Leveraging LLVM Optimizations to Speed up Constraint Solving

Benjamin Mikek
10 April 2024
Vienna, Austria
Background: SMT Constraints

• SMT (Satisfiability Modulo Theories) constraints encode first-order logic problems
• Symbolic execution (KLEE) generates SMT constraints
• Alive [PLDI15, SAS16, PLDI21] uses SMT to verify LLVM optimizations
• Many advanced solvers: Z3, CVC5, Boolector, Bitwuzla, Yices, etc.
• Theories for bitvectors, floating-point, integers, real numbers, etc.
Example: Can multiplication overflow?

```lisp
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (= (_ extract 63 32) (bvmul ((_ zero_extend 32) a) ((_ zero_extend 32) b)) #x00000000)))
(assert (bvuge (bvul (bvlw #x0000000000000000 a) b)))
(check-sat)
```

Example: Can multiplication overflow?

Is this constraint satisfiable?

```
1 (declare-fun a () (_ BitVec 32))
2 (declare-fun b () (_ BitVec 32))
3 (assert (not (= 
4     ((_ extract 63 32)
5         (bvmul ((_ zero_extend 32) a)
6             ((_ zero_extend 32) b)))
7     #x00000000)))
8 (assert (bvue #x00000000))
9 (check-sat)
```

Example: Can multiplication overflow?

```
1 (declare-fun a () (_ BitVec 32))
2 (declare-fun b () (_ BitVec 32))
3 (assert (not (= 
4   ((_ extract 63 32)
5     (bvmul ((_ zero_extend 32) a)
6       ((_ zero_extend 32) b)))
7     #x00000000)))
8 (assert (bvuge (bvdiv #x7fffffff a) b))
9 (check-sat)
```

Example: Can multiplication overflow?

```smt
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (= 
   ((_ extract 63 32)
     (bvmul ((_ zero_extend 32) a)
     ((_ zero_extend 32) b))))
   #x00000000)))
(assert (bvuge (bvuliv #xffffffff a) b))
(check-sat)
```


```
a = 1 0 1 1
b = 0 1 0 0

\[ \begin{array}{cccc}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 \\
\end{array} \times \begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 \\
\end{array} = \begin{array}{cccc}
0 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
```

\[ \begin{array}{cccc}
0 & 0 & 1 & 0 \\
\end{array} = \begin{array}{cccc}
0 & 0 & 0 & 0 \\
\end{array} \]
Example: Multiplication Overflow

Is this constraint satisfiable?
Example: Multiplication Overflow

Is this constraint satisfiable? No!

```
1 (declare-fun a () (_ BitVec 32))
2 (declare-fun b () (_ BitVec 32))
3 (assert (not (= 
4     ((_ extract 63 32)
5       (bvmul ((_ zero_extend 32) a) 
6         ((_ zero_extend 32) b)))
7       #x00000000))
8 (assert (bvuge (bvulev #x7fffffff a) b))
9 (check-sat)
```

Example: Multiplication Overflow

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (= 
  ((_ extract 63 32) 
    (bvmul ((_ zero_extend 32) a) 
             ((_ zero_extend 32) b))))
  #x00000000))
(assert (bvuge (bvul  #xfffffffff a) b))
(check-sat)
```


Is this constraint satisfiable?

But … z3 takes 10 minutes to solve it 😞
Example: Multiplication Overflow

```
1 (declare-fun a () (_ BitVec 32))
2 (declare-fun b () (_ BitVec 32))
3 (assert (not (=
4     ((_ extract 63 32)
5     (bvmul ((_ zero_extend 32) a)
6     ((_ zero_extend 32) b)))
7     #x00000000)))
8 (assert (bvuge (bvdiv #x00000000 a) b))
9 (check-sat)
```

> z3 complex.smt2
unsat

Takes 595 seconds 😞

> z3 simple.smt2
unsat

Takes 0.02 seconds 😊
Possible ✓
Already exists ✓
# SMTLIB and LLVM IR

<table>
<thead>
<tr>
<th></th>
<th>SMTLIB</th>
<th>LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitvector types</td>
<td>One for each integer width n</td>
<td>One for each width up to $2^{23}$</td>
</tr>
<tr>
<td>Floating point types</td>
<td>One for each integer pair eb, sb, but almost all in powers of 2</td>
<td>16-bit, 32-bit, 64-bit, 128-bit</td>
</tr>
<tr>
<td>Logic operations</td>
<td>and, or, xor, not, ite,=&gt;, ...</td>
<td>and, or, xor, select, ...</td>
</tr>
<tr>
<td>Bitvector math operations</td>
<td>bvadd, bvsb, bvmul, bvsdiv, bvudiv, ...</td>
<td>add, sub, mul, sdiv, udiv</td>
</tr>
<tr>
<td>Floating point math operations</td>
<td>fp.add, fp.sub, fp.div, fp.fma, ...</td>
<td>fadd, fsub, fdiv, llvm.fma, ...</td>
</tr>
<tr>
<td>Conversions</td>
<td>to_fp, to_fp_unsigned, fp.to_sbv, ...</td>
<td>sitofp, uito_fp, fptosi, ...</td>
</tr>
</tbody>
</table>
SLOT: Key Challenges

• SLOT has two parts: a front end and back end. Both have to preserve semantics
• LLVM is missing some SMT operations
• SMTLIB is missing some LLVM operations
• SMT constraints are *declarative*; LLVM is *imperative*
SLOT Translation

• Frontend: traverse the syntax tree of each SMT assertion
• Build an LLVM expression with the same semantics
• Most operations have 1-to-1 equivalents
  • bvmul -> mul, bvadd -> add, fp.add -> fadd
• Some expressions are more complex and may involve undefined behavior handling
SLOT by Example: Checking for Overflow

1 (declare-fun a () (_ BitVec 32))
2 (declare-fun b () (_ BitVec 32))
3 (assert (not (= 
4     ((_ extract 63 32) 
5         (bvmul ((_ zero_extend 32) a) 
6             ((_ zero_extend 32) b)))))
7     #x00000000))
8 (assert (bvuge (bvudiv #xffffffff a) b))
9 (check-sat)
The LLVM function returns true if the inputs satisfy the underlying constraint.

```assembly
define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 1, i32 %7
  %10 = icmp uge i32 %9, %b
  %11 = and i1 %6, %10
  ret i1 %11
}
```
Optimization

instcombine

```c
define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 -1, i32 %7
  %10 = icmp uge i32 %9, %b
  %11 = and i1 %6, %10
  ret i1 %11
}
```
Optimization

instcombine, gvn

define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 -1, i32 %7
  %10 = icmp uge i32 %9, %b
  %11 = and i1 %6, %10
  ret i1 %11
}

define i1 @SMT(i32 %a, i32 %b) {
  %b.fr = freeze i32 %b
  %umul = call { i32, i1 } @llvm.umul.with.overflow.i32(i32 %a, i32 %b.fr)
  %1 = extractvalue { i32, i1 } %umul, 1
  %mul.not.ov = xor i1 %1, true
  ret i1 false
}
Optimization

instcombine, gvn, instcombine

```c
1 define i1 @SMT(i32 %a, i32 %b) {
2   %0 = zext i32 %b to i64
3   %1 = zext i32 %a to i64
4   %2 = mul i64 %1, %0
5   %3 = lshr i64 %2, 32
6   %4 = trunc i64 %3 to i32
7   %5 = icmp eq i32 %4, 0
8   %6 = xor i1 %5, true
9   %7 = udiv i32 -1, %a
10  %8 = icmp eq i32 %a, 0
11  %9 = select i1 %8, i32 -1, i32 %7
12  %10 = icmp uge i32 %9, %b
13  %11 = and i1 %6, %10
14  ret i1 %11
15 }
```
Backend Translation

```c
1 define i1 @SMT(i32 %a, i32 %b) {
2     ret i1 false
3 }
```

```
1 (assert false)
2 (check-sat)
```

Can be solved almost instantly (0.02 seconds)!
Results

• Three logics: bitvectors, floating-point, and mixed
• Three solvers: Z3, CVC5, Boolector
• Three questions:
  • Can SLOT solve constraints for which solvers time out?
  • How much does SLOT speed up solving?
  • Which compiler optimization passes contribute?
Results

• SLOT increases the number of solvable constraints at 600 second timeouts substantially with all solvers:
  • 9-14% for floating-point
  • 32-67% for mixed
  • 15-18% for bitvectors

• SLOT can solve hundreds of constraints for which all existing solvers time out
Results

Floating-point

Mixed

Bitvector
Optimization Pass Contributions

• SMT constraints are simpler than programs
• All a single function (intraprocedural) [-10 passes]
• No memory operations [-7 passes]
• No branching (all one basic block) [-17 passes]
• Other reasons (debug info, backends, etc) [-16 passes]
• We disable vectorization, since this slows down solving
• 8 relatively simple passes are left
Optimization Pass Contributions

• The most effective optimizations passes are reassociate, instcombine, and global value numbering

• Solver developers can learn from these results about new optimizations to include in solvers

How many benchmarks does each pass change?

<table>
<thead>
<tr>
<th>Pass</th>
<th>QF_FP</th>
<th>QF_BVFP</th>
<th>QF_BV</th>
</tr>
</thead>
<tbody>
<tr>
<td>instcombine</td>
<td>99%</td>
<td>100%</td>
<td>78%</td>
</tr>
<tr>
<td>reassociate</td>
<td>78%</td>
<td>57%</td>
<td>26%</td>
</tr>
<tr>
<td>gvn</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
<td>43%</td>
</tr>
<tr>
<td>sccp</td>
<td>0%</td>
<td>&lt;1%</td>
<td>17%</td>
</tr>
<tr>
<td>dce</td>
<td>0%</td>
<td>&lt;1%</td>
<td>17%</td>
</tr>
<tr>
<td>instsimplify</td>
<td>0%</td>
<td>&lt;1%</td>
<td>16%</td>
</tr>
<tr>
<td>aggressive-instcombine</td>
<td>0%</td>
<td>0%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>adce</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Which passes contribute the most speedup?

<table>
<thead>
<tr>
<th>Pass</th>
<th>Count</th>
<th>Speedup without</th>
<th>Speedup with</th>
<th>Spread</th>
</tr>
</thead>
<tbody>
<tr>
<td>reassociate</td>
<td>2,168</td>
<td>1.58×</td>
<td>2.02×</td>
<td>0.44</td>
</tr>
<tr>
<td>instcombine</td>
<td>4,031</td>
<td>1.49×</td>
<td>1.83×</td>
<td>0.34</td>
</tr>
<tr>
<td>gvn</td>
<td>3,816</td>
<td>1.51×</td>
<td>1.85×</td>
<td>0.34</td>
</tr>
<tr>
<td>instsimplify</td>
<td>1,562</td>
<td>1.74×</td>
<td>1.75×</td>
<td>0.22</td>
</tr>
<tr>
<td>sccp</td>
<td>1,360</td>
<td>1.71×</td>
<td>1.86×</td>
<td>0.15</td>
</tr>
<tr>
<td>dce</td>
<td>1,705</td>
<td>1.78×</td>
<td>1.68×</td>
<td>−0.10</td>
</tr>
<tr>
<td>agg-instcombine</td>
<td>8</td>
<td>1.75×</td>
<td>1.22×</td>
<td>−0.53</td>
</tr>
</tbody>
</table>
Discussion

• What is simpler in the compiler context is not always simpler in the SMT context
• Solvers and SLOT form a sieve under portfolio methodology
• All presented results included the overhead of translation. Overhead is substantial for small constraints, but grows more slowly than solving time

<table>
<thead>
<tr>
<th>Time Interval</th>
<th>Floating-point</th>
<th>Mixed</th>
<th>Bitvector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-30</td>
<td>32.21%</td>
<td>22.17%</td>
<td>48.24%</td>
</tr>
<tr>
<td>30-60</td>
<td>0.02%</td>
<td>0.16%</td>
<td>3.96%</td>
</tr>
<tr>
<td>60-120</td>
<td>0.01%</td>
<td>0.20%</td>
<td>1.67%</td>
</tr>
<tr>
<td>120-300</td>
<td>0.01%</td>
<td>0.09%</td>
<td>2.01%</td>
</tr>
<tr>
<td>300-600</td>
<td>0.02%</td>
<td>0.01%</td>
<td>2.84%</td>
</tr>
</tbody>
</table>
Extending to Unbounded Theories

- SMTLIB also defines (mathematical) integers and real
- Integers can be converted to bitvectors
- Handling these theories requires bound inference
- Selected bounds will not always be enough, so we underapproximate and verify

<table>
<thead>
<tr>
<th>Logic</th>
<th>Decidable?</th>
<th>Theoretically Bounded?</th>
<th>Practically Bounded?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Integer Arithmetic</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Nonlinear Integer Arithmetic</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Linear Real Arithmetic</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Nonlinear Real Arithmetic</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1. Summary of theoretical results for unbounded SMT theories.
Extending to Unbounded Theories

- Large widths: faster to solve, but less likely to be correct
- Small widths: slower to solve, but more likely to be correct
Extending to Unbounded Theories

• Bound inference via abstract interpretation (STAUB) [PLDI 2024]
• STAUB + SLOT speeds up nonlinear integers by 1.48x (z3) and 2.76x (CVC5)
• Up to 3.93x (z3) and 2.31x (CVC5) for verified linear integer benchmarks
• Substantial speedups for selected real constraints (up to 7x), but no substantial speedup on average
Conclusion and Future Work

• LLVM optimization passes can be applied outside the compiler context with substantial benefits

• Using our pass contribution data to inform future developments of solvers

• Adapting SMT solver simplification tactics to LLVM

• New MLIR dialect for SMT constraints

• Extension to arrays and strings
Conclusion and Future Work

• LLVM optimization passes can be applied outside the compiler context with substantial benefits
• Using our pass contribution data to inform future developments of solvers
• Adapting SMT solver simplification tactics to LLVM
• New MLIR dialect for SMT constraints
• Extension to arrays and strings

Any questions?