Leveraging LLVM Optimizations to Speed up Constraint Solving

Benjamin Mikek

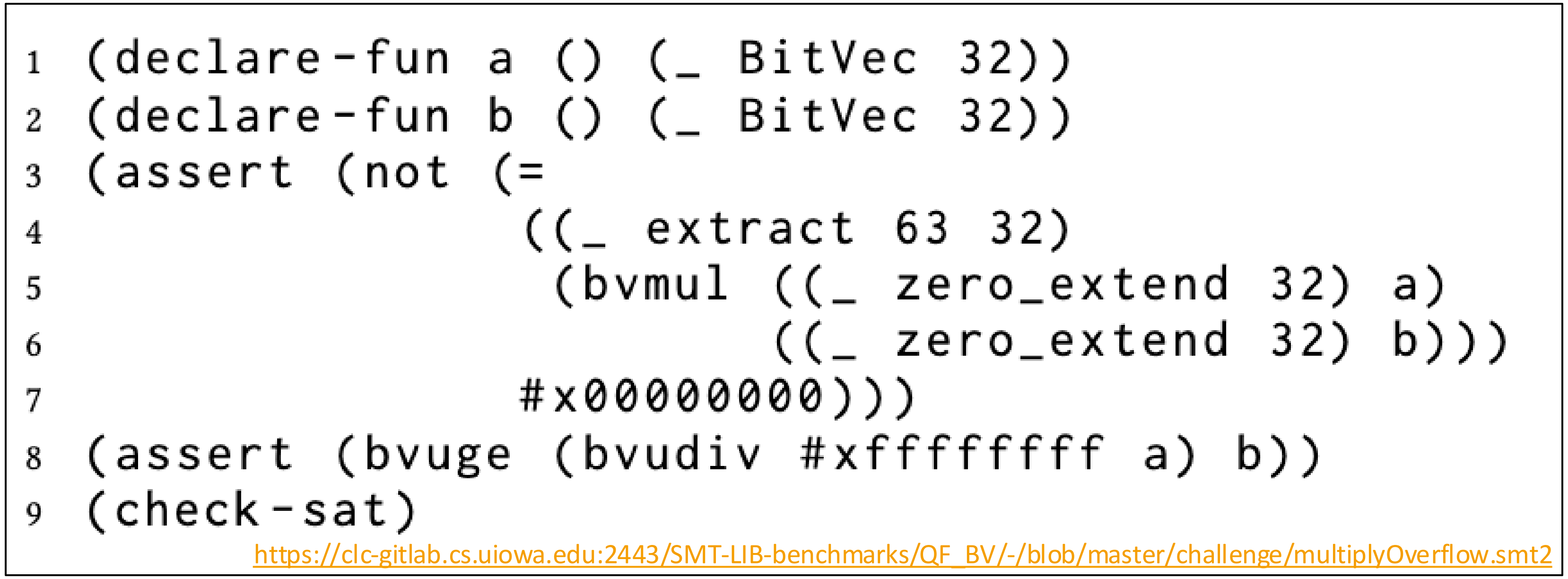
10 April 2024

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.

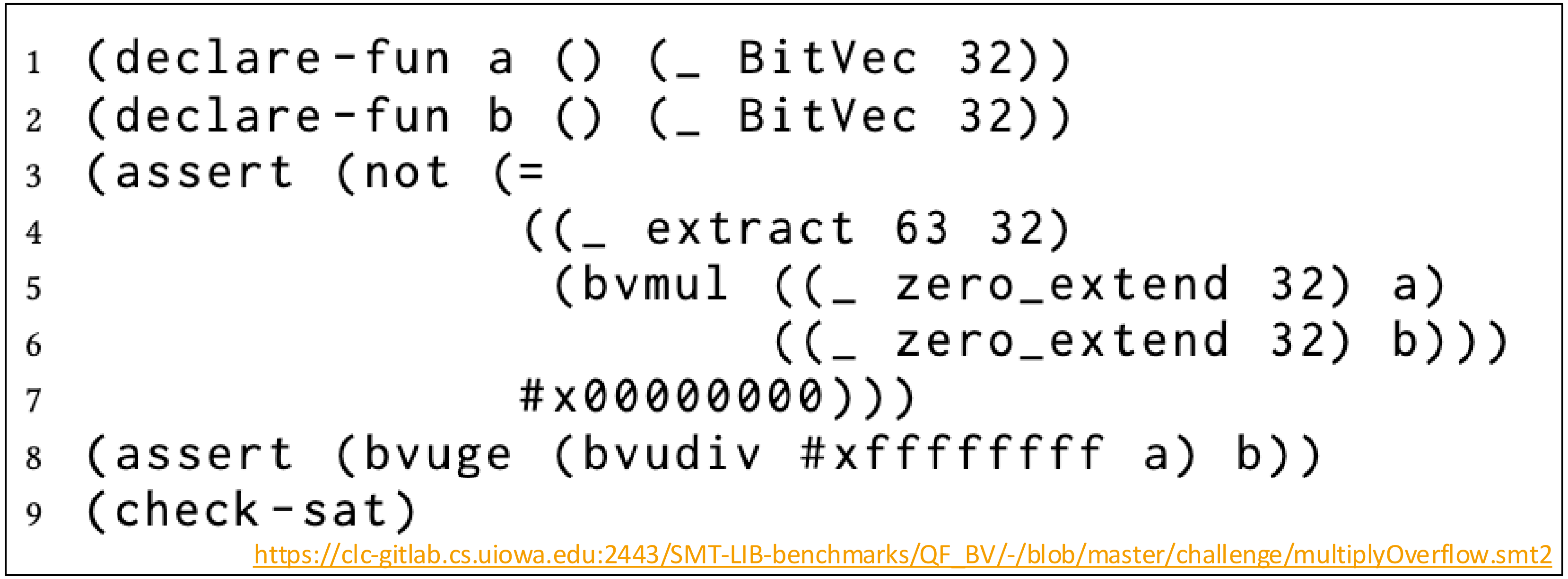
Example: Can multiplication overflow?



```
(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 (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

Example: Can multiplication overflow?

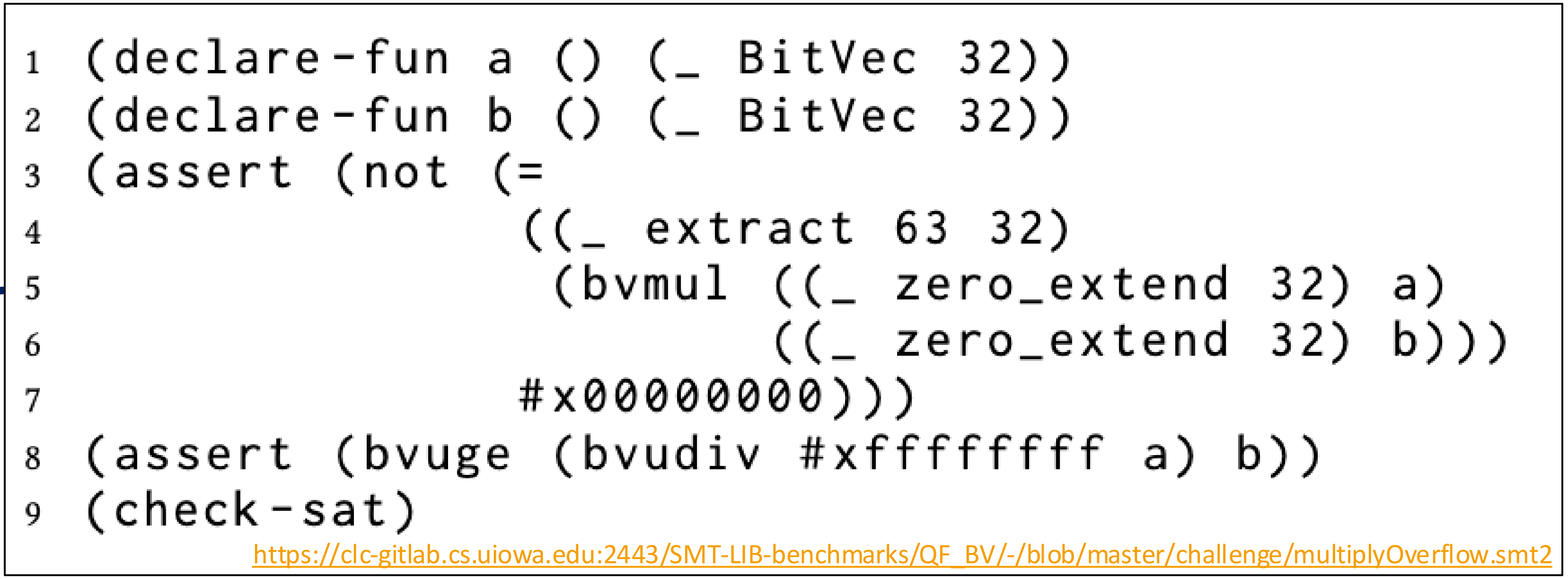


```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

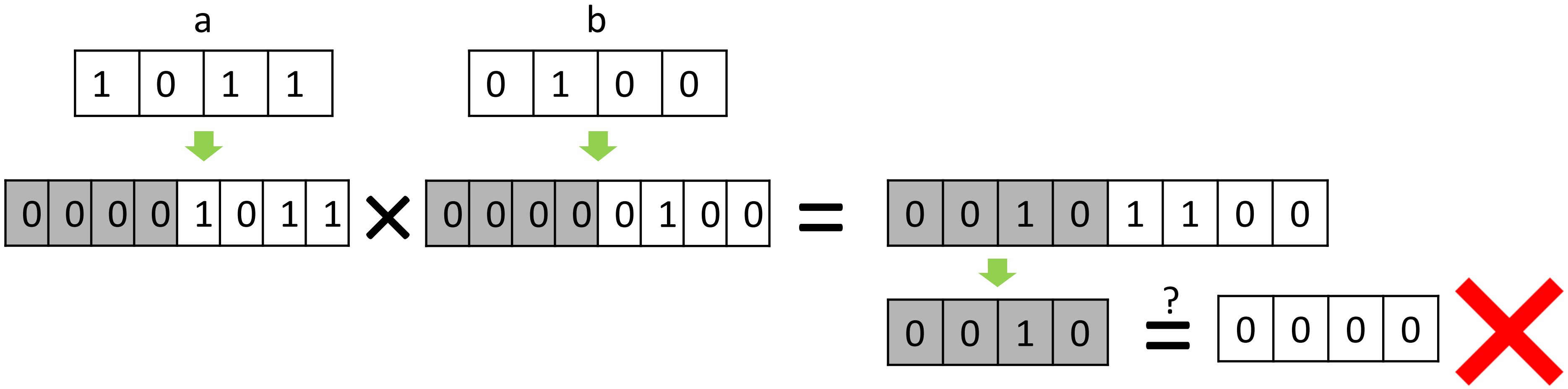
Is this constraint satisfiable?

Example: Can multiplication overflow?

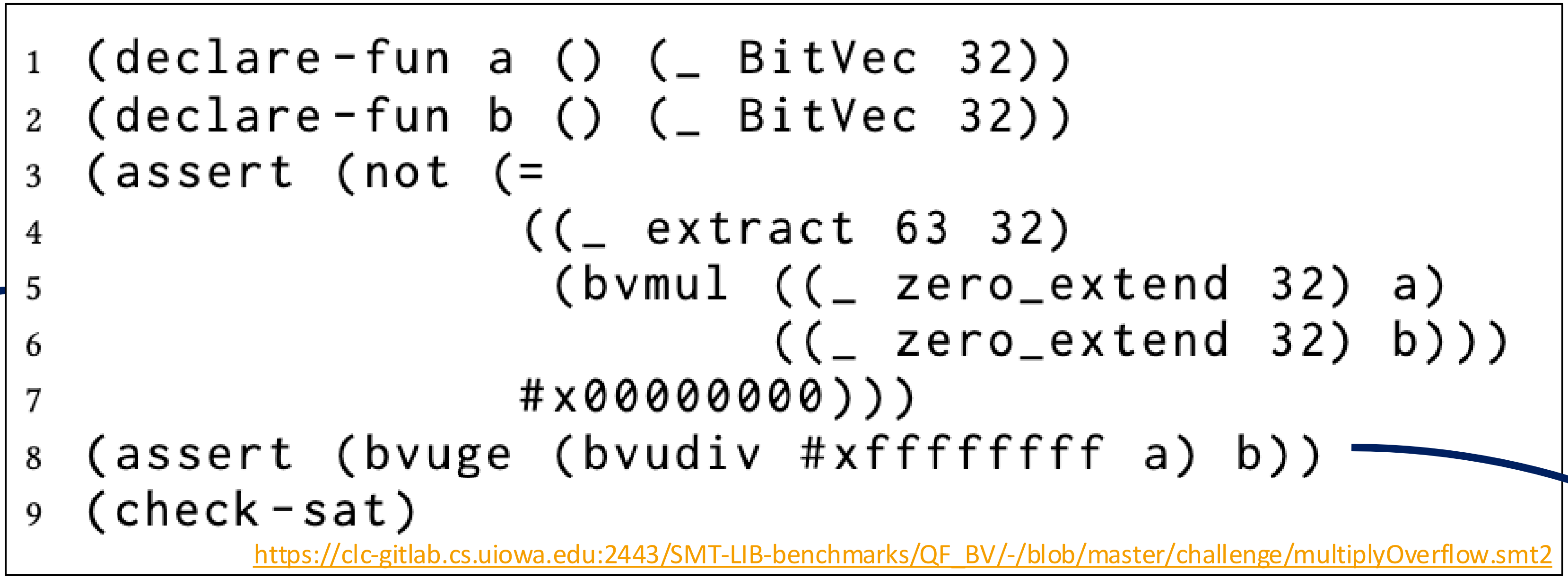


```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

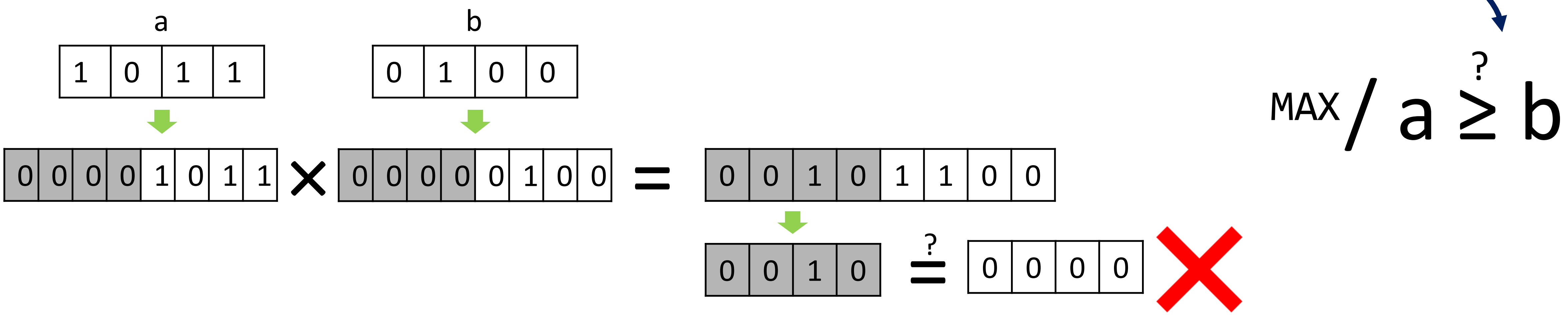


Example: Can multiplication overflow?

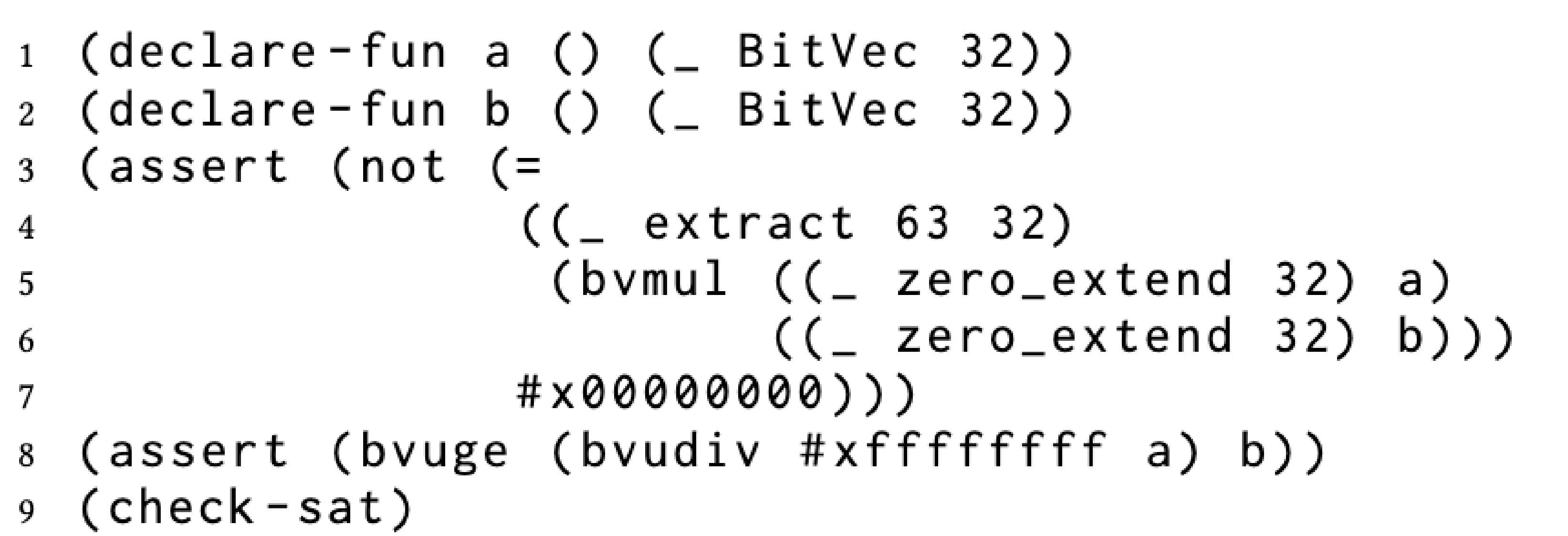


```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2



Example: Multiplication Overflow

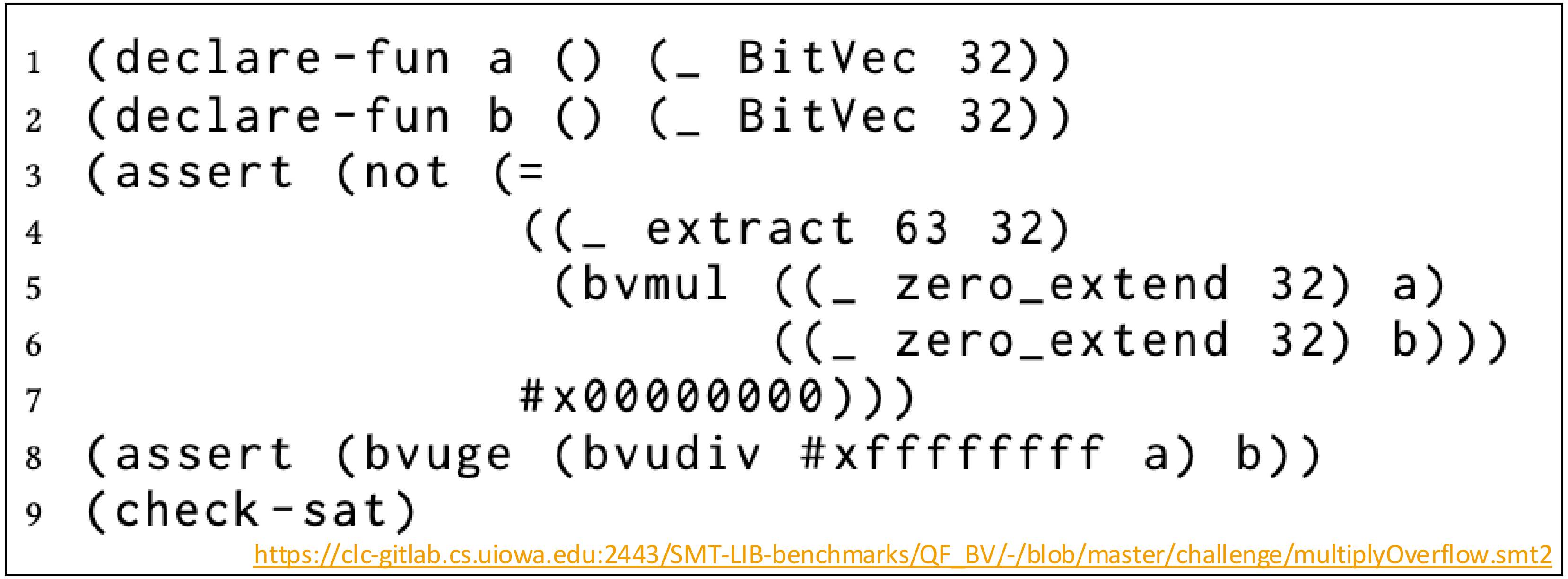


```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

Is this constraint satisfiable?

Example: Multiplication Overflow



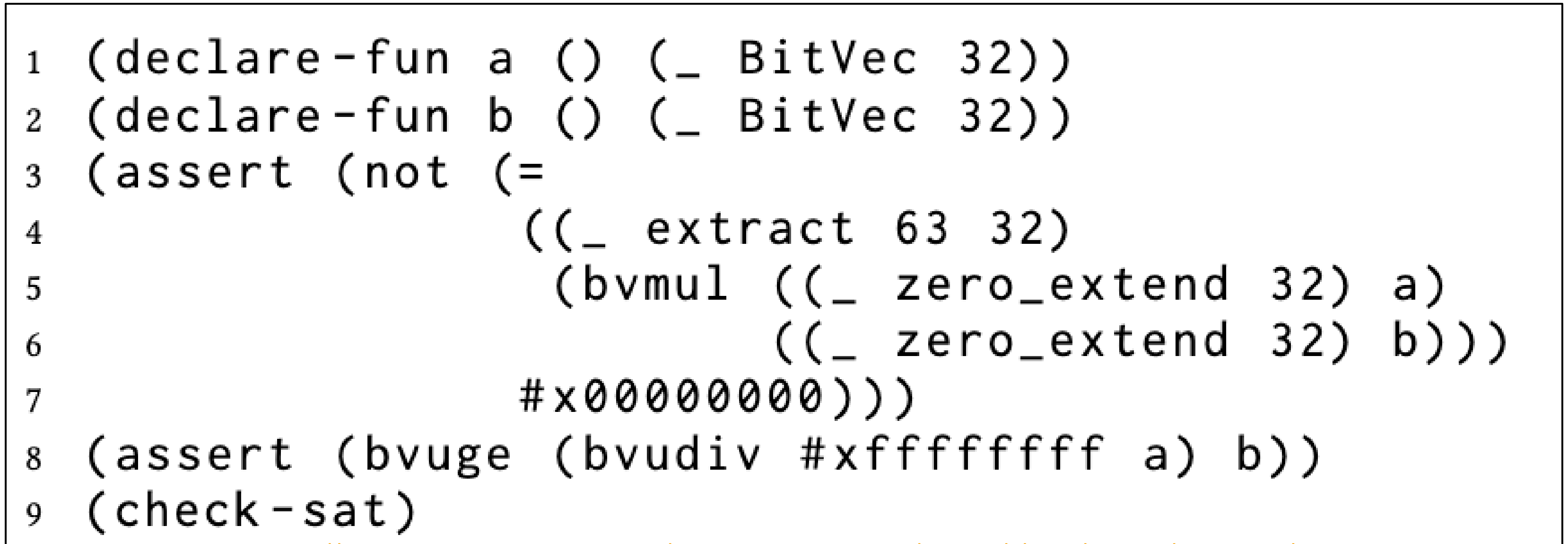
```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

Is this constraint satisfiable?

No!

Example: Multiplication Overflow



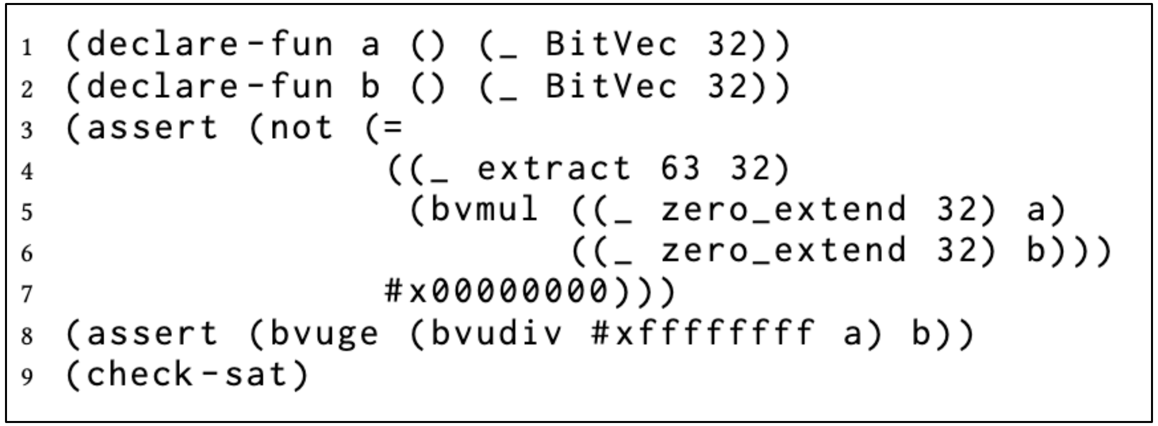
```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

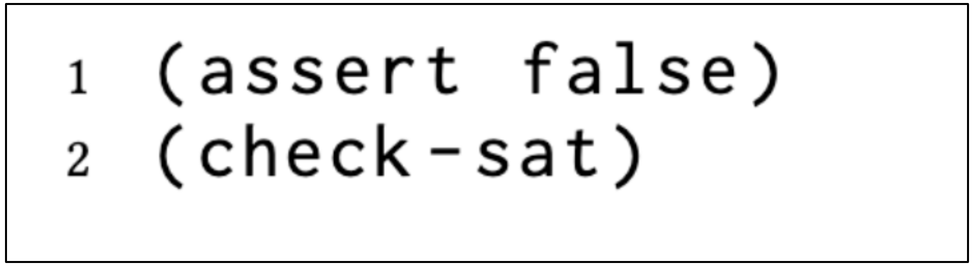
Is this constraint satisfiable?

But … z3 takes 10 minutes to solve it ☹

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 (bvudiv #xffffffff a) b))
(check-sat)
```



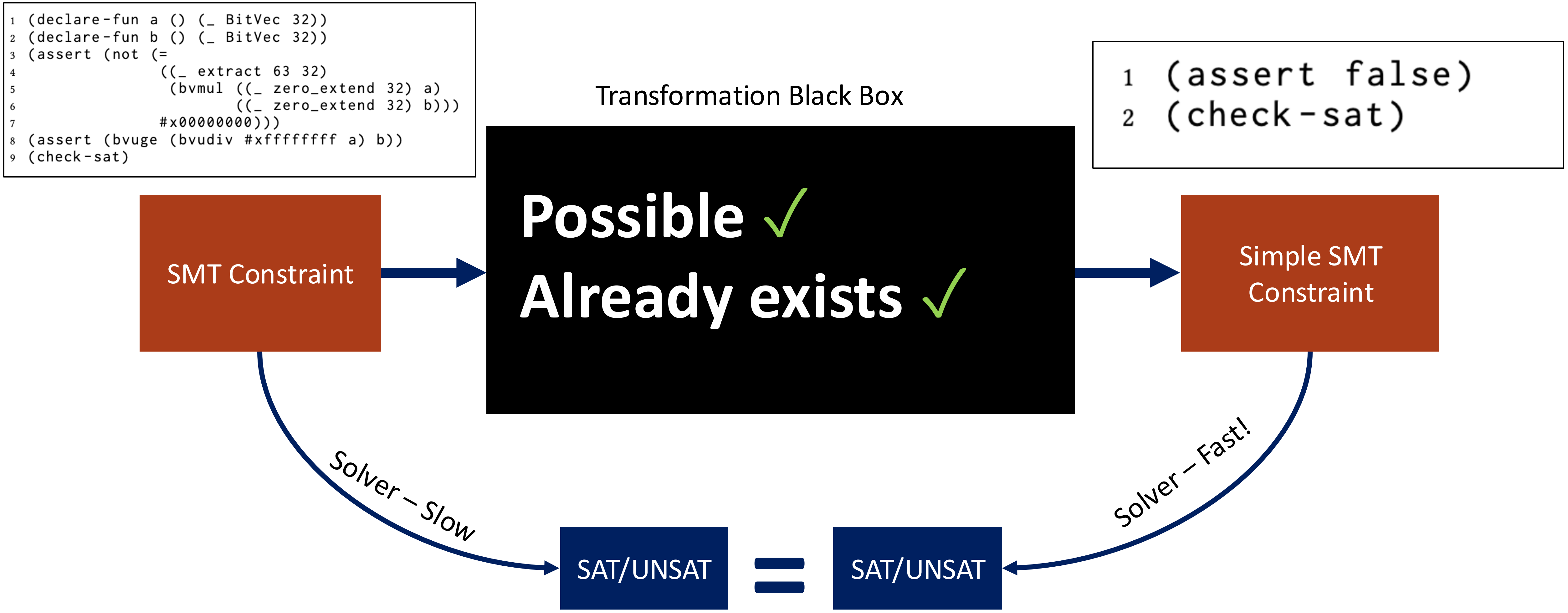
```
(assert false)
(check-sat)
```

```
> z3 complex.smt2
unsat
```

Takes 595 seconds ☹

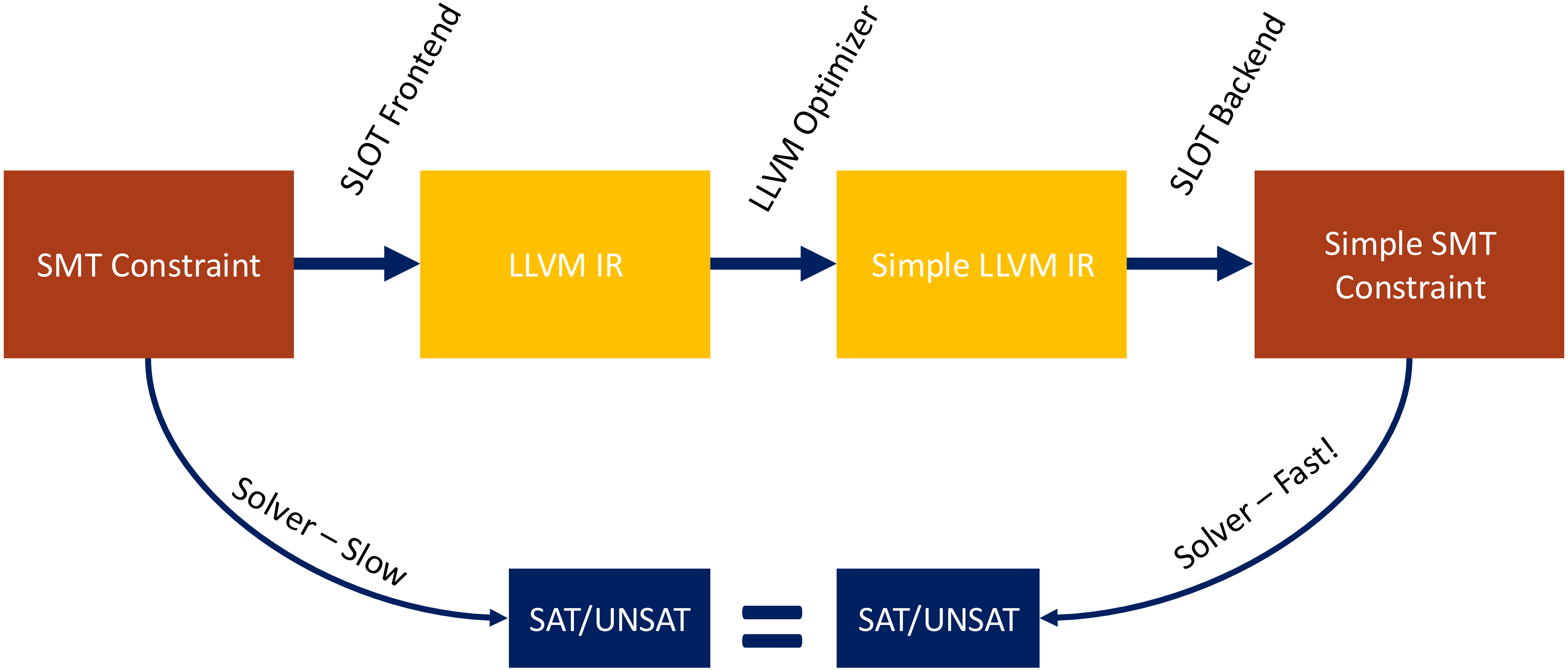
```
> z3 simple.smt2
unsat
```

Takes 0.02 seconds ☺



```
(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 (bvudiv #xffffffff a) b))
(check-sat)
```

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



SMT Constraint
SLOT Frontend
LLVM IR
LLVM Optimizer
Simple LLVM IR
SLOT Backend
Simple SMT Constraint
Solver – Slow
Solver – Fast!
SAT/UNSAT
SAT/UNSAT

SMTLIB and LLVM IR

	SMTLIB	LLVM
Bitvector types	One for each integer width n	One for each width up to 2^{23}
Floating point types	One for each integer pair eb, sb, but almost all in powers of 2	16-bit, 32-bit, 64-bit, 128-bit
Logic operations	and, or, xor, not, ite,=>, …	and, or, xor, select, …
Bitvector math operations	bvadd, bvsub, bvmul, bvsdiv, bvudiv, …	add, sub, mul, sdiv, udiv
Floating point math operations	fp.add, fp.sub, fp.div, fp.fma, …	fadd, fsub, fdiv, llvm.fma, …
Conversions	to_fp, to_fp_unsigned, fp.to_sbv, …	sitofp, uitofp, fptosi, …

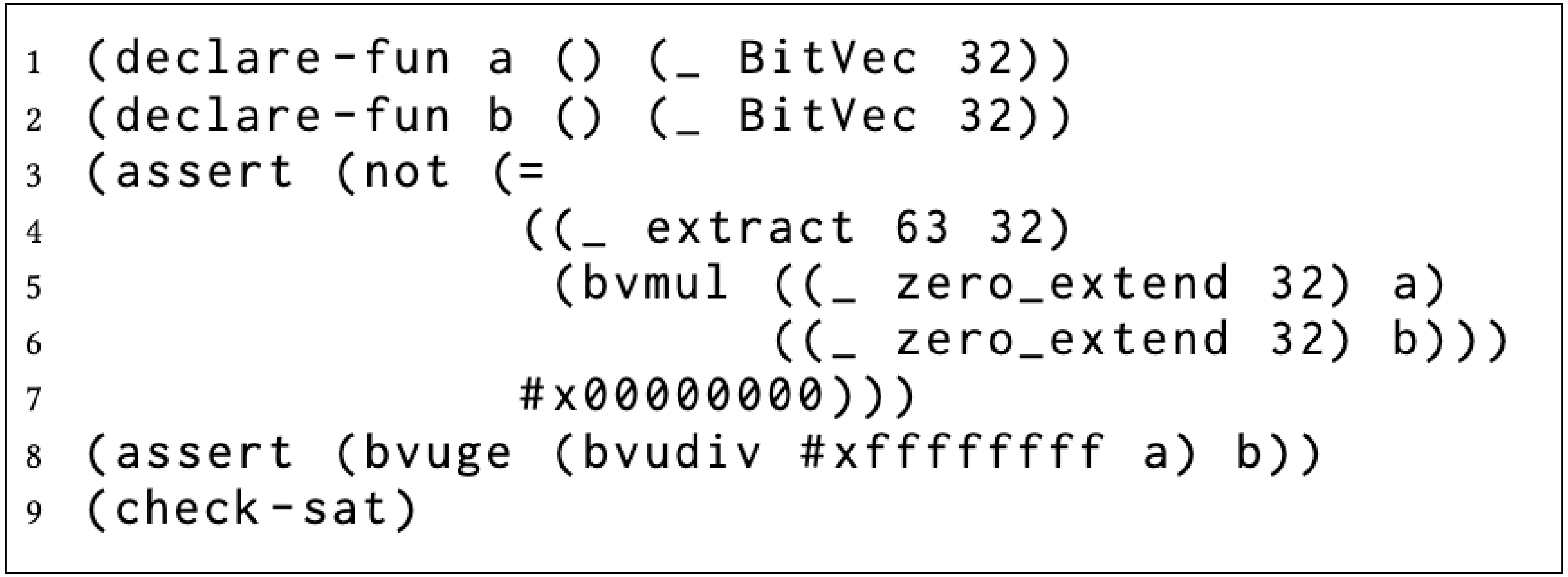
SLOT Translation

- Implemented as open source tool SLOT (SMT-LLVM Optimizing Translation) [FSE 2023]
- 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: Key Challenges

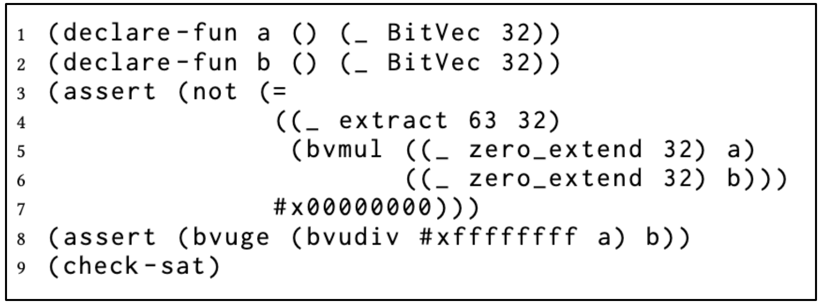
- SLOT has two parts: a front end and back end. Both have to preserve semantics
- LLVM is missing some SMT operations
- SMTLIB is missing some LLVM operations
- SMT constraints are *declarative*; LLVM is *imperative*

SLOT by Example: Checking for Overflow

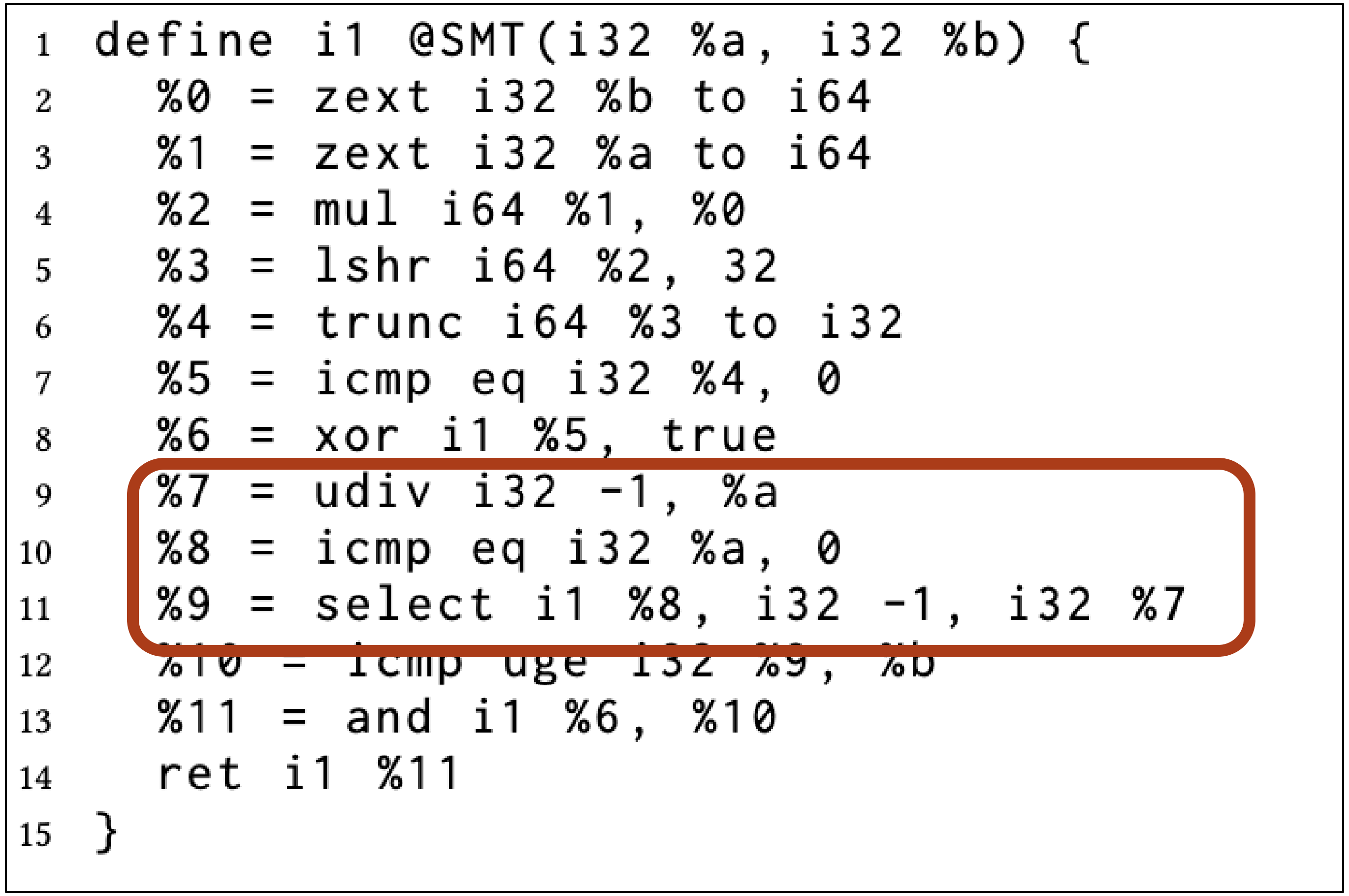


```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

Frontend Translation



```
(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 (bvudiv #xffffffff a) b))
(check-sat)
```

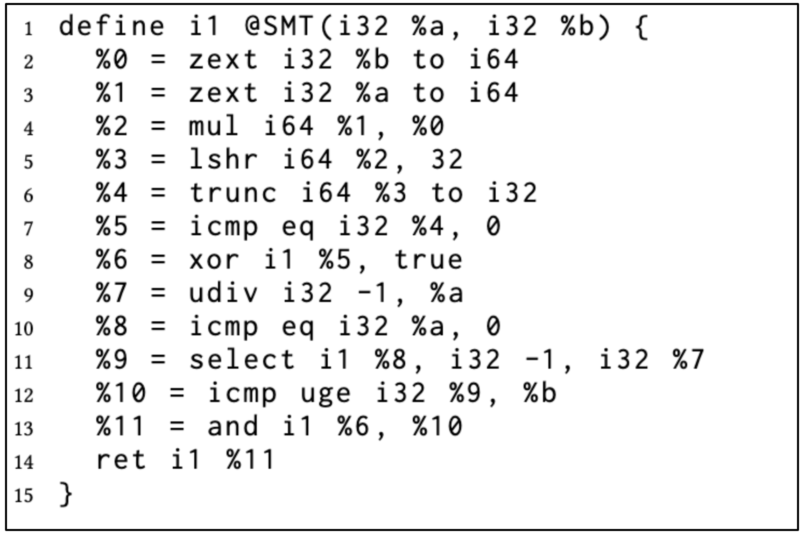


```
define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 -1, i32 %7
  %10 =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

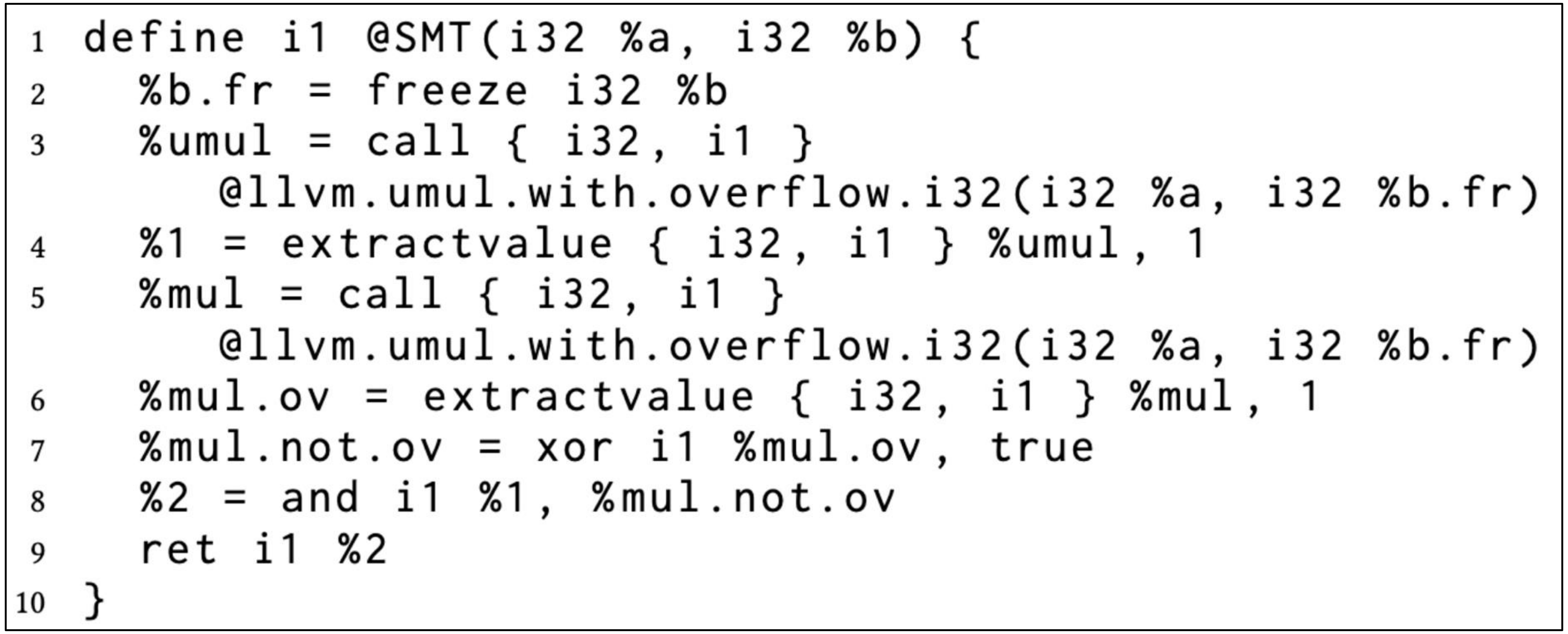
The LLVM function returns true if the inputs satisfy the underlying constraint.

Optimization

instcombine



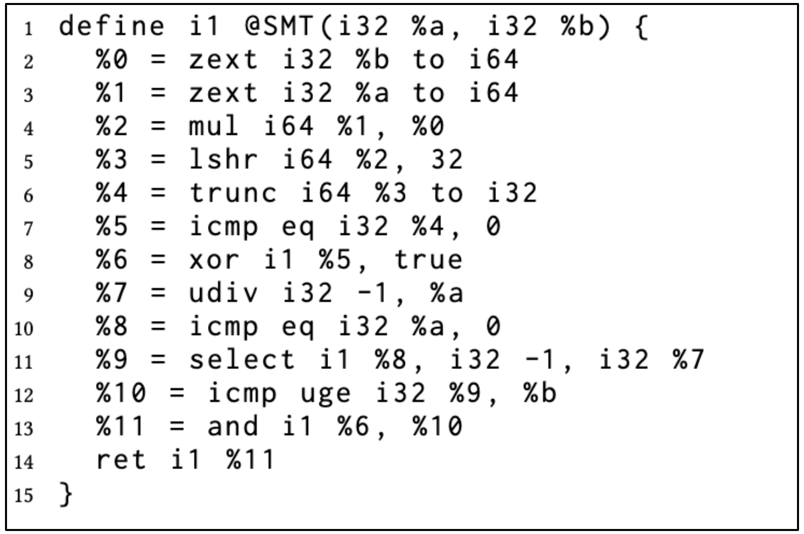
```
define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 -1, i32 %7
  %10 =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



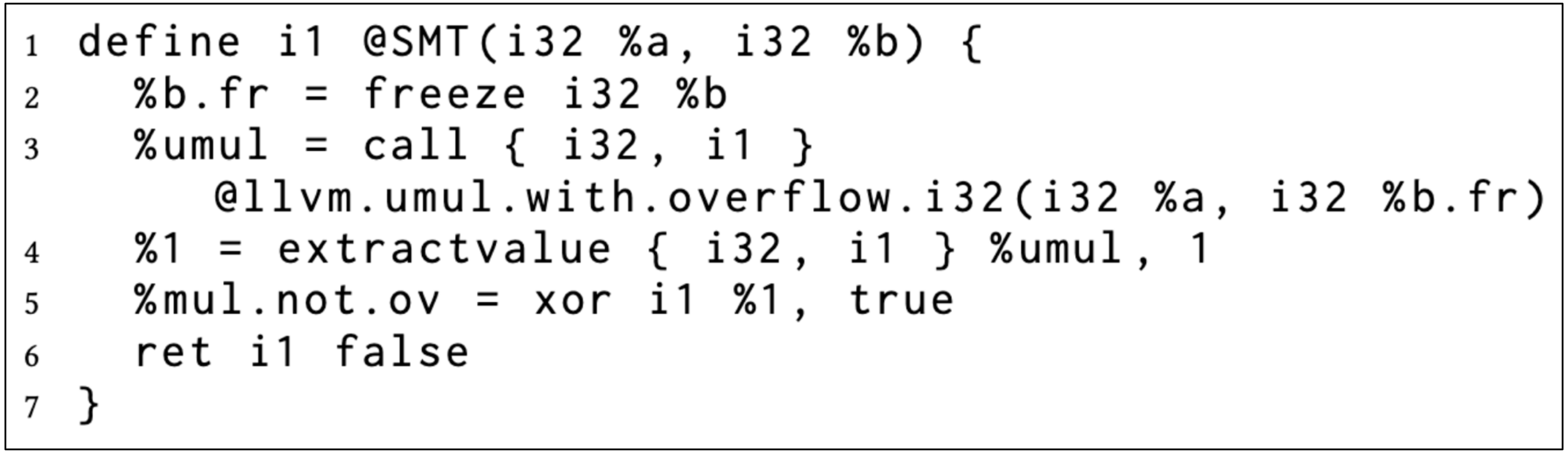
```
define i1 @SMT(i32 %a, i32 %b) {
  %b.fr = freeze i32 %b
  %umul = call { i32, i1 }
     @llvm.umul.with.overflow.i32(i32 %a, i32 %b.fr)
  %1 = extractvalue { i32, i1 } %umul, 1
  %mul = call { i32, i1 }
     @llvm.umul.with.overflow.i32(i32 %a, i32 %b.fr)
  %mul.ov = extractvalue { i32, i1 } %mul, 1
  %mul.not.ov = xor i1 %mul.ov, true
  %2 = and i1 %1, %mul.not.ov
  ret i1 %2
}
```

Optimization

instcombine, gvn



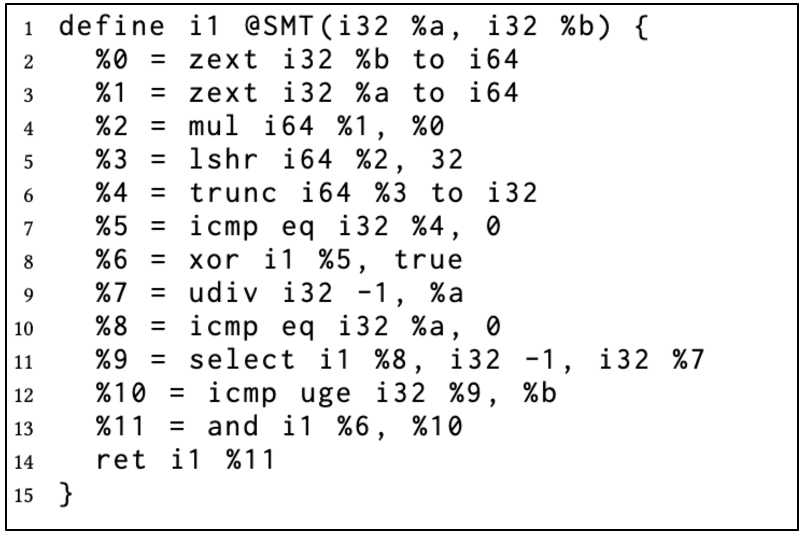
```
define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 -1, i32 %7
  %10 =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



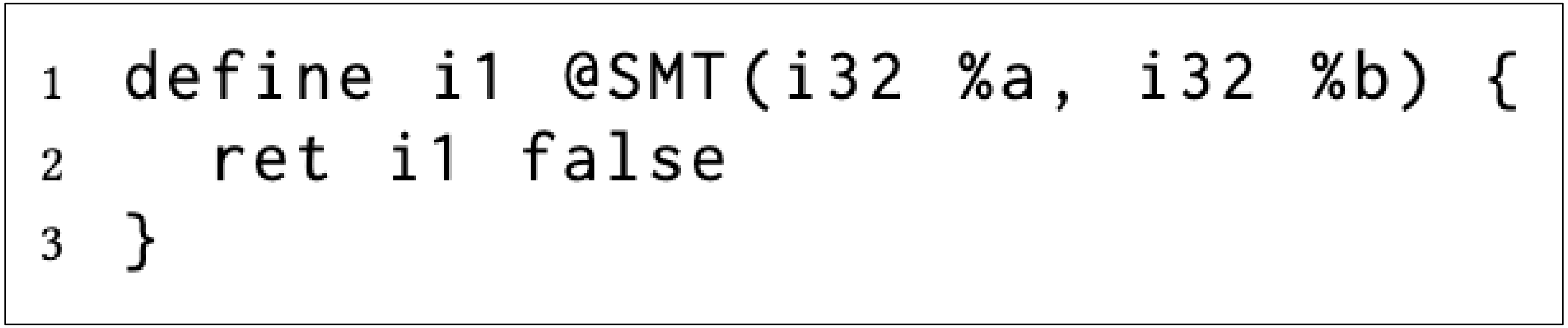
```
define i1 @SMT(i32 %a, i32 %b) {
  %b.fr = freeze i32 %b
  %umul = call { i32, i1 }
      @llvm.umul.with.overflow.i32(i32 %a, i32 %b.fr)
  %1 = extractvalue { i32, i1 } %umul, 1
  %mul.not.ov = xor i1 %1, true
  ret i1 false
}
```

Optimization

instcombine, gvn, instcombine

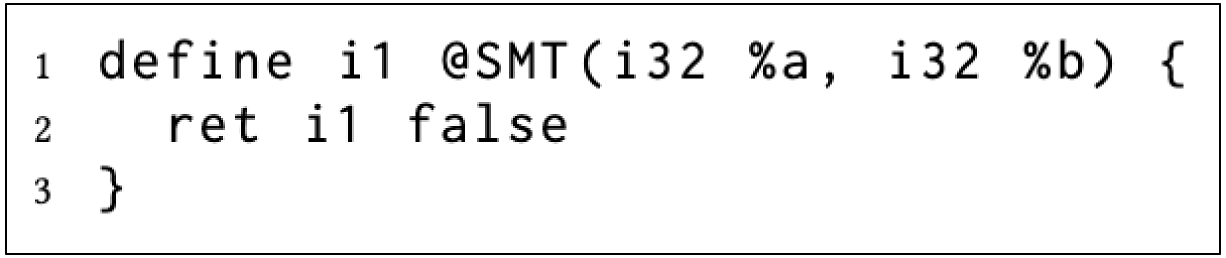


```
define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 -1, i32 %7
  %10 =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

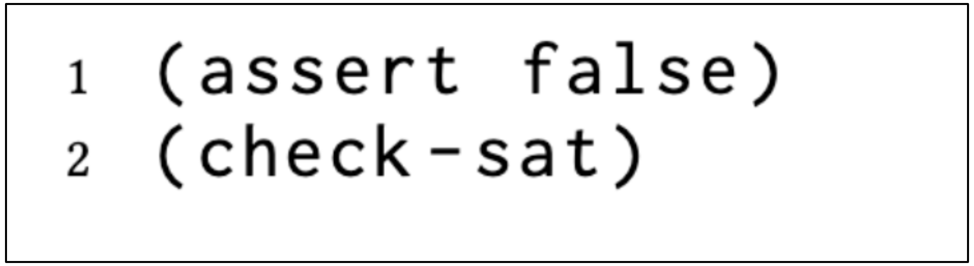


```
define i1 @SMT(i32 %a, i32 %b) {
  ret i1 false
}
```

Backend Translation

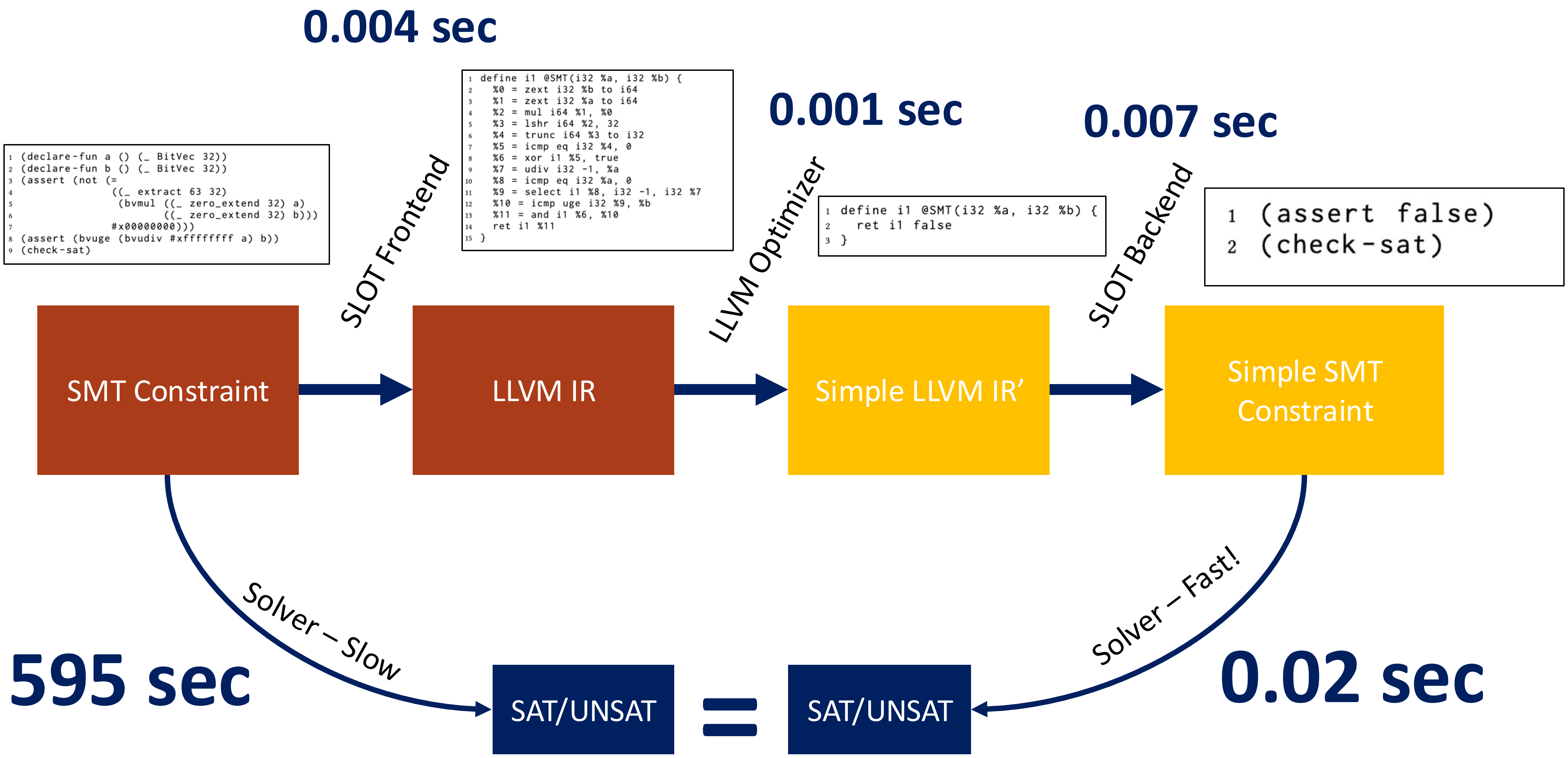


```
define i1 @SMT(i32 %a, i32 %b) {
  ret i1 false
}
```



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

Can be solved almost instantly (0.02 seconds)!



```
(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 (bvudiv #xffffffff a) b))
(check-sat)
```

```
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) {
  ret i1 false
}
```

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

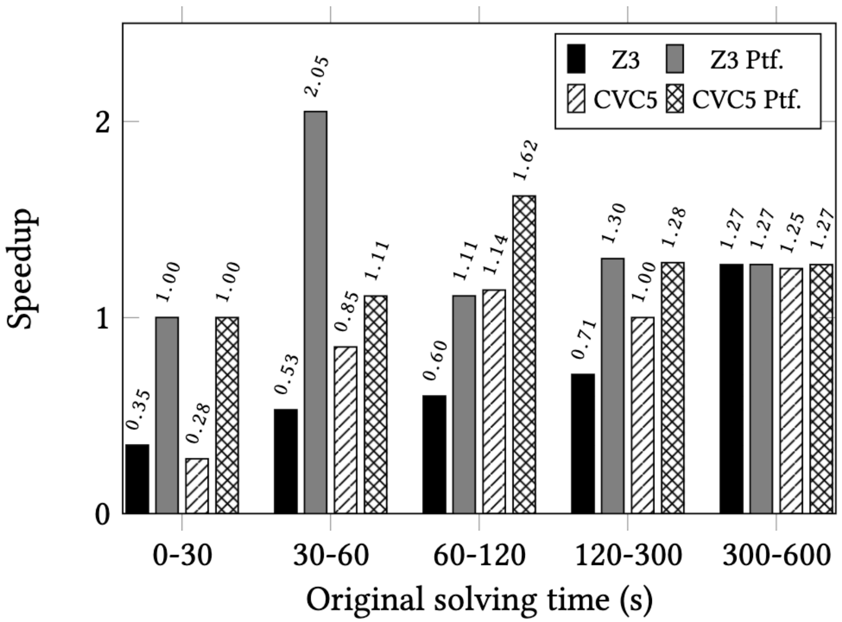
Results

- 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?

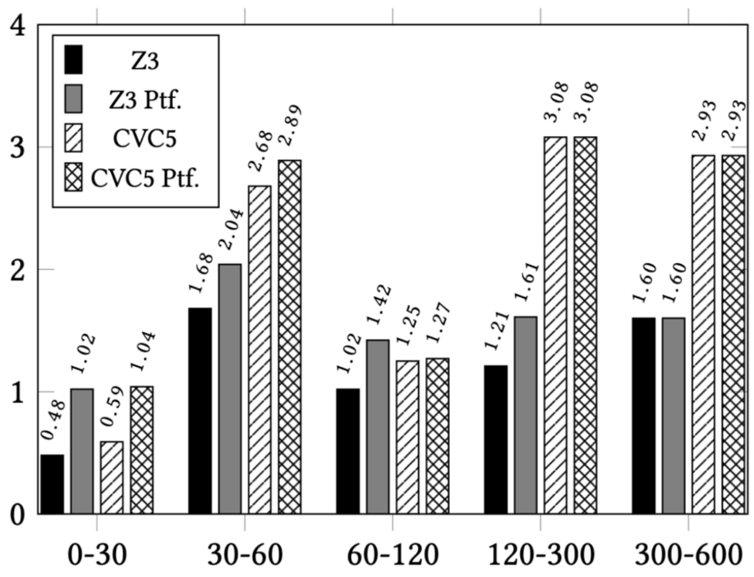
Results

- 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

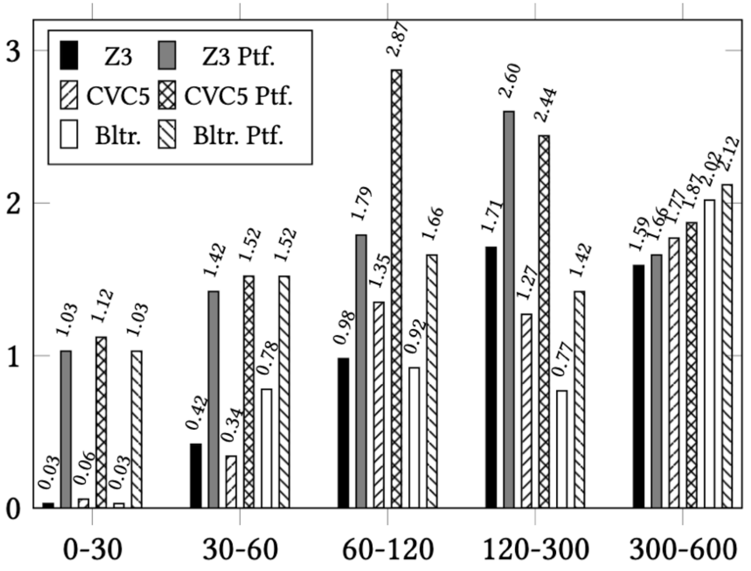
Results



Floating-point



Mixed



Bitvector

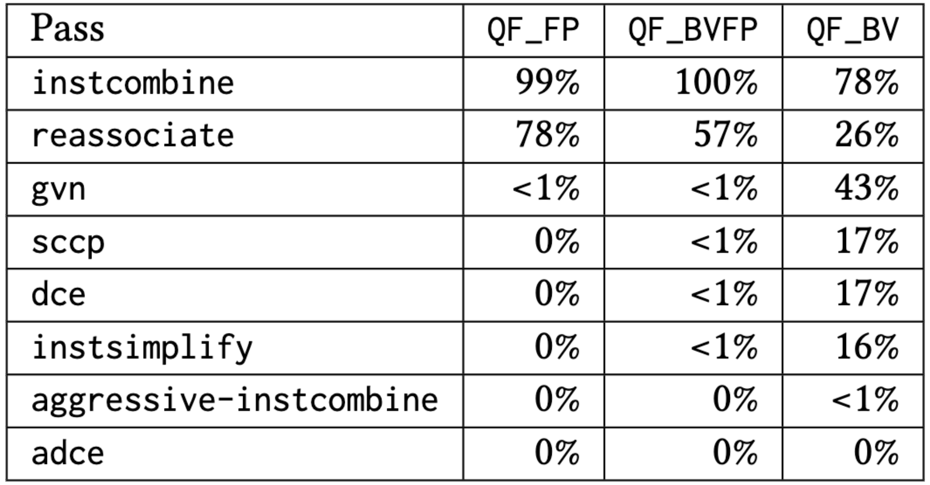
Optimization Pass Contributions

- 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

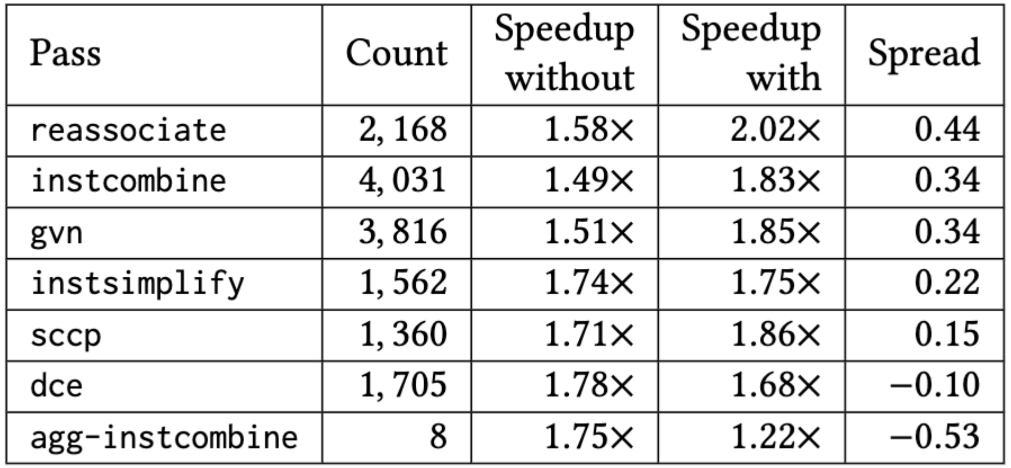
- The most effective optimizations passes are reassociate, instcombine, and global value numbering
- Solver developers can learn from these results about new optimizations to include in solvers

How many benchmarks does each pass change?



Pass	QF_FP	QF_BVFP	QF_BV
instcombine	99%	100%	78%
reassociate	78%	57%	26%
gvn	<1%	<1%	43%
sccp	0%	<1%	17%
dce	0%	<1%	17%
instsimplify	0%	<1%	16%
aggressive-instcombine	0%	0%	<1%
adce	0%	0%	0%

Which passes contribute the most speedup?



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

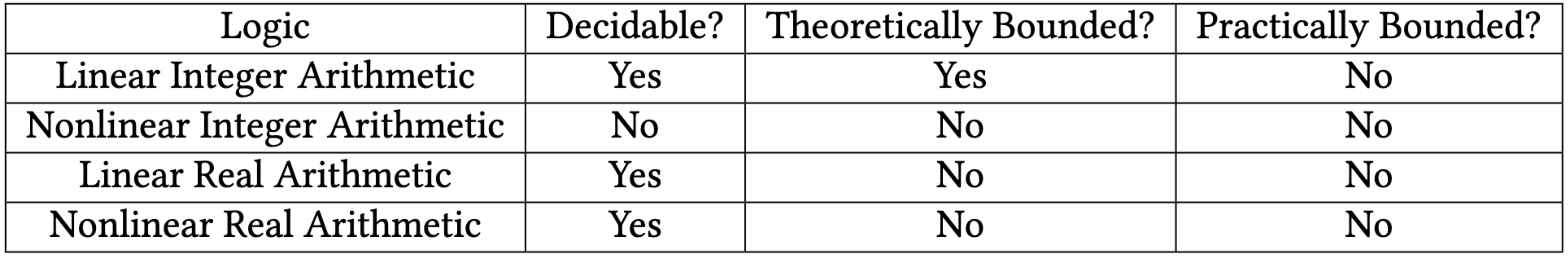
Discussion

- 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

Time Interval	Floating-point	Mixed	Bitvector
0-30	32.21%	22.17%	48.24%
30-60	0.02%	0.16%	3.96%
60-120	0.01%	0.20%	1.67%
120-300	0.01%	0.09%	2.01%
300-600	0.02%	0.01%	2.84%

Extending to Unbounded Theories

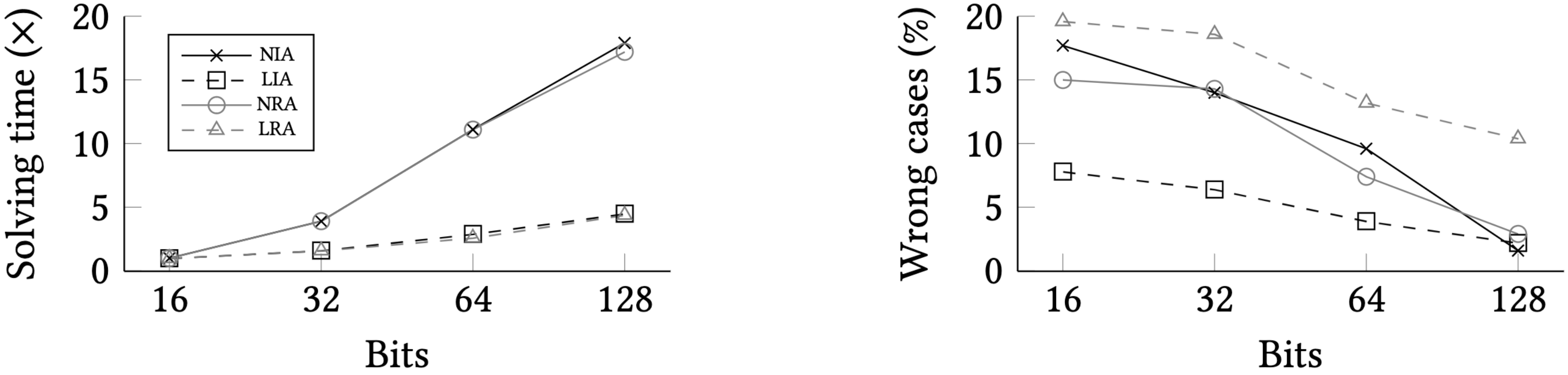
- SMTLIB also defines (mathematical) integers and real
- Integers can be converted to bitvectors
- Handling these theories requires *bound inference*
- Selected bounds will not always be enough, so we *underapproximate and verify*



Logic	Decidable?	Theoretically Bounded?	Practically Bounded?
Linear Integer Arithmetic	Yes	Yes	No
Nonlinear Integer Arithmetic	No	No	No
Linear Real Arithmetic	Yes	No	No
Nonlinear Real Arithmetic	Yes	No	No

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?