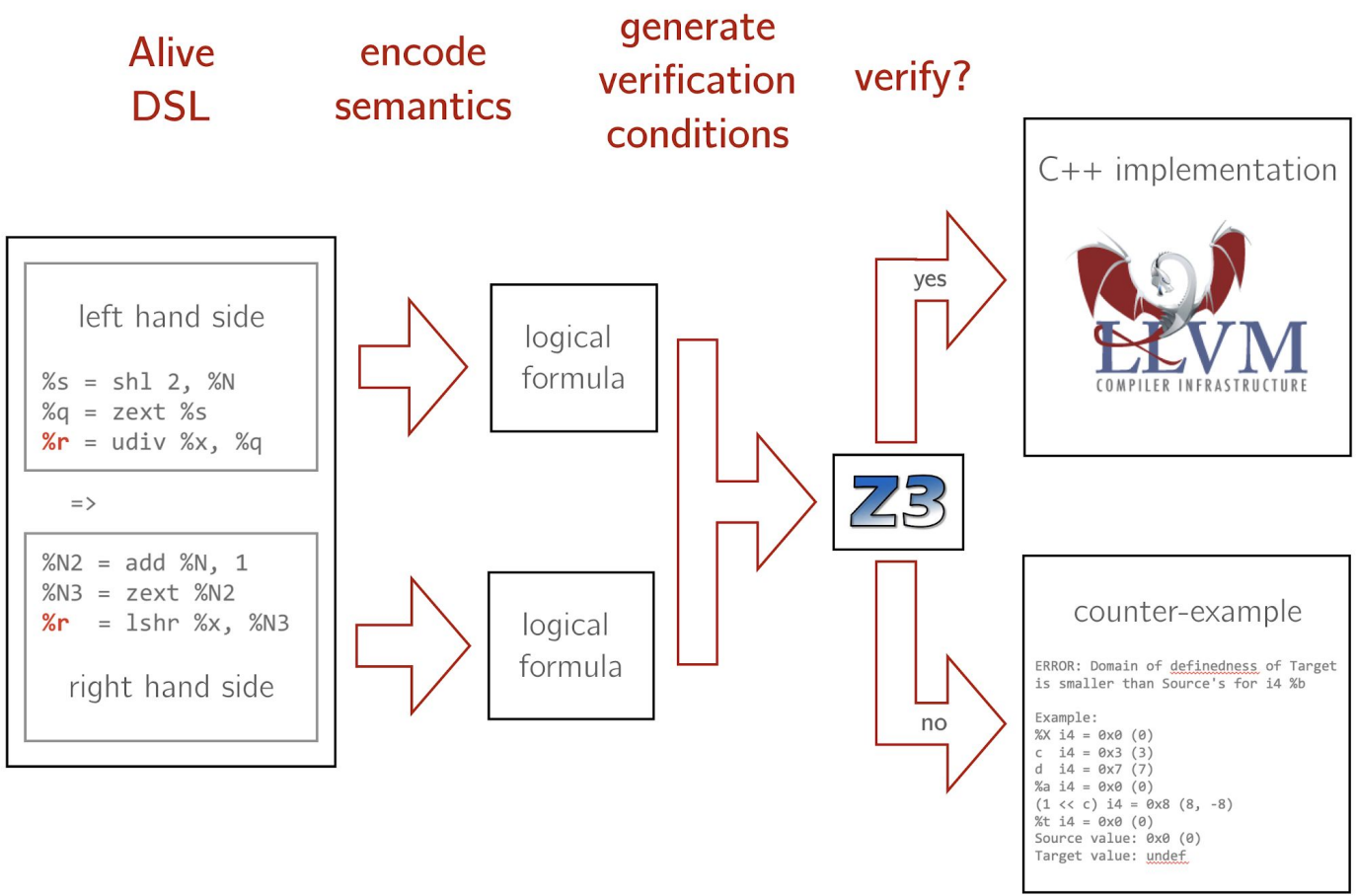
COL 731 Paper Presentation

Provably correct peephole optimizations with Alive

Compiler Verification

- Random testing effective but not perfect.
- Complete compiler verification and translation validations not practical for most real world use cases.
- Alive aims for a design point that is both practical and formal.
- Alive is a domain specific language for formally verifying some peephole optimizations in LLVM.



1. *CS 6120: Provably Correct Peephole Optimizations with Alive*. (2019, December 6) https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/alive/

Syntax

- A transformation has form precondition
  source template
  =>
  target template
- Syntax of templates resembles LLVM IR.
- Implements a subset of LLVM integer and pointer instructions.
- Does not support branches.

Type System

- Alive supports integers of any bitwidth till an upper limit(64, by default).
- Pointers, arrays and void also supported.
- Does not support floating point and aggregate types like vectors.
- Variables can be implicitly typed.
- Correctness checked for all feasible types according to the typing rules.

Undefined Behaviours

1. Undefined behaviour :- Anything may happen to program on its execution. Compiler may simply assume it does not occur.
   It may occur due to out of bounds and unaligned memory access.
2. Undefined value :- Mimics a free-floating register that can return any value each time it is read. Compiler may choose any possible value for optimization.

Undefined Behaviours

3. Poison Values :-

   Indicates that a side-effect free instruction produces undefined behaviour.

   It defers true undefined behaviour till it is used by an instruction with side effects.

   nuw, *nsw*, *exact* attributes may produce a poison value.

Checking correctness

- When undefined behaviours absent, we can just equivalence check for all possible values of inputs.
- In presence of undefined behaviours, we check
  - target is defined when source is defined.
  - target is poison free when source is defined and poison free.
  - source and target produce same result when source is defined and poison free.

Verification condition generation

For each instruction, Alive generates three SMT expressions

1. An expression for the result of the operation.
2. An expression representing constraints under which the instruction has defined behaviour.
3. An expression representing cases for which the instruction does not generate a poison value.

Definedness and poison-free constraints of an instruction are aggregates of its own constraints and those of its operands.

Verification condition generation

- *undef* values are encoded as a fresh SMT variable and added to a set *U.*
- Peephole optimizations make use of results of dataflow analyses through built-in predicates.
- These predicates are encoded using a fresh boolean variable *p* and a side constraint.
- If it is an over-approximation constraint is *p* => *s* where *s* denotes the condition of the analysis.

Verification condition generation

- Under-approximations are denoted as *s* => *p* .
  Eg. isPower2(%a) is encoded as *p* => *a != 0 Λ (a&(a-1)==0)*
  mayAlias(%a, %b) is *a* == *b* => *p*
- Results of precise analyses as encoded precisely.

Verification condition generation

Instruction	Definedness Constraint
sdiv a, b	$b \neq 0 \wedge (a \neq \mathit{INT\_MIN} \vee b \neq -1)$
udiv a, b	$b \neq 0$
srem a, b	$b \neq 0 \wedge (a \neq \mathit{INT\_MIN} \vee b \neq -1)$
urem a, b	$b \neq 0$
shl a, b	$b <_u B$
lshr a, b	$b <_u B$
ashr a, b	$b <_u B$

Verification condition generation

Instruction	Constraints for Poison-free execution
add nsw a, b add nuw a, b	$SExt(a, 1) + SExt(b, 1) = SExt(a + b, 1)$ $ZExt(a, 1) + ZExt(b, 1) = ZExt(a + b, 1)$
sub nsw a, b sub nuw a, b	$SExt(a, 1) - SExt(b, 1) = SExt(a - b, 1)$ $ZExt(a, 1) - ZExt(b, 1) = ZExt(a - b, 1)$
mul nsw a, b mul nuw a, b	$SExt(a, B) \times SExt(b, B) = SExt(a \times b, B)$ $ZExt(a, B) \times ZExt(b, B) = ZExt(a \times b, B)$
sdiv exact a, b udiv exact a, b	$(a \div b) \times b = a$ $(a \div_u b) \times b = a$
shl nsw a, b shl nuw a, b	$(a << b) >> b = a$ $(a << b) >>_u b = a$
ashr exact a, b lshr exact a, b	$(a >> b) << b = a$ $(a >>_u b) << b = a$

Correctness criteria

ϕ denotes preconditions.

δ definedness constraints of source

ρ poison free constraints of source

ι result of executing source instructions

I set of input variables

P set of fresh variables for predicates

$\psi \equiv \rho \wedge \phi \wedge \delta$

Correctness criteria

1. $\forall_{\mathcal{I},P,\overline{\mathcal{U}}}\ \exists_{\mathcal{U}} : \psi \implies \overline{\delta}$
2. $\forall_{\mathcal{I},P,\overline{\mathcal{U}}}\ \exists_{\mathcal{U}} : \psi \implies \overline{\rho}$
3. $\forall_{\mathcal{I},P,\overline{\mathcal{U}}}\ \exists_{\mathcal{U}} : \psi \implies \iota = \bar{\iota}$

Verification condition generation

To encode memory accesses, we create a variable to represent the pointer resulting from alloca.

1. It should be different than 0.
2. It must be properly aligned.
3. Enforce that the allocated memory range does not overlap with other allocated regions
4. Ensure that the allocated memory range does not wrap around the memory space.

These constraints are added to a set α for each pointer.

Verification condition generation

- Load is encode using select and concat.
  Eg. %v = load i16* %p becomes v = concat(select(m,p+1),select(m,p))
- Store is encoded using ite, store and extract.
  Eg. store %v, %p becomes
  ite(δ, m'', m), m' = store(m,p,extract(v,7,0)),
  m'' = store(m', p+1, extract(v,15,8))

4. $\forall_{\mathcal{I},\overline{\mathcal{U}},i}\ \exists \mathcal{U} : \phi \wedge \alpha \wedge \overline{\alpha} \implies \mathsf{select}(m,i) = \mathsf{select}(\overline{m},i)$

Verification condition generation

The array encoding we saw might be inefficient for SMT solvers so we use Ackermann's expansion.

store(m,q,v) becomes ite(p=q, v,m)

Load instructions take the whole memory expression built so far recursively.
select(m,q) becomes ite(q=p1, v1, ite (.....))

Attribute inference

- Difficult to decide when to place nsw, nuw, exact attributes in LLVM optimizations.
- Removing attributes in transformed code constraints subsequent optimization passes.
- If an optimization is correct without an attribute in source code, then it is also correct with it.
- If an optimization is correct with an attribute in target code, then it remains correct without it.

Attribute inference

- Alive can infer when it is safe to add attributes to target code or remove attributes from source code.
- Attributes only affect the poison free constraints so these constraints are generated conditionally based on presence or absence of attributes.
- A fresh boolean variable f is introduced for each instruction and for each attribute and poison free constraints become $\rho \equiv f_1 \implies p_1 \wedge \ldots \wedge f_n \implies p_n$
- p_i denote the poison free condition when f_i is enabled.

Attribute inference

Let $\mathcal{F}, \overline{\mathcal{F}}$ denote the set of all f variables in source and target respectively.

The disjunction of all models satisfying $\exists_{\mathcal{F},\overline{\mathcal{F}}} : c_1 \wedge c_2 \wedge c_3 \wedge c_4$ gives the optimal set of attributes.

Attribute inference

Pseudocode:

$\Phi := \mathsf{true}$

for each type assignment **do**

$f := \exists_{\mathcal{F},\overline{\mathcal{F}}} : \Phi \wedge c_1 \wedge c_2 \wedge c_3 \wedge c_4$

$\mu := \mathsf{false}$

while f is satisfiable with model m **do**

$b := \{l \mid l \in \mathcal{F} \wedge m(l)\} \cup \{\neg l \mid l \in \overline{\mathcal{F}} \wedge \neg m(l)\}$

$\mu := \mu \vee \bigwedge b$

$f := f \wedge \neg \bigwedge b$

$\Phi := \Phi \wedge \mu$

Generating C++ code

- After a transformation has been proved correct, Alive can turn it into C++ code.
- The generated code can be linked into LLVM and used as an LLVM optimization pass.
- Code generator translates the source code into conditions using LLVM's pattern matching library. If DAG of LLVM instructions matches and preconditions are met, the root instruction from the source is replaced by its counterpart in the target.
- Alive is parametric over types so if given constraints are not enough to unify types in a particular instruction, then conditions are inserted to check their equality in the C++ code generated.

Implementation

- Alive is implemented in Python and uses Z3 SMT solver.
- Typing constraints are over quantified or quantifier-free bitvector theories.
- Correctness constraints are negated before querying solver so for transformations without undefined values in source, we get quantifier-free formulas and formulas with single quantifier otherwise.

Evaluation

- Alive translated 334 out of 1028 InstCombine transformations.
- It was able to find 8 incorrect InstCombine transformations . 2 in AddSub and 6 in MulDivRem file.
- It was able to infer weaker precondition attributes and stronger postcondition attributes for 70 transformations.
- Transformations involving floating point cannot yet be expressed in Alive.

Incorrect Transformations

Transformation:

%a = sdiv %X, C

%r = sub 0, %a

=>

%r = sdiv %X, -C

Example:

%X i4 = 0x8 (8, -8)

C i4 = 0x1 (1)

%a i4 = 0x8 (8, -8)

Source value: 0x8 (8, -8)

Target value: undef

Failed SMT query:

```
(set-info :status unknown)
(declare-fun C () (_ BitVec 4))
(declare-fun %X () (_ BitVec 4))
(assert
 (let (($x323 (and (distinct %X (_ bv8 4)) true)))
 (let (($x332 (or $x323 (and (distinct (bvneg C) (_ bv15
4)) true))))
 (let ((?x329 (bvneg C)))
 (let (($x330 (and (distinct ?x329 (_ bv0 4)) true)))
 (let (($x348 (and $x330 $x332)))
 (let (($x326 (or $x323 (and (distinct C (_ bv15 4))
true))))
 (let (($x321 (and (distinct C (_ bv0 4)) true)))
 (and $x321 $x326 (not $x348)))))))))
(check-sat)
```

Incorrect Transformations

Transformation:

Pre: C2 % (1<<C1) == 0
%s = shl nsw %X, C1
%r = sdiv %s, C2
=>
%r = sdiv %X,
C2/(1<<C1)

Example:

%X i4 = 0xF (15, -1)

C1 i4 = 0x3 (3)
C2 i4 = 0x8 (8, -8)
%s i4 = 0x8 (8, -8)
Source value: 0x1 (1)
Target value: 0xF (15, -1)

Transformation:

%Op0 = lshr %X, C1
%r = udiv %Op0, C2
=>
%r = udiv %X, C2 << C1

Example:

%X i8 = 0x00 (0)

C1 i8 = 0x04 (4)
C2 i8 = 0x80 (128, -128)
%Op0 i8 = 0x00 (0)
Source value: 0x00 (0)
Target value: undef

Transformation:

%Op1 =sub 0,%X
%r= srem %Op0, %Op1
=>
%r= srem %Op0, %X

Example:

%X i4 = 0xF (15, -1)

%Op0 i4 = 0x8 (8, -8)
%Op1 i4 = 0x1 (1)
Source value: 0x0 (0)
Target value: undef

Incorrect Transformations

Transformation:

```
%B = sub 0, %A
%C = sub nsw %x, %B
 =>
%C = add nsw %x, %A
```

Example:

%A i4 = 0x8 (8, -8)
%x i4 = 0xA (10, -6)
%B i4 = 0x8 (8, -8)
Source value: 0x2 (2)
Target value: poison

Transformation:

```
Pre: isPowerOf2(C1)
%r= mul nsw %x, C1
 =>
%r= shl nsw %x, log2(C1)
```

Example:

%x i4 = 0x1 (1)
C1 i4 = 0x8 (8, -8)
Source value: 0x8 (8, -8)
Target value: poison

Transformation:

```
Pre: !WillNotOverflowSignedMul(C1, C2)
%Op0 = sdiv %X, C1
%r= sdiv %Op0, C2
=>
%r = 0
```

Example:

%X i9 = 0x100 (256, -256)
C1 i9 = 0x100 (256, -256)
C2 i9 = 0x1FF (511, -1)
%Op0 i9 = 0x001 (1)

Source value: 0x1FF (511, -1)
Target value: 0x000 (0)

Incorrect Transformations

Transformation:

Pre: isPowerOf2(%Power) && hasOneUse(%Y)
%s= shl %Power, %A
%Y= lshr %s, %B
%r= udiv %X, %Y
=>
%sub = sub %A, %B
%Y = shl %Power, %sub
%r = udiv %X, %Y

Example:
%Power i4 = 0x8 (8, -8)
%A i4 = 0x0 (0)

%B i4 = 0x1 (1)
%X i4 = 0x0 (0)
%s i4 = 0x8 (8, -8)
%Y i4 = 0x4 (4)
%sub i4 = 0xF (15, -1)
Source value: 0x0 (0)
Target value: undef

Conclusion

- Compiler LLVM+Alive was made by replacing InstCombine optimizer from LLVM with the C++ generated by Alive for the transformations.
- Compilation with -O3 flag using LLVM+Alive was on an average 7% faster than LLVM on SPEC 2000 and SPEC 2006.
- Execution time was 3% slower than LLVM with O3.
- These may be because Alive runs only a fraction of InstCombine instructions.

Conclusion

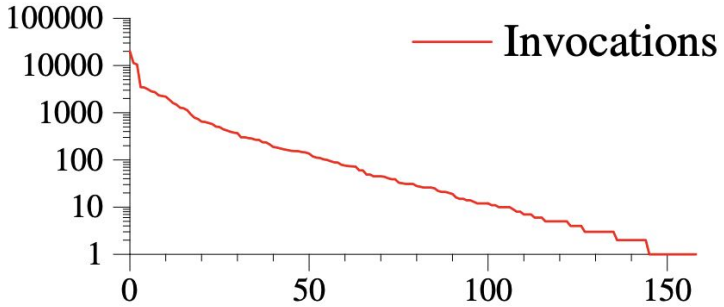


Figure reports the number of times that each optimization fired during compilation of the LLVM nightly test suite and SPEC bench- marks using LLVM+Alive at -O3. Alive optimizations fired about 87,000 times in total. The top ten optimizations account for approximately 70% of the total invocations and there is a long tail of infrequently-used optimizations.