ægraphs: Acyclic E-graphs
for Efficient Optimization in a Production Compiler

Chris Fallin (Fastly)
EGRAPHS 2023
chris@cfallin.org
In the beginning, there was a compiler backend...
Cranelift
Cranelift

• Open-source general-purpose optimizing compiler backend
  • Written in Rust + a pattern-matching DSL (~200KLoC, ~130KLoC tests)
  • SSA input, four ISAs (x86-64, aarch64, riscv64, s390x)
Cranelift

- Open-source general-purpose optimizing compiler backend
  - Written in Rust + a pattern-matching DSL (~200KLoC, ~130KLoC tests)
  - SSA input, four ISAs (x86-64, aarch64, riscv64, s390x)
- Used in production as part of Wasmtime
- O(3-5) active developers at any time
Cranelift

- **Speed**: JIT focus
- **Simplicity**: “not LLVM”
- **Verifiability**: explicitly design with fuzzing + formal techniques + … in mind
- **Research-friendliness**: we need new ideas to compete with larger peers
1. Why we want a rewrite-based optimizer
2. How to turn a CFG into an egraph and back again
3. Cycles why they occur, and what to do about them
4. Results how well does it work?
5. Lessons in translating research to production
1. Why we want a rewrite-based optimizer
2. How to turn a CFG into an egraph and back again
3. Cycles why they occur, and what to do about them
4. Results how well does it work?
5. Lessons in translating research to production
Cranelift, circa mid-2022

• Focus on codegen quality & mid-end optimizations
• We had: GVN, constant folding, LICM, some simple rewrites
Cranelift, circa mid-2022

- Focus on codegen quality & mid-end optimizations
- We had: GVN, constant folding, LICM, some simple rewrites

```
v1 = ...
v2 = iadd_imm v1, 16
    ...
v10 = iadd_imm v1, 16
```

```
v1 = ...
v2 = iadd_imm v1, 16
    ...
v10 -> v2
```
Cranelift, circa mid-2022

- Focus on codegen quality & mid-end optimizations
- We had: GVN, constant folding, LICM, some simple rewrites
- We added: alias analysis => redundant load elim + store-to-load forwarding
Cranelift, circa mid-2022

- Focus on codegen quality & mid-end optimizations
- We had: GVN, constant folding, LICM, some simple rewrites
- We added: alias analysis => redundant load elim + store-to-load forwarding

```plaintext
v1 = ...

v2 = load.i64 v1+8  
v10 = load.i64 v1+8
...

v1 = ...

v2 = load.i64 v1+8
...

v10 -> v2
```
The Pass-Order Problem

• What do we do with this program?

\[
\begin{align*}
v_1 &= \ldots \\
v_2 &= \text{iadd\_imm} \ v_1, 16 \\
v_3 &= \text{load\_i64} \ v_2 \\
\ldots
\end{align*}
\]

\[
\begin{align*}
v_{10} &= \text{iadd\_imm} \ v_1, 16 \\
v_{11} &= \text{load\_i64} \ v_{10}
\end{align*}
\]
The Pass-Order Problem

- What do we do with this program?

\[
\begin{align*}
v1 &= \ldots \\
v2 &= \text{iadd_imm} \ v1, 16 \\
v3 &= \text{load.i64} \ v2 \\
\ldots \\
v10 \rightarrow v2 \\
v11 &= \text{load.i64} \ v10
\end{align*}
\]
The Pass-Order Problem

• What do we do with this program?

\[
\begin{align*}
v1 &= \ldots \\
v2 &= \text{iadd_imm} \ v1, \ 16 \\
v3 &= \text{load.i64} \ v2 \\
\ldots & \ \\
v10 \to v2 \\
v11 \to v3
\end{align*}
\]
The Pass-Order Problem

• What do we do with this program?

\[
\begin{align*}
v1 &= \ldots \\
v2 &= \text{load.i64 } v1+8 \\
v3 &= \text{iadd } v2, v1 \\
\ldots \\
v10 &= \text{load.i64 } v1+8 \\
v11 &= \text{iadd } v10, v1
\end{align*}
\]
The Pass-Order Problem

- What do we do with this program?

\[
\begin{align*}
  v_1 & = \ldots \\
  v_2 & = \text{load.i64 } v_1 + 8 \\
  v_3 & = \text{iadd } v_2, v_1 \\
  \ldots & \\
  v_{10} & \rightarrow v_2 \\
  v_{11} & \rightarrow v_3
\end{align*}
\]
The Pass-Order Problem

• Proposed optimization pipeline:

```rust
fn optimize(&mut self) {
    self.gvn();
    self.rle();
    self.gvn();
}
```
The Pass-Order Problem

- Proposed optimization pipeline:
  
  ```rust
  fn optimize(&mut self) {
    self.gvn();
    self.rle();
    self.gvn();
    self.gvn();
    self.rle(); // XXX just in case
  }
  ```
The Pass-Order Problem

• Surely other production compilers have solved this problem?
The Pass-Order Problem

• Surely other production compilers have solved this problem?

GCC:
The Pass-Order Problem

• Surely other production compilers have solved this problem?

GCC: NEXT_PASS (pass_cse);
The Pass-Order Problem

• Surely other production compilers have solved this problem?

    NEXT_PASS (pass_cselim);
    ...
    NEXT_PASS (pass_cse_sincos);
    NEXT_PASS (pass_cse_reciprocals);

GCC:
    ...
    NEXT_PASS (pass_cse);
    ...
    NEXT_PASS (pass_cse_after_global_opts);
    ...
    NEXT_PASS (pass_cse2);
    ...
    NEXT_PASS (pass_postreload_cse);
The Pass-Order Problem

- Surely other production compilers have solved this problem?

GCC:

```c
TERMINATE_PASS_LIST (all_passes)
POP_INSERT_PASSES ()
NEXT_PASS (pass_postreload);
NEXT_PASS (pass_ira);
NEXT_PASS (pass_early_remat);
NEXT_PASS (pass_live_range_shrinkage);
NEXT_PASS (pass_partition_blocks);
NEXT_PASS (pass_if_after_combine);
NEXT_PASS (pass_initialize_regs);
NEXT_PASS (pass_rtl_dse1);
NEXT_PASS (pass_cse2);
NEXT_PASS (pass_cse_after_global_opts);
NEXT_PASS (pass_rtl_store_motion);
IN ...
```
The Pass-Order Problem
(or: “Fix-point All The Things?”)

- Goal: find a way to put all (ish) of our optimizations in a single fixpoint loop

... cprop, algebraic rewrites, strength reduction, ...
The Pass-Order Problem
(or: “Fix-point All The Things?”)

• Goal: find a way to put all (ish) of our optimizations in a single fixpoint loop
  • Remember: compile-cost focus! (We can’t afford to run a pass N times)
  • We’d prefer not to maintain a brittle heuristic pass order

GVN
RLE

... cprop, algebraic rewrites, strength reduction, ...
Adding some “simple” rewrites

```plaintext
match opcode {
    Opcode::IaddImm
  | Opcode::ImulImm
  | Opcode::BorImm
  | Opcode::BandImm
  | Opcode::BxorImm => {
      // Fold binary_op(C2, binary_op(C1, x)) into binary_op(binary_op(C1, C2), x)
      if let ValueDef::Result(arg inst, ) = pos.func.dfg.value def(arg) {
```
Adding some “simple” rewrites (!)

```rust
match opcode {
    Opcode::IaddImm
    | Opcode::ImulImm
    | Opcode::BorImm
    | Opcode::BandImm
    | Opcode::BxorImm => {
        // Fold binary_op(C2, binary_op(C1, x)) into binary_op(binary_op(C1, C2), x)
        if let ValueDef::Result(arg_inst, _) = pos.func.dfg.value_def(arg) {
            if let InstructionData::BinaryImm64 {
                opcode: prev_opcode,
                arg: prev_arg,
                imm: prev_imm,
            } = &pos.func.dfg.insts[arg_inst]
            {
                if opcode == *prev_opcode
                    && ty == pos.func.dfg.ctrl_typevar(arg_inst)
                {
                    let lhs: i64 = imm.into();
                    let rhs: i64 = (*prev_imm).into();
                }
            }
        }
    }
```
Adding some “simple” rewrites (!!!)
Adding some simple rewrites

```
(rule (simplify
    (iadd (fits_in_64 ty)
      (iconst ty (u64_from_imm64 k1))
      (iconst ty (u64_from_imm64 k2))))
    (subsume (iconst ty (imm64_masked ty (u64_add k1 k2)))))))
```

*(Cranelift’s ISLE term-rewriting DSL)*
Adding some simple rewrites

;; ineg(ineg(x)) == x.
(rule (simplify (ineg ty (ineg ty x))) (subsume x))
Many kinds of optimizations can be expressed as value rewrites

- Constant prop \((1 + 2 \Rightarrow 3)\), algebraic \((x + 0 \Rightarrow x)\), strength reduction, …
Rewrite Systems for Optimization

• Many kinds of optimizations can be expressed as value rewrites
  • Constant prop \((1 + 2 \Rightarrow 3)\), algebraic \((x + 0 \Rightarrow x)\), strength reduction, …
• Those that can’t are often “rewrite-adjacent”
  • Normalization of input terms to rewriter => GVN
  • Placement of rewritten terms => LICM, code motion in general
Rewrite Systems for Optimization

- Many kinds of optimizations can be expressed as value rewrites
  - Constant prop (1 + 2 => 3), algebraic (x + 0 => x), strength reduction, …
- Those that can’t are often “rewrite-adjacent”
  - Normalization of input terms to rewriter => GVN
  - Placement of rewritten terms => LICM, code motion in general
- Rewriting is a well-defined framework that works well for verification!
  - “This value is equal to that value”
1. Why  
   we want a rewrite-based optimizer

2. How  
   to turn a CFG into an egraph and back again

3. Cycles  
   why they occur, and what to do about them

4. Results  
   how well does it work?

5. Lessons  
   in translating research to production
1. Why  we want a rewrite-based optimizer
2. How  to turn a CFG into an egraph and back again
3. Cycles why they occur, and what to do about them
4. Results how well does it work?
5. Lessons in translating research to production
Optimization pipeline

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8
Optimization pipeline

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)
### Optimization pipeline

- **block0(v0, v1):**
  - \( v2 = iadd \, v0, \, v1 \)
  - \( v3 = isub \, v0, \, v2 \)
  - If \( v3 \),
    - block1(v2),
    - block2(v3)

- **block1(v4):**
  - \( v5 = iconst \, 1 \)
  - \( v6 = isub \, v4, \, v5 \)
  - \( br \) block3(v6)

- **block2(v7):**
  - \( br \) block3(v7)

- **block3(v8):**
  - return v8

---

```
block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)

block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)

block2(v7):
br block3(v7)

block3(v8):
return v8
```
Optimization pipeline

\[
\begin{align*}
\text{block0}(v0, v1): & \\
& v2 = iadd v0, v1 \\
& v3 = isub v0, v2 \\
& \text{if v3,} \\
& \text{block1}(v2), \\
& \text{block2}(v3) \\
\text{block1}(v4): & \\
& v5 = iconst 1 \\
& v6 = isub v4, v5 \\
& \text{br block3}(v6) \\
\text{block2}(v7): & \\
& \text{br block3}(v7) \\
\text{block3}(v8): & \\
& \text{return v8}
\end{align*}
\]

\[x + 0 \Rightarrow x\]

\[\ldots\]
Optimization pipeline

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8

x + 0 => x
...

E-graph + CFG == ???

```
block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8
```
E-graph + CFG == ???

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8
E-graph + CFG == ???

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8

egraph per basic block:
+ simple
- limited rewrite scope
- limited sharing/amortization
- rules out control optimizations
E-graph + CFG == ???

\[ \text{block0}(v0, v1): \]
\[ v2 = \text{iadd} v0, v1 \]
\[ v3 = \text{isub} v0, v2 \]
\[ \text{if } v3, \]
\[ \text{block1}(v2), \]
\[ \text{block2}(v3) \]

\[ \text{block1}(v4): \]
\[ v5 = \text{iconst} 1 \]
\[ v6 = \text{isub} v4, v5 \]
\[ \text{br block3}(v6) \]

\[ \text{block2}(v7): \]
\[ v7 = \text{isub} v4, v5 \]
\[ \text{br block3}(v7) \]

\[ \text{block3}(v8): \]
\[ \text{return } v8 \]
E-graph + CFG == ???

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8
E-graph + CFG == ???

Region-nodes in egraph:

+ powerful optimizations!
+ strongly normalizing
+ more compact IR
+ cheaper analysis?

- very different from CFG (conversion overheads)
- side-effects are tricky
- issues with irreducible control flow
E-graph + CFG == ???

Region-nodes in egraph:

+ powerful optimizations!
+ strongly normalizing
+ more compact IR
+ cheaper analysis?

- very different from CFG (conversion overheads)
- side-effects are tricky
- issues with irreducible control flow

Jamey Sharp’s prototype: https://github.com/jameysharp/optir
E-graph + CFG == ???

block0(v0, v1):
    v2 = iadd v0, v1
    v3 = isub v0, v2
    if v3,
        block1(v2),
        block2(v3)

block1(v4):
    v5 = icnst 1
    v6 = isub v4, v5
    br block3(v6)

block2(v7):
    br block3(v7)

block3(v8):
    return v8
E-graph + CFG == ???

block0(v0, v1):
  if v3,
  block1(v2),
  block2(v3)

block1(v4):
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8

v0 + v1 - 1 -
E-graph + CFG == ???

block0(v0, v1):
  if v3,
  block1(v2),
  block2(v3)

block1(v4):
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8
E-graph + CFG == ???

CFG skeleton contains:
- all blocks, with blockparams
- side-effecting operators
- block terminators (branches)
E-graph + CFG == ???

egraph contains:
- blockparam values, as terminals
- all pure operators, 
  *without* associated location
E-graph + CFG == ???

block0(v0, v1):
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8
E-graph + CFG == ???

egraph with CFG skeleton:

+ cheap to convert to/from CFG
  + algorithmically and in implementation
+ optimizations across function scope (mostly)

- harder to express rewrites that alter side-effects
- need special support for “seeing through” blockparams
E-graph + CFG == ???

*egraph with CFG skeleton:*

+ cheap to convert to/from CFG
  + algorithmically *and* in implementation
+ optimizations across function scope (mostly)

- harder to express rewrites that alter side-effects
- need special support for “seeing through” blockparams

---

good enough for now! (incremental approach)
Optimization pipeline

\[
\begin{align*}
\text{block0}(v0, v1): & \\
& v2 = \text{iadd} v0, v1 \\
& v3 = \text{isub} v0, v2 \\
& \text{if } v3, \\
& \text{block1}(v2), \\
& \text{block2}(v3)
\end{align*}
\]

\[
\begin{align*}
\text{block1}(v4): & \\
& v5 = \text{iconst} 1 \\
& v6 = \text{isub} v4, v5 \\
& \text{br block3}(v6)
\end{align*}
\]

\[
\begin{align*}
\text{block2}(v7): & \\
& \text{br block3}(v7)
\end{align*}
\]

\[
\begin{align*}
\text{block3}(v8): & \\
& \text{return } v8
\end{align*}
\]

\[
\begin{align*}
x + 0 & \Rightarrow x \\
\ldots
\end{align*}
\]
Optimization pipeline

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8

x + 0 => x
...

block8(v8, v1):
  v2 = iadd v8, v1
  v3 = isub v8, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8
Lowering to a CFG

block0(v0, v1):
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8
Lowering to a CFG

block0(v0, v1):
if v3,
  block1(v2),
  block2(v3)
Lowering to a CFG

block0(v0, v1):
    v2 = iadd v0, v1
    v3 = isub v0, v2
    if v3,
       block1(v2),
       block2(v3)
Elaboration

block0(v0, v1):
    store ec0, ec4
    return ec2

\[ v0 + v1 - 42 = ec2 + 42 = ec3 \]

ec0 \rightarrow v0
ec1 \rightarrow v1
ec2 \rightarrow +
ec3 \rightarrow 42
ec4 \rightarrow -
Elaboration

block0(v0, v1):
  store ec0, ec4
  return ec2

eclass  elaborated
ec0    v0
ec1    v1
Elaboration

block0(v0, v1):
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2

eclass    elaborated
ec0       v0
ec1       v1
ec4       v2

* Note: assume extraction (node selection) is done already!
Elaboration

block0(v0, v1):
    v3 = iadd ec0, ec1
    v2 = isub ec2, ec3
    store ec0, ec4
    return ec2

eclass  | elaborated
--------|----------
ec0     | v0       
ec1     | v1       
ec2     | v3       
ec4     | v2       

diagram:
ec0     
/   
ec2  +  42  ec3
|    |
\  /  \  
ec4 - ec1
Elaboration

block0(v0, v1):
  v3 = iadd v0, v1
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2

table:

<table>
<thead>
<tr>
<th>eclass</th>
<th>elaborated</th>
</tr>
</thead>
<tbody>
<tr>
<td>ec0</td>
<td>v0</td>
</tr>
<tr>
<td>ec1</td>
<td>v1</td>
</tr>
<tr>
<td>ec2</td>
<td>v3</td>
</tr>
<tr>
<td>ec4</td>
<td>v2</td>
</tr>
</tbody>
</table>
block0(v0, v1):
  v3 = iadd v0, v1
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
Elaboration

block0(v0, v1):
  v3 = iadd v0, v1
  v4 = icnst 42
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2

eclass | elaborated
-------|--------
ec0    | v0
ec1    | v1
ec2    | v3
ec3    | v4
ec4    | v2
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store ec0, ec4
  return ec2

eclass | elaborated
-------|---------
ec0    | v0      
ec1    | v1      
ec2    | v3      
ec3    | v4      
ec4    | v2      

Elaboration

block0(v0, v1):
  v3 = iadd v0, v1
  v4 = icnst 42
  v2 = isub v3, v4
  store v0, v2
  return ec2

class  elaborated
ec0    v0
ec1    v1
ec2    v3
ec3    v4
ec4    v2
Elaboration

\[
\text{block0}(v0, v1): \\
v3 = iadd v0, v1 \\
v4 = iconst 42 \\
v2 = isub v3, v4 \\
store v0, v2 \\
return ec2
\]

eclass | elaborated
---|---
ec0 | v0
ec1 | v1
ec2 | v3
ec3 | v4
ec4 | v2
Elaboration

block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store v0, v2
  return v3

eclass  elaborated
ec0     v0
ec1     v1
ec2     v3
ec3     v4
ec4     v2
Elaboration... twice?

```
block0(v0, v1):
    ...

block1:
    return ec2

block2:
    return ec4
```

```
ec0  v0  v1  ec1
   ν  +  42  ec3

ec2  -  ec4
```
Elaboration... twice?

block0(v0, v1):
  v2 = iadd v0, v1

block1:
  return v2

block2:
  v3 = iconst 42
  v4 = isub v2, v3
  return v4
Elaboration... twice?

block0(v0, v1):
  v2 = iadd v0, v1

partial redundancy!

block1:
  return v2

block2:
  v3 = icnst 42
  v4 = isub v2, v3
  return v4

block2:
  ... no use of v2 ...
  return v50
 SSA

 block0

 block1

 block2

 block3

 CFG
A dominates B if all paths to B first pass through A.
A dominates B if all paths to B first pass through A. Dominance forms a tree. Many compiler algorithms work by traversing the domtree.
SSA: A value’s definition dominates its uses.
SSA

block0:
  v1 = ...

block1
  v2 = ... v1 ...

block2
  v3 = ... v1 ...

block3
  v4 = ... v1 ...

CFG

Dominator Tree

block0
  → block1
  → block2
  → block3

SSA: A value's definition dominates its uses.
SSA: A value’s definition dominates its uses.
SSA: A value’s definition dominates its uses.

CFG

Dominator Tree

block0

block1 block2 block3

SSA

block0: v1 = ...

block1 v2 = ... v1 ...

block2 v3 = ... v1 ...

block3 v4 = ... v2 ...

CFG
block0(v0, v1):
  v2 = iadd v0, v1

block1
  v3 = iadd v0, v1

block2

block3

GVN (Global Value Numbering):
If an operator dominates a duplicate copy of itself, reuse the original.
GVN (Global Value Numbering): If an operator dominates a duplicate copy of itself, reuse the original.
block0(v0, v1):
  v2 = iadd v0, v1

block1
v3 <- v2

block2

block3

GVN (Global Value Numbering):
If an operator dominates a duplicate copy of itself, reuse the original.

Implement with domtree preorder traversal and a scoped map.
block0(v0, v1):
  v2 = iadd v0, v1

v3 <- v2

Implement with domtree preorder traversal and a scoped map.
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  ... = op ec2

block2:
  ...
  ... = op ec2
  return ec4

block3:
  ...
  ... = op ec2
Scoped Elaboration

block0(v0, v1):
...

block1:
... = op ec2

block2:
... = op ec2
return ec4

block3:
... = op ec2

block0

block1  block2
  ↓
block3
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  ... = op ec2

block2:
  ... = op ec2
  return ec4

block3:
  ... = op ec2
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  ... = op ec2

block2:
  ... = op ec2
  return ec4

block3:
  ... = op ec2
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  ... = op ec2

block2:
  ...
  ... = op ec2
  return ec4

block3:
  ...
  ... = op ec2

eclass  elaborated
  ec0    v0
  ec1    v1
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  v2 = ...
  ... = op v2

block2:
  ... = op ec2
  return ec4

block3:
  ... = op ec2

eclass elaborated
  ec0   v0
  ec1   v1
  ec2   v2
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  v2 = ...
  ... = op v2

block2:
  ... = op ec2
  return ec4

block3:
  ... = op ec2
Scoped Elaboration

block0(v0, v1):
    ...

block1:
    v2 = ...
    ... = op v2

block2:
    ... = op ec2
    return ec4

block3:
    ... = op v2
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  v2 = ...
  ... = op v2

block2:
  ... = op ec2
  return ec4

block3:
  ... = op v2

reuse!

block0

block1

block2

block3

eclass  elaborated
ec0      v0
ec1      v1
ec2      v2
Scoped Elaboration

block0(v0, v1):
    ...

block1:
    v2 = ...
    ... = op v2

block2:
    ... = op ec2
    return ec4

block3:
    ... = op v2
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  v2 = ...
  ... = op v2

block2:
  ... = op ec2
  return ec4

block3:
  ... = op v2

block0
  block1
    block2
      block3

eclass   elaborated
  ec0   v0
  ec1   v1
  ec2   v2
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  v2 = ...
  ... = op v2

block2:
  ... = op ec2
  return ec4

block3:
  ... = op v2

eclass elaborated
  ec0  v0
  ec1  v1
Scoped Elaboration

block0(v0, v1):
  ...

block1:
  v2 = ...
  ... = op v2

block2:
  ... = op ec2
  return ec4

block3:
  ... = op v2

eclass  elaborated
  ec0  v0
  ec1  v1
Scoped Elaboration

block0(v0, v1):
... 

block1:
v2 = ...
... = op v2

block2:
v3 = ...
... = op v3
return ec4

block3:
... = op v2

eclass  elaborated

ec0: v0
ec1: v1
ec2: v3
Scoped Elaboration

```plaintext
block0(v0, v1):
  ...

block1:
  v2 = ...
  ... = op v2

block2:
  v3 = ...
  ... = op v3
  return ec4

block3:
  ... = op v2

duplicate!
```

```
block0
  block1
  block2
  block3

eclass  elaborated
  ec0  v0
  ec1  v1
  ec2  v3
```
Scoped Elaboration

Scoped elaboration subsumes GVN

block0(v0, v1):
  ...

block1:
  v2 = ...
  ... = op v2

block2:
  v3 = ...
  ... = op v3
  return ec4

block3:
  ... = op v2

eclass  elaborated
  ec0   v0
  ec1   v1
  ec2   v3
Scoped Elaboration

Scoped elaboration subsumes GVN + LICM (choose to insert higher in loopnest + scoped map)
Scoped Elaboration

Scoped elaboration subsumes GVN + LICM (choose to insert higher in loopnest + scoped map) + Rematerialization (choose to create duplicate anyway)
1. Why  we want a rewrite-based optimizer
2. How  to turn a CFG into an egraph and back again
3. Cycles  why they occur, and what to do about them
4. Results  how well does it work?
5. Lessons  in translating research to production
1. Why we want a rewrite-based optimizer
2. How to turn a CFG into an egraph and back again
3. Cycles why they occur, and what to do about them
4. Results how well does it work?
5. Lessons in translating research to production
Rewrites and Repair

Rewrite: \( x + 0 \Rightarrow x \)
Rewrites and Repair

Rewrite: \( x + 0 \Rightarrow x \)
Rewrites and Repair

Rewrite: $x + 0 \Rightarrow x$
Rewrites and Repair

Rewrite: $x + 0 \Rightarrow x$

Fixup requires backlinks (parent pointers) and re-interning, which are costly.
Rewrites and Repair

Rewrite: $x + 0 \Rightarrow x$

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*.
Rewrites and Repair

Rewrite: $x + 0 \Rightarrow x$

Fixup requires backlinks (parent pointers) and re-interning, which are costly

Parents:
$\{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2)\}$

Parents:
$\{ec5 := (+ ec3, ec2)\}$
Rewrites and Repair

Rewrite: \( x + 0 \Rightarrow x \)

Fixup requires backlinks (parent pointers) and re-interning, which are costly.

Parents:
\[ \{ ec3 := (+ ec0, ec1), \ ec4 := (+ ec0, ec2), \ ec5 := (+ ec3 ec2) \} \]
Rewrites and Repair

Rewrite: \( x + 0 \Rightarrow x \)

Fixup requires backlinks (parent pointers) and re-interning, which are costly.

Re-intern
\[ ec3 := (+ ec0 ec1) \]

Parents:
\{ ec3 := (+ ec0 ec1),
    ec4 := (+ ec0 ec2),
    ec5 := (+ ec3 ec2) \}
Rewrites and Repair

Rewrite: \( x + 0 \Rightarrow x \)

Fixup requires backlinks (parent pointers) and re-interning, which are costly.

Re-intern
\[ ec4 := (+ ec0 ec2) \]

Parents:
\{ ec3 := (+ ec0 ec1),
     ec4 := (+ ec0 ec2),
     ec5 := (+ ec3 ec2) \}
Rewrites and Repair

Rewrite: \( x + 0 \Rightarrow x \)

Fixup requires backlinks (parent pointers) and re-interning, which are costly

Parents:
\[
\{ ec3 := (+ ec0, ec1),
\quad ec4 := (+ ec0, ec2),
\quad ec5 := (+ ec3 ec2) \}
\]

Re-intern
\[
ec5 := (+ ec3 ec2)
= (+ ec0 ec2)
= ec4
\]
Rewrites and Repair

Rewrite: $x + 0 \Rightarrow x$

Fixup requires backlinks (parent pointers) and re-interning, which are costly.

Eliminate:
- Parent lists?
- Duplicated storage of nodes?
- Merging of parent lists, with dedup’ing?
Rewrites and Repair

Rewrite: \( x + 0 \Rightarrow x \)

Fixup requires backlinks (parent pointers) and re-interning, which are costly

Eliminate:
- Parent lists?
- Duplicated storage of nodes?
- Merging of parent lists, with dedup’ing?

\( \Rightarrow \) compile + memory overhead too high vs. traditional compiler pipeline
Rewrites and Repair

Idea: no need to repair uses if eclass is in *final form* before we use it!
Rewrites and Repair

Idea: no need to repair uses if eclass is in *final form* before we use it!
Rewrite eagerly?!
Rewrite eagerly!?
Rewrite eagerly?!
Rewrite eagerly?!  

Rewrite already occurred → ec5 hash-conses to ec4
Rewrite eagerly?!

How do we handle this cycle?
Rewrite eagerly?!

How do we handle this cycle?
- Cycles preclude single pass (imply fixpoint algorithm)
Rewrite eagerly?!

How do we handle this cycle?
- Cycles preclude single pass (imply fixpoint algorithm)
- We’re rewriting the arg after its use (no longer eager) → need parent lists again
Cycles in E-graphs
Cycles in E-graphs

\[ x + 0 \Rightarrow x \]
Cycles in E-graphs

\[ x + 0 \rightarrow x \]
Cycles in E-graphs

\[ x + 0 \Rightarrow x \]

Observation:
- egraph does not record rewrite “direction”
Cycles in E-graphs

$C + 0 \Rightarrow C$

Observation:
- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with $C$
  - rewrite with $C => C + 0$
**Cycles in E-graphs**

\[
x + 0 \Rightarrow x
\]

Observation:
- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with \(x\)
  - rewrite with \(x \Rightarrow x + 0\)
- rewrite rules that equate *part* to *whole* are (reverse)-generative
Cycles in E-graphs

\[ x + 0 \Rightarrow x \]

Observation:
- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with \( x \)
  - rewrite with \( x \Rightarrow x + 0 \)
- rewrite rules that equate part to whole are (reverse)-generative

Cycles occur even if original egraph is acyclic (e.g., from SSA)
Persistent immutable e-classes

![Diagram showing e-classes ec0, ec1, and ec3 with v0 and 0 nodes]
Persistent immutable e-classes

- Never rewrite a node
- Represent e-classes as trees of union nodes
Persistent immutable e-classes

- Never rewrite a node
- Represent eclasses as trees of union nodes
Persistent immutable e-classes

- Never rewrite a node
- Represent eclasses as trees of union nodes
- As we build the egraph, track latest id for a given value
- Invoke rewrite rules when a node is created
- enter into hashcons map with final union’d ID
Persistent immutable e-classes

Eager rewriting

Acyclicity

Persistent immutable data structure
Persistent immutable e-classes

- Eager rewriting
- Acyclicility

Enables persistent immutable data structure (otherwise, uses don’t pick up optimized defs)
Persistent immutable e-classes

Eager rewriting

Enables (otherwise, uses don’t pick up optimized defs)

Persistent immutable data structure

Maintains (creating a cycle requires mutable args)

Acyclicility
Persistent immutable e-classes

Eager rewriting

- Enables (otherwise, uses don’t pick up optimized defs)

Persistent immutable data structure

- Allows (otherwise, need to revisit + do fixpoint)

Acyclicity

- Maintains (creating a cycle requires mutable args)
E-graph vs. ægraph

**Egg-style egraph:**
- batched rewriting + repair

**Ægraph:**
- eager rewriting + immutable union nodes

**Pros:**
- Strongly normalizing
- Supports arbitrarily cyclic input
  
**Cons:**
- Requires parent pointers and rehashing on fixup
- Repair step is a fixpoint

- Can miss rewrites (depending on rule structure)
- Cannot support cyclic input (e.g., seeing through phi-nodes)

- Single-pass rewrite
- No parent pointers (minimal memory + maintenance overhead)
E-graph vs. ægraph

**egg-style egraph:**
batched rewriting + repair

- Strongly normalizing
- Supports arbitrarily cyclic input

- Requires parent pointers and rehashing on fixup
- Repair step is a fixpoint

**ægraph:**
eager rewriting + immutable union nodes

- Can miss rewrites (depending on rule structure)
- Cannot support cyclic input (e.g., seeing through phi-nodes)

+ Single-pass rewrite
+ No parent pointers (minimal memory + maintenance overhead)

We can avoid batched repair because *there is no repair*
How We Write Rules

• Two fundamental compromises: acyclicity and more targeted rewrites
How We Write Rules

• Two fundamental compromises: acyclicity and more targeted rewrites
  • No catchall associativity or commutativity rewrites!

(iadd a b) => (iadd b a)

(iadd a (iadd b c))
=> (iadd (iadd a b) c)
How We Write Rules

• Two fundamental compromises: acyclicity and *more targeted rewrites*
  
• No catchall associativity or commutativity rewrites!

\[
(iadd\ a\ b) \Rightarrow (iadd\ b\ a)
\]

\[
(iadd\ a\ (iadd\ b\ c)) \Rightarrow (iadd\ (iadd\ a\ b)\ c)
\]
How We Write Rules

• Two fundamental compromises: acyclicity and more targeted rewrites
  • Only limited “non-directional” rewrites
How We Write Rules

• Two fundamental compromises: acyclicity and *more targeted rewrites*
  
• Only limited “non-directional” rewrites
  
(bnot (band a b)) => (bor (bnot a) (bnot b))
How We Write Rules

• Two fundamental compromises: acyclicity and *more targeted rewrites*
  
• Only limited “non-directional” rewrites

\((\text{bnot} \ (\text{band} \ a \ b)) \Rightarrow (\text{bor} \ (\text{bnot} \ a) \ (\text{bnot} \ b))\)

→ OK (part of a “strategy”: push bnots downward)

→ But let’s not also have the other direction!
How We Write Rules

- Two fundamental compromises: acyclicity and more targeted rewrites
  - No catchall associativity or commutativity rewrites!
  - Only limited “non-directional” rewrites
How We Write Rules

- Two fundamental compromises: acyclicity and more targeted rewrites
  - *No catchall associativity or commutativity rewrites!*
  - Only limited “non-directional” rewrites
- Acyclicity precludes rules that operate over blockparams (phis)
How We Write Rules

• Two fundamental compromises: acyclicity and *more targeted rewrites*
  • *No catchall associativity or commutativity rewrites!*
  • Only limited “non-directional” rewrites
  • Acyclicity precludes rules that operate over blockparams (phis)

These limitations are OK!
At least as powerful as traditional rewrites; and we’ve solved phase ordering; and we can make use of “multi-version” + cost-based extraction.
What E-graphs Gave Us

- If not full EqSat + repair phase, what do ægraphs take from e-graphs?
- **Rewriting**: a powerful unifying paradigm for optimizations
- **Multiple value representations**: explores all rewrite paths; cost function makes final resolution in principled way
- **Sea-of-nodes IR for pure values**: natural framework for code motion
The ægraph Passes

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = icst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8

x + 0 => x
The ægraph Passes

1. Build ægraph and eagerly rewrite
The ægraph Passes

block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)

block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)

block2(v7):
  br block3(v7)

block3(v8):
  return v8

x + 0 => x
...
2. Perform extraction
The ægraph Passes

2. Perform extraction

- Greedy heuristic
- Dynamic programming (single pass)
The ægraph Passes

3. Scoped elaboration
The ægraph Passes

1. Build and rewrite
2. Extraction
3. Scoped elaboration

Three linear passes, no fix-point loops
1. Why we want a rewrite-based optimizer
2. How to turn a CFG into an egraph and back again
3. Cycles why they occur, and what to do about them
4. Results how well does it work?
5. Lessons in translating research to production
1. Why we want a rewrite-based optimizer
2. How to turn a CFG into an egraph and back again
3. Cycles why they occur, and what to do about them
4. Results how well does it work?
5. Lessons in translating research to production
Performance
Performance

SpiderMonkey.wasm

- 11% faster runtime
- 2% longer compile-time
Performance

SpiderMonkey.wasm

- 11% faster runtime
- 2% longer compile-time

bz2

- 3% higher runtime
- 2% faster compile-time
# Performance

## Speedups

<table>
<thead>
<tr>
<th>Name</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>gimli</td>
<td>22%</td>
</tr>
<tr>
<td>spidermonkey</td>
<td>11%</td>
</tr>
<tr>
<td>minicsv</td>
<td>9%</td>
</tr>
<tr>
<td>ratelimit</td>
<td>8%</td>
</tr>
<tr>
<td>switch</td>
<td>3%</td>
</tr>
<tr>
<td>fib2</td>
<td>3%</td>
</tr>
<tr>
<td>intgemm</td>
<td>1%</td>
</tr>
</tbody>
</table>
# Performance

<table>
<thead>
<tr>
<th>Speedups</th>
<th>Slowdowns</th>
</tr>
</thead>
<tbody>
<tr>
<td>gimli</td>
<td>random</td>
</tr>
<tr>
<td>22%</td>
<td>-31%</td>
</tr>
<tr>
<td>spidermonkey</td>
<td>hex-simd</td>
</tr>
<tr>
<td>11%</td>
<td>-16%</td>
</tr>
<tr>
<td>minicsv</td>
<td>meshoptimizer</td>
</tr>
<tr>
<td>9%</td>
<td>-14%</td>
</tr>
<tr>
<td>ratelimit</td>
<td>ed25519</td>
</tr>
<tr>
<td>8%</td>
<td>-13%</td>
</tr>
<tr>
<td>switch</td>
<td>blake3-simd</td>
</tr>
<tr>
<td>3%</td>
<td>-6%</td>
</tr>
<tr>
<td>fib2</td>
<td>keccak</td>
</tr>
<tr>
<td>3%</td>
<td>-4%</td>
</tr>
<tr>
<td>intgemm</td>
<td>bz2</td>
</tr>
<tr>
<td>1%</td>
<td>-3%</td>
</tr>
</tbody>
</table>
Performance

**Speedups**

- Instruction scheduling: #6260
- Missing opt rules
  - Magic div constants: #6049

**Slowdowns**

- random: -31%
- hex-simd: -16%
- meshoptimizer: -14%
- ed25519: -13%
- blake3-simd: -6%
- keccak: -4%
- bz2: -3%
I just did some benchmarking of egraphs and the perf improvement is huge on the benchmark I tried:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (mean ± σ)</th>
<th>Range (min – max)</th>
<th>User [s]</th>
<th>System [s]</th>
<th>Runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark 1: ./raytracer_cg_clif</td>
<td>8.553 ± 0.010 s</td>
<td>8.543 s – 8.568 s</td>
<td>8.539</td>
<td>0.014</td>
<td>10</td>
</tr>
<tr>
<td>Benchmark 2: ./raytracer_cg_clif_egraph</td>
<td>6.068 ± 0.017 s</td>
<td>6.047 s – 6.108 s</td>
<td>6.057</td>
<td>0.011</td>
<td>10</td>
</tr>
<tr>
<td>Benchmark 3: ./raytracer_cg_clif_release</td>
<td>6.450 ± 0.021 s</td>
<td>6.410 s – 6.482 s</td>
<td>6.439</td>
<td>0.012</td>
<td>10</td>
</tr>
<tr>
<td>Benchmark 4: ./raytracer_cg_clif_release_egraph</td>
<td>5.853 ± 0.053 s</td>
<td>5.779 s – 5.908 s</td>
<td>5.841</td>
<td>0.012</td>
<td>10</td>
</tr>
</tbody>
</table>

Summary
'./raytracer_cg_clif_release_egraph' ran
1.04 ± 0.01 times faster than './raytracer_cg_clif_egraph'
1.10 ± 0.01 times faster than './raytracer_cg_clif_release'
1.46 ± 0.01 times faster than './raytracer_cg_clif'
Project Health & Enablement

PRs to add mid-end opts

egraph-based mid-end

Time
Project Health & Enablement

PRs to add mid-end opts

egraph-based mid-end

Time
Optimize sign extension via shifts (#6220)
egraphs: Add 'bmask' bit pattern optimization rule (#6196)
Add 'multi_lane' precondition to 'bitselect' => `{u,s}{min,max}` rewrite (#6201)
ISLE: simplify select/bitselect when both choices are the same (#6141)
Add egraph cprop optimizations for `splat` (#6148)
ISLE: rewrite loose inequalities to strict inequalities and strict inequalities to equalities (#6130)
ISLE: rewrite `and`/`or` of `icmp` (#6095)
cranelift: rewrite `iabs(ineg(x))` and `iabs(iabs(x))` (#6072)
cranelift: rewrite `x*-1` to `ineg(x)` (#6052)
cranelift: cancel `ineg` when args to `imul` (#6053)
cranelift: simplify `icmp` against UMAX/SMIN/SMAX (#6037)
cranelift: simplify `x-x` to `0` (#6032)
cranelift: simplify `fneg(fneg(x))` to `x` (#6034)
cranelift: simplify `ineg(ineg(x))` to `x` (#6033)
Add egraph optimization for fneg's cancelling out (#5910)
cranelift: Generalize `(x << k)` optimization (#5746)
cranelift: Optimize `select+icmp` into `{s,u}{min,max}` (#5546)
cranelift: Collapse double extends into a single extend (#5772)
Generalize and/or/xor optimizations (#5744)
Algebraic opts: Reuse `iconst 0` from LHS (#5724)
Add some minor souper-harvested optimizations (#5735)
cranelift: Only build iconst for ints <= 64 bits (#5723)
Legalize `(and,or,xor) NOT` into component instructions (#5709)
egraphs/cprop: Don't extend constants to `i128` (#5717)
Generalize u/sextend constant folding to all types (#5706)
cranelift: Correctly wrap shifts in constant propagation (#5695)
Constant-fold icmp instructions (#5666)
cranelift: Rewrite `or((and(x, y), not(y)) => or(x, not(y)))` (#5676)
cranelift: Rewrite `(x>>k)<<k` into masking off the bottom `k` bits (#5673)
cranelift: constant propagate shifts (#5671)
cranelift: Add egraph rule to rewrite `x * C` to `x << log2(C)` when `C` is a power of two (#5647)
egraph opt rules: do `(icmp cc x x) == {0,1}` only for integer types. (#5438)
Optimize sign extension via shifts (#6220)
egraphs: Add `bmask` bit pattern optimization rule (#6196)
Add `multi_lane` precondition to `bitselect` => `{u,s}{min,max}` rewrite (#6201)
ISLE: simplify select/bitselect when both choices are the same (#6141)
Add egraph cprop optimizations for `splat` (#6148)
ISLE: rewrite loose inequalities to strict inequalities and strict inequalities to equalities (#6130)
ISLE: rewrite `and`/`or` of `icmp` (#6095)
cranef: add synonyms for all variations of `icmp` (#6081)
cranelf: rewrite `iabs(ineg(x))` and `iabs(iabs(x))` (#6072)
cranelf: rewrite `x*-1` to `ineg(x)` (#6052)
cranelf: cancel `ineg` when args to `imul` (#6053)
cranelf: simplify `icmp` against UMAX/SMIN/SMAX (#6037)
cranelf: simplify `x-x` to `0` (#6032)
cranelf: simplify `fneg(fneg(x))` to `x` (#6034)
cranelf: simplify `ineg(ineg(x))` to `x` (#6033)
Add egraph optimization for fneg's cancelling out (#5910)
cranelf: Generalize `(x << k) >> k` optimization (#5746)
cranelf: Optimize `select+icmp` into `{s,u}{min,max}` (#5546)
cranelf: Collapse double extends into a single extend (#5772)
Generalize and/or/xor optimizations (#5744)
Algebraic opts: Reuse `iconst 0` from LHS (#5724)
Add some minor souper-harvested optimizations (#5735)
cranelf: Only build iconst for ints <= 64 bits (#5723)
Legalize `b{and,or,xor}_not` into component instructions (#5709)
egraphs/cprop: Don't extend constants to `i128` (#5717)
Generalize u/sextend constant folding to all types (#5706)
cranelf: Correctly wrap shifts in constant propagation (#5695)
Constant-fold icmp instructions (#5666)
cranelf: Rewrite `or(and(x, y), not(y))` => `or(x, not(y))` (#5676)
cranelf: Rewrite `(x>>k)<<k` into masking off the bottom `k` bits (#5673)
cranelf: constant propagate shifts (#5671)
cranelf: Add egraph rule to rewrite `x * C` by `x << log2(C)` when `C` is a power of two (#5647)
egraph opt rules: do `(icmp cc x x) == {0,1}` only for integer types. (#5438)
Project Health & Enablement

```latex
+ ;; A reduction-of-an-extend back to the same original type is the same as not
+ ;; actually doing the extend in the first place.
+ (rule (simplify (ireduce ty (sextend _ x @(value_type ty)))) x)
+ (rule (simplify (ireduce ty (uextend _ x @(value_type ty)))) x)
```

Nobody would take the time to write a manual pass to do that!
Performance: Qualitative Discussion

Q: How did we achieve near-parity?
Performance: Qualitative Discussion

Q: How did we achieve near-parity?
A: By doing nearly the same amount of work!
Q: *How did we achieve near-parity?*

A: By doing nearly the same amount of work!

- E-graph interning \(\approx\) GVN
- E-nodes are stored as instructions (same data structure)
- Initially, rewrites in egraph are equivalent to old pipeline
Q: How did we achieve near-parity?

A: By doing nearly the same amount of work!

• Differences: code placement (reconstruct all vs. incremental) multi-version (selection, rewrite multiple paths)
Performance: Qualitative Discussion

Q: How did we achieve near-parity?
A: By doing nearly the same amount of work!

“Pay as you go” is crucial for incremental adoption!
Possible Future Plans

• Instruction selector as extraction pass
  • We have left-hand-side patterns for what the ISA can do efficiently
  • Why not lower directly from eclasses?
  • Somewhat complex interactions with scoped elaboration + pass direction
Possible Future Plans

• Optimization through block parameters (phi-nodes)
  • Sparse conditional constant propagation! Unify branch-folding + const-prop
  • Challenge: deal with cycles
  • Are there limited forms that operate in a single pass? (skip if backedge?)
Possible Future Plans

• Non-greedy instruction selection
  • We do extraction before elaboration
  • Optimal extraction depends on elaboration:
    • multiple uses of a value can “share” its cost
    • if another inst needs a value that is expensive, it becomes sunk cost
Possible Future Plans

• Fused / unrolled rewrites
  • We have efficient rule dispatch (decision tree), but only one step at a time
  • Can we statically unroll a path of rewrites?
  • … and even elide insertion of intermediates if we know they’re “bad” (more expensive, always subsumed)?
Possible Future Plans

• Instruction scheduling
  • The ægraph throws away location information
  • Scoped elaboration recomputes it
  • The “as late as possible” schedule that results is often quite bad
  • Heuristics from (i) register pressure, (ii) original code order, (iii) other?
1. Why we want a rewrite-based optimizer
2. How to turn a CFG into an egraph and back again
3. Cycles why they occur, and what to do about them
4. Results how well does it work?
5. Lessons in translating research to production
1. Why we want a rewrite-based optimizer
2. How to turn a CFG into an egraph and back again
3. Cycles why they occur, and what to do about them
4. Results how well does it work?
5. Lessons in translating research to production
Efficiency

• This is *really important* in production software

• Every percentage point counts: 1% might cost an engineer-month to regain; and costs a lot operationally at scale
Efficiency

• This is *really important* in production software

  • Every percentage point counts: 1% might cost an engineer-month to regain; and costs a lot operationally at scale

• There is inherent tension w.r.t. moving fast + experimenting to “just get results”, but “algorithm is robustly fast” is its own kind of result too
Efficiency

• This is *really important* in production software

• Every percentage point counts: 1% might cost an engineer-month to regain; and costs a lot operationally at scale

• There is inherent tension w.r.t. moving fast + experimenting to “just get results”, but “algorithm is robustly fast” is its own kind of result too

• Robustness/predictability is important (and distinct from “fast on average”)
Efficiency

• This is *really important* in production software
  
  • Every percentage point counts: 1% might cost an engineer-month to regain; and costs a lot operationally at scale
  
  • There is inherent tension w.r.t. moving fast + experimenting to “just get results”, but “algorithm is robustly fast” is its own kind of result too
  
  • Robustness/predictability is important (and distinct from “fast on average”)
  
  • We (practicing software engineers) need to do a better job of documenting “all the usual tricks”!
Limits Induce Creativity

- This work in Cranelift started with “standard” e-graphs and egg
- When it wasn’t fast enough, I could have stopped and moved on!
- Requires the “correct” amount of unjustified optimism
Limits Induce Creativity

• “Bottom-up” vs. “top-down” thinking
• “I want to do eqsat” —> optimize all the computation needed for this, vs…
  • I tried this first!
• … “I have N linear passes” —> which ideas can I keep?
Tradeoffs... and Incrementalism

- It’s OK to not solve the entire problem!
  - The only real requirement is that we run the program correctly*
  - Sometimes “this is the best point on the effort Pareto curve” and we’re done
Tradeoffs... and Incrementalism

• It’s OK to not solve the entire problem!
  • The only real requirement is that we run the program correctly*
• Sometimes “this is the best point on the effort Pareto curve” and we’re done
• Sometimes, we can come up with better ideas later
• And this happens all the time in Cranelift
• View the codebase as a living, evolving understanding of problem domain
Tradeoffs... and Incrementalism

- *Design for incrementalism* by:
  - Building frameworks (rewrite language/infra, ...)
  - Building guardrails (good testing, typesafe abstractions, well-documented invariants)
Tradeoffs... and Incrementalism

- Design for incrementalism by:
  - Building frameworks (rewrite language/infra, ...)
  - Building guardrails (good testing, typesafe abstractions, well-documented invariants)
  - Accept limits and ship, then fulfill last 20% of needs while plane is flying
Community Leverage Multipliers

• Let’s talk about “design for ___” a bit more
Community Leverage Multipliers

• Let’s talk about “design for ___” a bit more

  • Design for community: find abstractions that allow modular, typesafe work and enable many uses (verification!)
Community Leverage Multipliers

• Let’s talk about “design for ___” a bit more
  • Design for community: find abstractions that allow modular, typesafe work and enable many uses (verification!)
  • We picked up the e-graph idea because
    • It’s a clean abstraction
    • It allows modular, easy contributions of mid-end optimizations
    • It bridges the gap with academia a bit and pulls in new ideas
E-graphs... in Industry?

- Isn’t this bona-fide research? Am I not a software engineer in... industry?
• Isn’t this bona-fide research? Am I not a software engineer in... industry?
• Secret: software engineering is full of research problems
  • Caveat: pick a domain like compilers
• Different kinds of problems with different considerations
E-graphs... in Industry?

• Isn’t this bona-fide research? Am I not a software engineer in... industry?
  • Secret: software engineering is full of research problems
    • Caveat: pick a domain like compilers
  • Different kinds of problems with different considerations
  • Different approach to risk; later in pipeline, less speculative
    • (thank you for exploring e-graphs first!)
E-graphs... in Industry?

- Research is *totally* relevant to industry if it addresses industry’s needs: robust, reliable, simple, reliable, fast, reliable
E-graphs... in Industry?

- Research is *totally* relevant to industry if it addresses industry’s needs: robust, reliable, simple, reliable, fast, reliable
- Industry sometimes presents opportunities to rethink key infra (e.g. compiler)
  - It can be hard to convincingly make a case for this in a vacuum in academia
  - But good reasons exist (security, simplicity, agility, ...)!
E-graphs... in Industry?

- Research is *totally* relevant to industry if it addresses industry’s needs: robust, reliable, simple, reliable, fast, reliable

- Industry sometimes presents opportunities to rethink key infra (e.g. compiler)
  - It can be hard to convincingly make a case for this in a vacuum in academia
  - But good reasons exist (security, simplicity, agility, ...)

- Academia is idea-rich and searches for problems/motivations;
  Industry is problem-rich and searches for ideas/solutions

- Bridging the two is incredibly fruitful and rewarding!
Work with Cranelift!

- We love mentoring students and collaborating with researchers
  - Verification (VeriSL, VeriWasm, ...); chaos-mode randomized testing; exceptions; typed func-refs; e-graph-based fuzzing mutators; extensions of custom DSLs; ...
  
- There are many open problems and the need to solve them is immediate and directly motivated

- It’s how we can work “smarter not harder” and keep in the game, as an underdog — we all win!
Thanks!

• Links
  • https://cranelift.dev/
  • https://bytecodealliance.zulipchat.com/
  • https://cfallin.org/