Superoptimization of WebAssembly Process Graphs

By: Dennis Sprokholt

Contents

Contents

Superoptimization: Program Optimization

- Peephole Optimizations[1]:
  - `y := 0` ⇒ `y := y XOR y`
  - `y := x+x` ⇒ `y := x << 1`
  - `y := x+1` ⇒ `y := -~x`

[1]Warren, H.S. (2013). *Hacker's delight*. Pearson Education.

Superoptimization - Optimizing Compilers

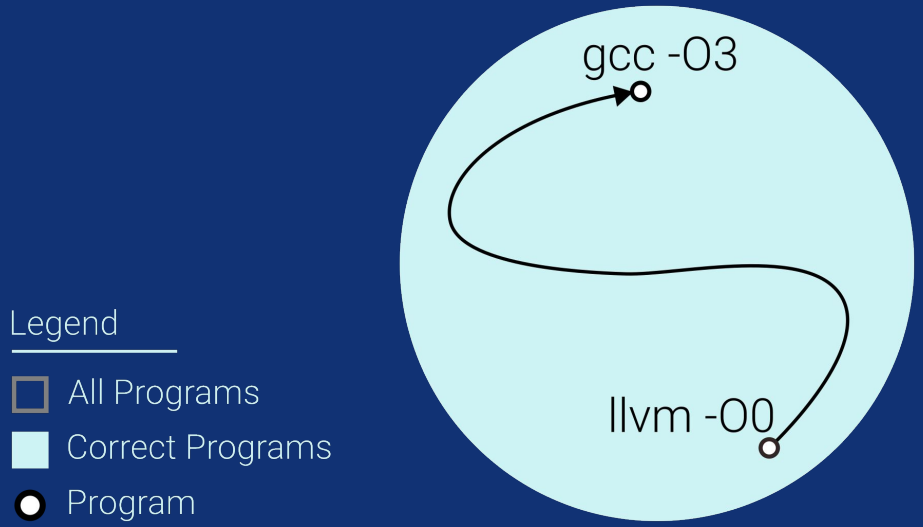


Figure adapted from: Schkufza, E., Sharma, R., & Aiken, A. (2013). Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, *41*(1), 305-316.

Superoptimization - Non-obvious Optimization

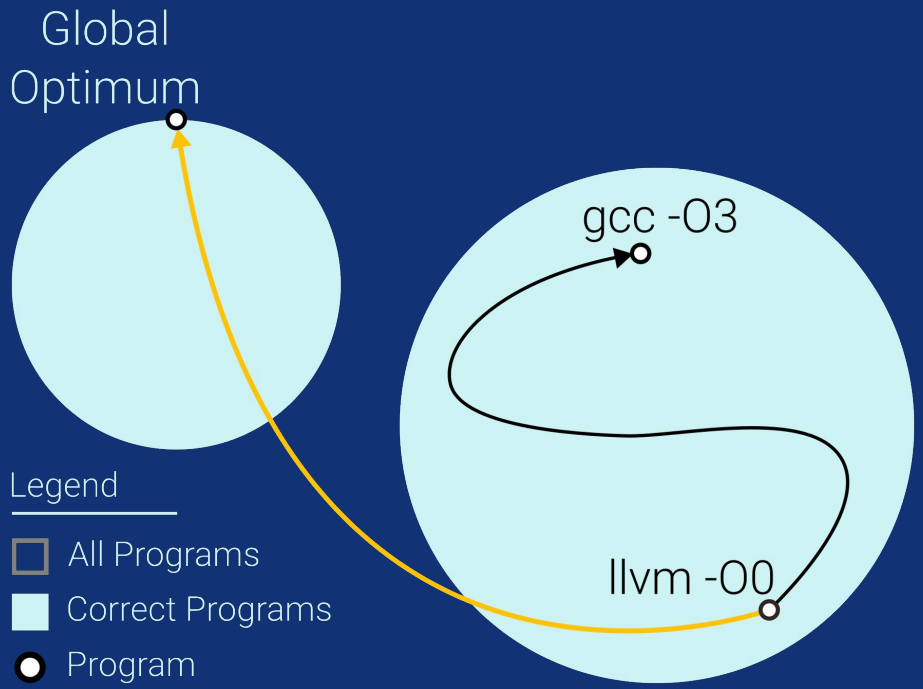


Figure adapted from: Schkufza, E., Sharma, R., & Aiken, A. (2013). Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, *41*(1), 305-316.

Superoptimization - Enumerative Search

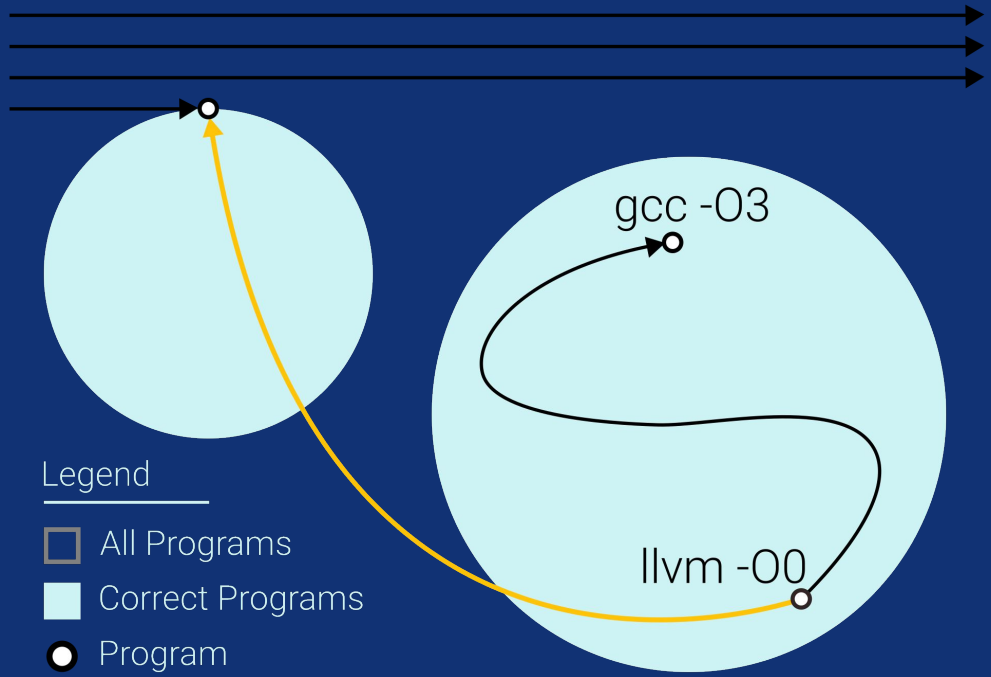


Figure adapted from: Schkufza, E., Sharma, R., & Aiken, A. (2013). Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, *41*(1), 305-316.

Superoptimization - Enumerative Search Issues

- There are many programs:
  - Assume 50 available instructions: 50^d programs of size d
    - Program of size 47:

      $50^{47} \approx 10^{80} \approx$ Number of atoms in the universe

- Program Equivalence Checking is Undecidable
  - Preserve input/output relation
  - Includes side-effects

Superoptimization - Existing Solutions

- Superoptimize many *small* fragments in a program
- Sliding window
- Search space pruning
- Stochastic traversal

Contents

Research Goals

- “Superoptimize” larger control flow structures
  - Loops

Research Goals: Examples

- Count 1-bits in 32-bit word

```
fn popcount( mut x: u32 ) -> u32 {
  let mut count = 0;
  while x != 0 {
    if ( x & 1 ) != 0 {
      count += 1;
    }
    x >>= 1;
  }
  return count;
}
```

- Many ISAs have: `popcnt32` instruction

Contents

WebAssembly

- Superoptimize WebAssembly programs
- Targets the Web
- *Abstraction* over machine code

- Secure (isolated address space)
- Compact (stack machine)
- Low-level (fast)
- Portable

Contents

Process Graphs - SMT Solvers

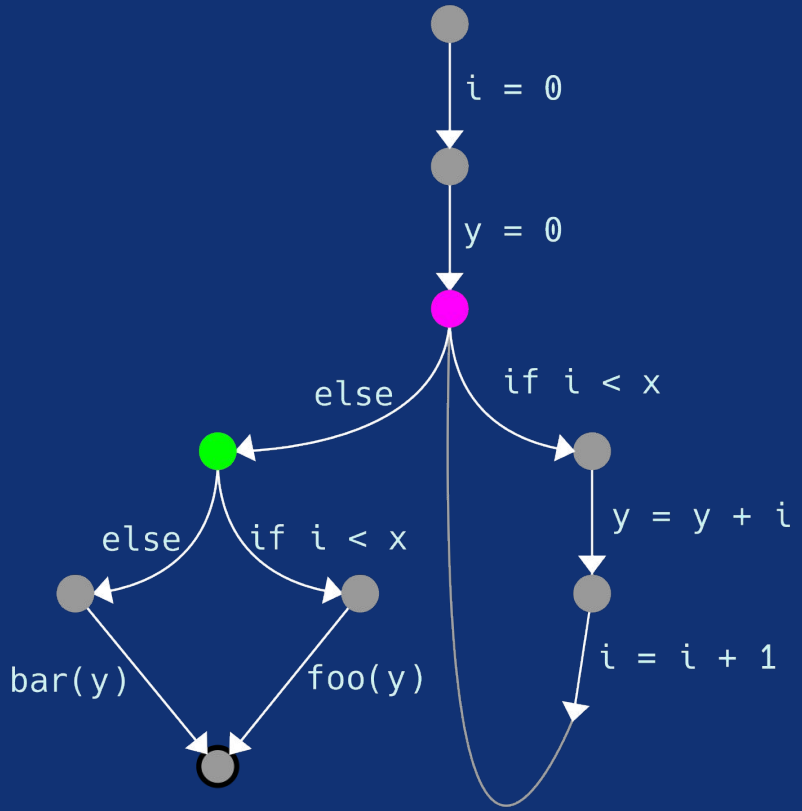
- SAT Solvers
  - $P \vee Q \Rightarrow \{P \mapsto T, Q \mapsto T\}$
  - $P \wedge \neg P \Rightarrow UNSAT$

- SMT Solvers
  - Satisfiability Modulo Theories
  - $x < y \wedge y < 10 \Rightarrow \{x \mapsto 3, y \mapsto 5\}$
  - $10 < x \wedge x < 3 \Rightarrow UNSAT$

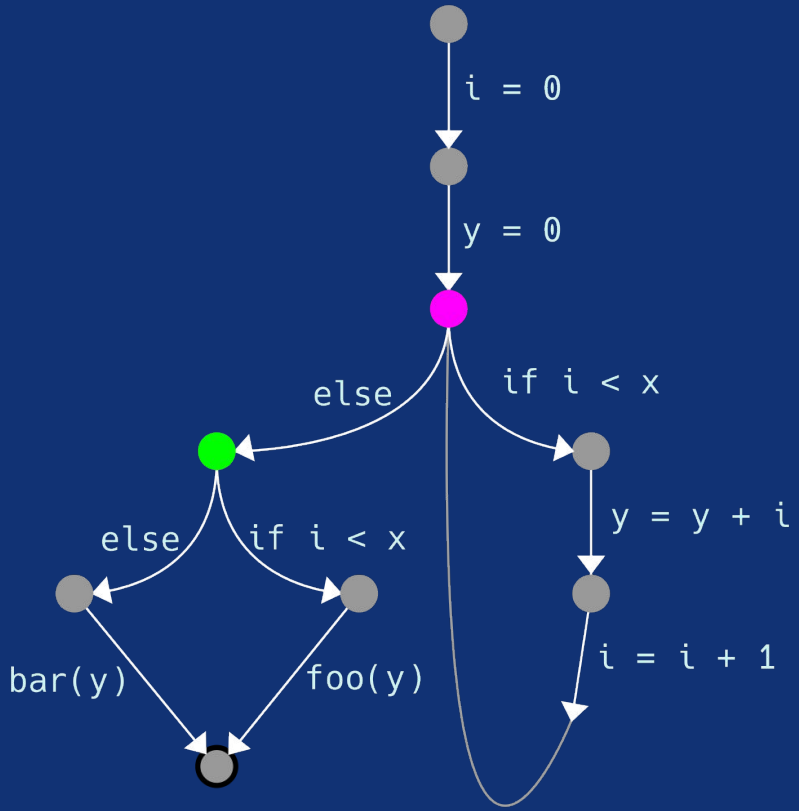
- Z3

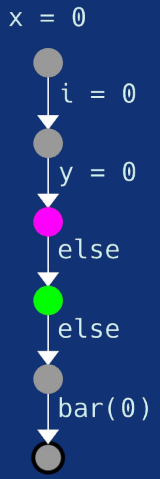
Process Graphs

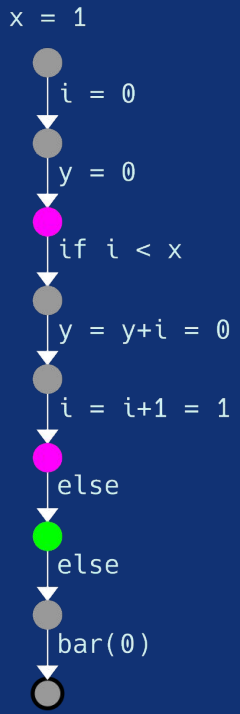
```
fn f( x: u32 ) {
  let mut i = 0;
  let mut y = 0;
  while i < x {
    y = y + i;
    i = i + 1;
  }
  if i < x {
    foo( y );
  } else {
    bar( y );
  }
}
```

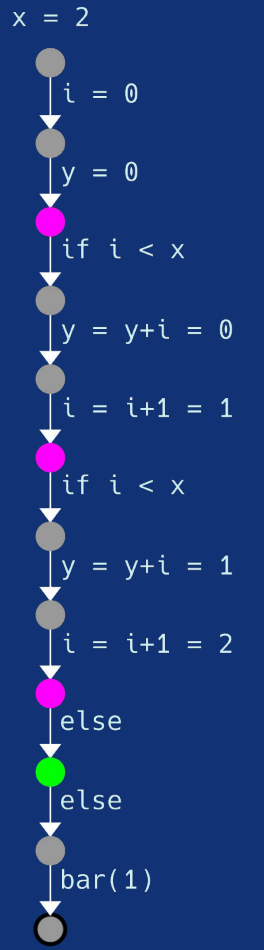


Process Graphs - Concrete Execution







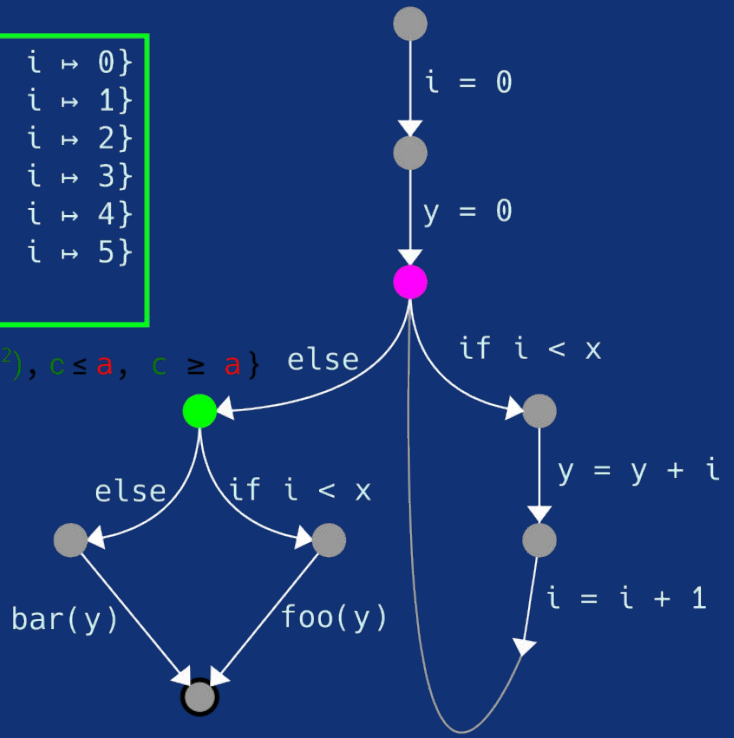


Process Graphs - Configurations

$\{\{x \mapsto a, y \mapsto b, i \mapsto c\} \mid a \in [0..2^{32}), b \in [0..2^{32}), c \in [0..2^{32})\}$

```
{ {x ↦ 0, y ↦ 0,  i ↦ 0}
, {x ↦ 1, y ↦ 0,  i ↦ 1}
, {x ↦ 2, y ↦ 1,  i ↦ 2}
, {x ↦ 3, y ↦ 3,  i ↦ 3}
, {x ↦ 4, y ↦ 6,  i ↦ 4}
, {x ↦ 5, y ↦ 10, i ↦ 5}
, ... }
```

$\{\{x \mapsto a, y \mapsto b, i \mapsto c\} \mid a \in [0..2^{32}), b = \Sigma_{j=0}^{c-1} j, c \in [0..2^{32}), c \leq a, c \geq a\}$



Process Graphs - Configurations

$\{\{x \mapsto a, y \mapsto b, i \mapsto c\} \mid a \in [0..2^{32}), b \in [0..2^{32}), c \in [0..2^{32})\}$

```
{ {x ↦ 0, y ↦ 0,  i ↦ 0}
, {x ↦ 1, y ↦ 0,  i ↦ 1}
, {x ↦ 2, y ↦ 1,  i ↦ 2}
, {x ↦ 3, y ↦ 3,  i ↦ 3}
, {x ↦ 4, y ↦ 6,  i ↦ 4}
, {x ↦ 5, y ↦ 10, i ↦ 5}
, ... }
```

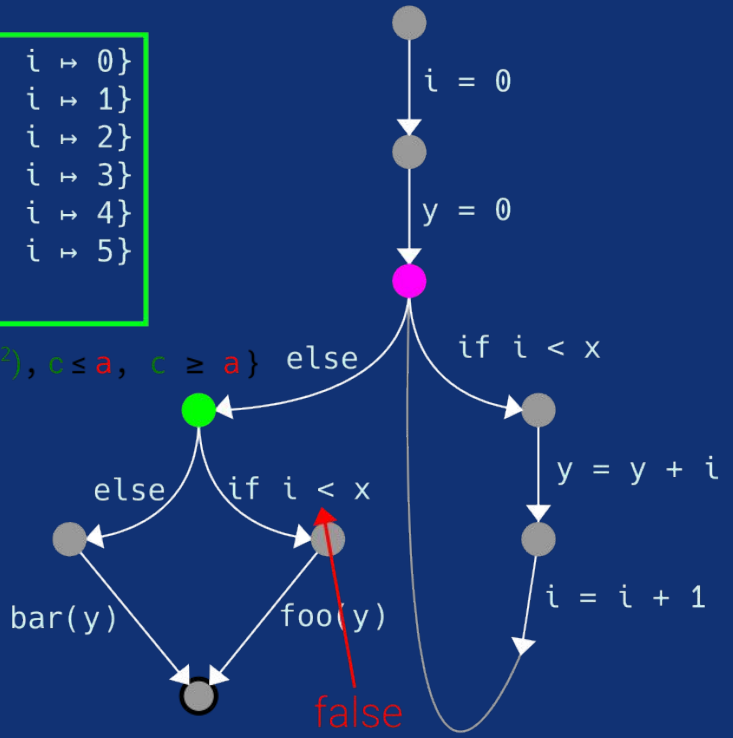
$\{\{x \mapsto a, y \mapsto b, i \mapsto c\} \mid a \in [0..2^{32}), b = \Sigma_{j=0}^{c-1} j, c \in [0..2^{32}), c \le a, c \ge a\}$

$p = (c < a) \wedge a \in [0..2^{32}) \wedge c \in [0..2^{32}) \wedge c \le a \wedge c \ge a$

Z3: $\{ p \mapsto \bot \}$

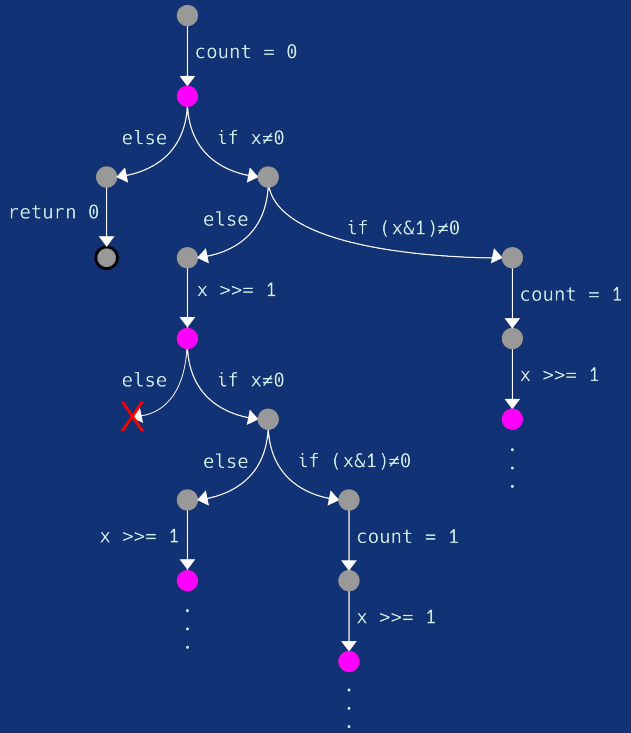
$p = (c < a) \wedge a \in [0..2^{32}) \wedge c \in [0..2^{32}) \wedge c \le a \wedge c \ge a \wedge p \neq \bot$

Z3: UNSAT



Process Graphs - Process Trees & Driving

- Popcount:



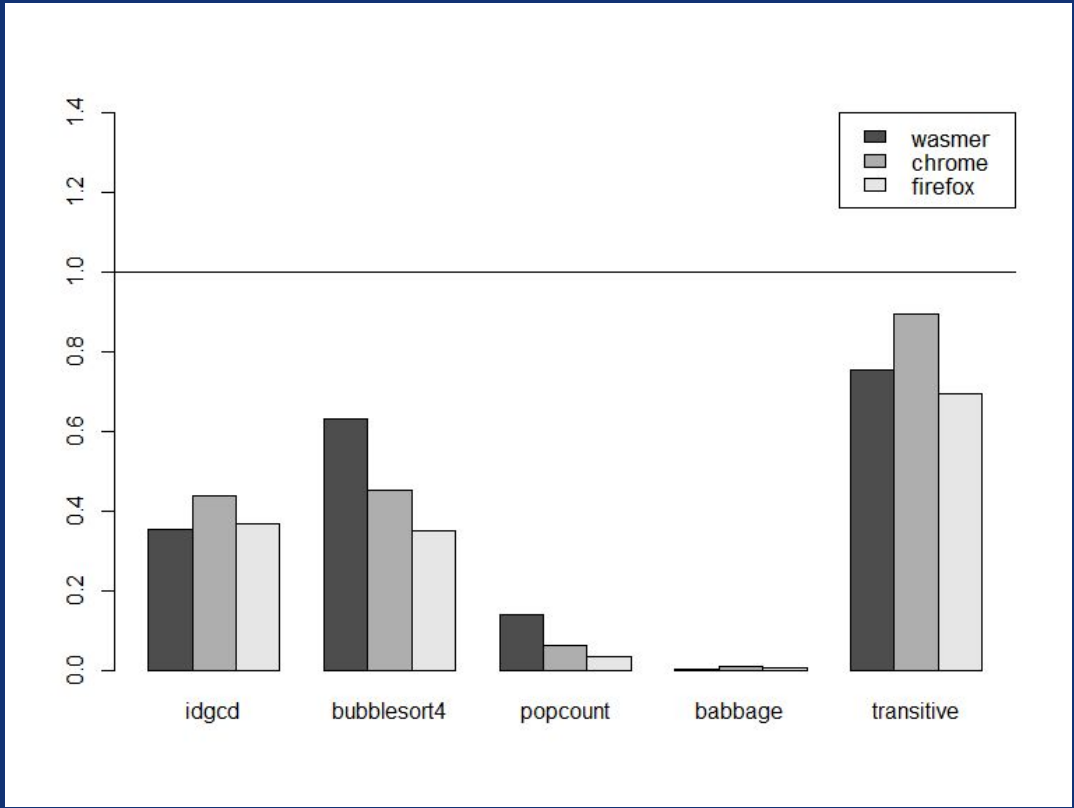
- Driving: Simulate execution with partial knowledge of the input
- Expand the graph into a tree
  - Eliminate unreachable branches
  - Replace constants
- When all infinite branches are eliminated, the tree is *finite*
- For popcount: Observe at most 32 iterations

Process Graphs - Synthesis

- Extract properties from the instructions in the *finite* tree
  - Types of arithmetic instructions (e.g., 32-bit ints, 64-bit ints)
  - Memory Operations
  - Called functions
  - . . .
- Brute force for some time (and timeout)
  - Only *linear* instructions sequences (no backward/forward branches)

Contents

Results - Small Artificial Programs



Results - Large Programs

File	Timeout	Constants Replaced	Branches Eliminated	Time Taken	Output File Size %
`bitwise_IO`	1000ms	5 / 394	1 / 43	3 sec	99.56%
`lua_mini`	1000ms	6 / 1,280	0 / 78	~ 2 min	99.76%
`raytracer`	200ms	N/A	29 / 2,277	~ 2 min	99.48%
`raytracer`	200ms	115 / 27,682	30 / 2,277	~ 30 min	99.35%
`lua`	200ms	N/A	15 / 5,125	~ 4 min	99.78%
`lua`	200ms	47 / 48,383	15 / 5,125	~ 45 min	99.75%
`z3`	200ms	N/A	803 / 487,686	~ 10 hours	99.06%
`z3` (aborted)	50ms	807 / 2,086,551	437 / 262,036	~ 20 hours	N/A

Results - Large Program Partial Evaluation

- Conditional Zero Constant. Hard to find without SMT solver

```
a = mem[ x ];
b = a & 0xFF;
if a != 0 {
  b = 0;
  ...
} else {
  a = b;
}
// b dead
```

```
a = mem[ x ];
if a != 0 {
  b = 0;
  ...
} else {
  a = 0;
}
// b dead
```

Contents

Conclusion

- Convert programs into process graphs
  - Store symbolic information at the nodes
  - Partial evaluation with SMT solver works (if you can spare the optimization time)
  - Currently only ~1% improvements

- Driving trees with an SMT solver *may be* a good idea
  - Great results on small (artificial) programs
  - Currently too costly for larger programs

Contents

Future Work

- Find profitable fragments in larger programs (in reasonable time)
  - Heuristic (on static properties)
  - Profiling

- Abstract Interpretation

End - Questions?

Comic: https://xkcd.com/303/

Extra - Abstract Interpretation

```
if a > b {
  if b > c {
    let x = ( a <= c ); // Always false. Tell constant propagation
    …
  }
}
```

Extra - Bubblesort