Superoptimization of WebAssembly Process Graphs

By: Dennis Sprokholt
Contents

- Superoptimization
- Research Goals
- WebAssembly
- Process Graphs
- Results
- Conclusion
- Future Work
Superoptimization: Program Optimization

- Peephole Optimizations\(^1\):

\[
\begin{align*}
\text{o} & \quad y := 0 \quad \Rightarrow \quad y := y \ XOR \ y \\
\text{o} & \quad y := x+x \quad \Rightarrow \quad y := x \ll 1 \\
\text{o} & \quad y := x+1 \quad \Rightarrow \quad y := \neg x
\end{align*}
\]

Superoptimization - Optimizing Compilers

Superoptimization - Non-obvious Optimization

Superoptimization - Enumerative Search

Superoptimization - Enumerative Search Issues

- There are many programs:
  - Assume 50 available instructions: \(50^d\) programs of size \(d\)
    - Program of size 47:
      \[ 50^{47} \approx 10^{80} \approx \text{Number of atoms in the universe} \]

- 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
Research Goals

● “Superoptimize” larger control flow structures
  ○ Loops
Research Goals: Examples

- Count 1-bits in 32-bit word

```rust
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
WebAssembly

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

- Secure (isolated address space)
- Compact (stack machine)
- Low-level (fast)
- Portable
Process Graphs - SMT Solvers

- SAT Solvers
  - $P \lor Q \implies \{P \mapsto T, \ Q \mapsto T\}$
  - $P \land \lnot P \implies \text{UNSAT}$

- SMT Solvers
  - Satisfiability Modulo Theories
    - $x < y \land y < 10 \implies \{x \mapsto 3, \ y \mapsto 5\}$
    - $10 < x \land x < 3 \implies \text{UNSAT}$

- Z3
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

```
x = 0
  i = 0
  y = 0
  else
    if i < x
      y = y + i
      i = i + 1
    else
      bar(y)
      foo(y)
  else
    if i < x
      y = y + i
      i = i + 1
    else
      bar(y)
      foo(y)

x = 1
  i = 0
  y = 0
  else
    if i < x
      y = y + i
      i = i + 1
    else
      bar(y)
      foo(y)

x = 2
  i = 0
  y = 0
  else
    if i < x
      y = y + i
      i = i + 1
    else
      bar(y)
      foo(y)
```
Process Graphs - Configurations

\{ \{ x \mapsto a, y \mapsto b, i \mapsto c \} | a, b, c \in [0..2^{32}] \}

\{ \{ x \mapsto 0, y \mapsto 0, i \mapsto 0 \},
\{ x \mapsto 1, y \mapsto 0, i \mapsto 1 \},
\{ x \mapsto 2, y \mapsto 1, i \mapsto 2 \},
\{ x \mapsto 3, y \mapsto 3, i \mapsto 3 \},
\{ x \mapsto 4, y \mapsto 6, i \mapsto 4 \},
\{ x \mapsto 5, y \mapsto 10, i \mapsto 5 \},
\ldots \}
Process Graphs - Configurations

\{ \{ x \mapsto a, y \mapsto b, i \mapsto \varepsilon \} \mid a \in [0..2^{32}], b \in [0..2^{32}], \varepsilon \in [0..2^{32}] \}\}

\{ \{ x \mapsto 0, y \mapsto 0, i \mapsto 0 \}, \\
\{ x \mapsto 1, y \mapsto 0, i \mapsto 1 \}, \\
\{ x \mapsto 2, y \mapsto 1, i \mapsto 2 \}, \\
\{ x \mapsto 3, y \mapsto 3, i \mapsto 3 \}, \\
\{ x \mapsto 4, y \mapsto 6, i \mapsto 4 \}, \\
\{ x \mapsto 5, y \mapsto 10, i \mapsto 5 \}, \ldots \}\}

p = (c < a) \land a \in [0..2^{32}] \land c \in [0..2^{32}] \land c \leq a \land c \geq a

Z3: \{ p \models \bot \}

p = (c < a) \land a \in [0..2^{32}] \land c \in [0..2^{32}] \land c \leq a \land c \geq a \land p \not\models \bot

Z3: UNSAT

\text{if } i < x \text{ then }
\text{else}
\text{if } i < x \text{ then } y = y + i \text{ end then }
\text{else}
bar(y) \text{ end then }
\text{else fals}\text{e end then }
i = i + 1
Process Graphs - Process Trees & Driving

- **Popcount:**
  - `count = 0`
  - `if x≠0`
  - `if (x&1)≠0`
  - `x >>= 1`
  - `count = 1`
  - `else`

- **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)
Results - Small Artificial Programs
<table>
<thead>
<tr>
<th>File</th>
<th>Timeout</th>
<th>Constants Replaced</th>
<th>Branches Eliminated</th>
<th>Time Taken</th>
<th>Output File Size %</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitwise_IO</td>
<td>1000ms</td>
<td>5 / 394</td>
<td>1 / 43</td>
<td>3 sec</td>
<td>99.56%</td>
</tr>
<tr>
<td>lua_mini</td>
<td>1000ms</td>
<td>6 / 1,280</td>
<td>0 / 78</td>
<td>~ 2 min</td>
<td>99.76%</td>
</tr>
<tr>
<td>raytracer</td>
<td>200ms</td>
<td>N/A</td>
<td>29 / 2,277</td>
<td>~ 2 min</td>
<td>99.48%</td>
</tr>
<tr>
<td>raytracer</td>
<td>200ms</td>
<td>115 / 27,682</td>
<td>30 / 2,277</td>
<td>~ 30 min</td>
<td>99.35%</td>
</tr>
<tr>
<td>lua</td>
<td>200ms</td>
<td>N/A</td>
<td>15 / 5,125</td>
<td>~ 4 min</td>
<td>99.78%</td>
</tr>
<tr>
<td>lua</td>
<td>200ms</td>
<td>47 / 48,383</td>
<td>15 / 5,125</td>
<td>~ 45 min</td>
<td>99.75%</td>
</tr>
<tr>
<td>z3</td>
<td>200ms</td>
<td>N/A</td>
<td>803 / 487,686</td>
<td>~ 10 hours</td>
<td>99.06%</td>
</tr>
<tr>
<td>z3 (aborted)</td>
<td>50ms</td>
<td>807 / 2,086,551</td>
<td>437 / 262,036</td>
<td>~ 20 hours</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Results - Large Program Partial Evaluation

- Conditional Zero Constant. Hard to find without SMT solver

```c
a = mem[ x ];

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

```c
a = mem[ x ];

if a != 0 {
    b = 0;
    ...
} else {
    a = 0;
}
// b dead
```
Contents

● Superoptimization
● Research Goals
● WebAssembly
● Process Graphs
● Results
● Conclusion
● Future Work
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

- Superoptimization
- Research Goals
- WebAssembly
- Process Graphs
- Results
- Conclusion
- Future Work
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/
if a > b {
    if b > c {
        let x = ( a <= c ); // Always false. Tell constant propagation
        ...
    }
}

Extra - Bubblesort