul %x, %z
  %r = mul %a, %y
  ret %r

Done: 320
Optimization is correct!
```

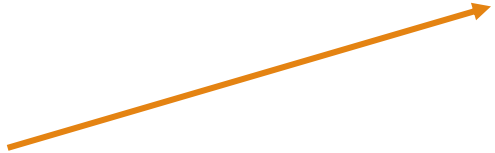
Memory

```
define i8 @alloca_malloc() {
  %ptr = alloca i64 1
  store i8 10, * %ptr
  %v = load i8, * %ptr
  ret i8 %v
}
=>
define i8 @alloca_malloc() {
  %ptr = alloca i64 1
  store i8 20, * %ptr
  %v = load i8, * %ptr
  ret i8 %v
}
```

```
Transformation doesn't verify!
ERROR: Value mismatch

Example:

Source:
* %ptr = pointer(local, block_id=256, offset=0)
i8 %v = #x0a (10)

Target:
* %ptr = pointer(local, block_id=256, offset=0)
i8 %v = #x14 (20)

Source value: #x0a (10)
Target value: #x14 (20)
```

Memory model

```
* %ptr = pointer(local, block_id=256, offset=0)
```



Local:
- Stack
- Locally-allocated heap

Non-local:
- Global variables
- Function arguments

Each allocation:
- Distinct block id
- Ids never reused (even after dealloc)

Offset within the block
ptrtoint %ptr = start_of(block_id) + offset

- gep only changes offset
- block_id can’t be fabricated (aliasing rules)