Speeding up SMT Solving via Compiler Optimization

Benjamin Mikek
bmikek@gatech.edu
Georgia Institute of Technology
USA

Qirun Zhang
qrzhang@gatech.edu
Georgia Institute of Technology
USA

Abstract

SMT solvers are fundamental tools for reasoning about constraints in practical problems like symbolic execution and program synthesis. Faster SMT solving can improve the performance and precision of those analysis tools. Existing approaches typically speed up SMT solving by developing new heuristics inside particular solvers, which requires nontrivial engineering efforts. This paper presents a new perspective on speeding up SMT solving. We propose SMT-LLVM Optimizing Translation (SLOT), a solver-agnostic pre-processing approach that utilizes existing compiler optimizations to simplify SMT problem instances. We implement SLOT for the two most application-critical SMT theories, bitvectors, and floating-point numbers. Our extensive evaluation based on the standard SMT-LIB benchmarks shows that SLOT can substantially increase the number of solvable SMT formulas given fixed timeouts and achieve mean speedups of nearly 3× for large benchmarks.

CCS Concepts

• **Software and its engineering → Formal software verification**.

Keywords

SMT Solvers, simplification, compiler optimization

ACM Reference Format:
Benjamin Mikek and Qirun Zhang. 2023. Speeding up SMT Solving via Compiler Optimization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616357

1 Introduction

Satisfiability Modulo Theories (SMT) constraints are first-order logical formulas with functions and variables from various theories, such as real numbers, integers, and, as relevant to software engineering, bitvectors, and floating-point numbers. State-of-the-art solvers like CVC5 [2] and Z3 [15] use a complex mix of heuristics, theory-specific engines, and SAT solver calls to efficiently reason about SMT constraints. Yet many constraints still take a prohibitively long time to solve. Improving solver performance can improve the results for real-world applications. For example, in symbolic execution, reducing solving time equates to greater code coverage [12].

The most popular approach to speeding up SMT solving is developing more powerful solving strategies. These have sometimes taken the form of new solvers like Boolector [35], or of new algorithms in existing solvers. For example, Berzish *et al.* [6] introduce new heuristics for string constraints involving regular expressions while Bjørner *et al.* [7] improve Z3's performance for custom theories. FastSMT [1] speeds up solving by using machine learning to choose the best solver heuristics.

This paper proposes a new perspective on improving SMT solving: instead of developing more advanced solving tactics, our key insight is to repurpose existing compiler optimization techniques to the SMT problem. In particular, we propose a translation-based pre-processing step, **S**MT-**L**LVM **O**ptimizing **T**ranslation (SLOT), which can directly optimize input SMT-LIB formulas. Conceptually, our approach has three main advantages:

- **Simplicity:** End-users of SMT solvers can benefit from compiler optimizations as a black box, without detailed knowledge of SMT-specific optimizations.
- **Solver-independence:** Because it is a pre-processing step on SMT constraints, semantics-preserving optimization can be used in applications that use any solver(s).
- **Extensibility:** New compiler optimizations can be directly applied to further improve SMT solving without the need to make complex changes to solvers.

SLOT bypasses the need to re-implement compiler optimizations in SMT solvers by translating the constraints, rather than the optimizations. While not all compiler optimizations are useful for the SMT context, the combination of semantics-preserving optimization with existing solvers creates a sieve: some constraints are caught quickly by existing solver heuristics, while others are handled better by SLOT.

We have implemented SLOT for the SMT theories of bitvectors and floating-point numbers. Constraints in these theories are the most relevant to software engineering because they model machine arithmetic; for example, they are used in practical tools for symbolic execution [12], translation validation [25], and program synthesis [8]. In Section 4, we show that the semantics of these two theories can be exactly represented in LLVM IR. The key challenge for SLOT is bridging the substantial semantic gap between SMT constraints and LLVM IR. Translation is particularly challenging because the languages, one declarative and the other imperative, were designed for entirely different purposes.

Figure 1 illustrates the three components of SLOT.

- *SLOT Frontend* translates SMT constraints to LLVM IR. This step ensures that every SMT function is converted to an equivalent sequence of LLVM instructions.
- *LLVM Optimization* utilizes the LLVM optimizer to optimize the translated constraint almost for free.

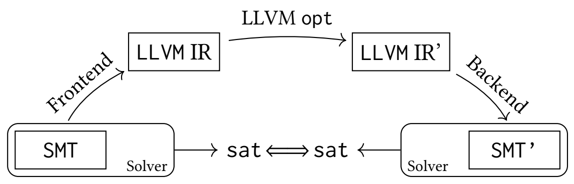


Figure 1: Overview of SLOT's translation and optimization process. The output constraint (SMT') is satisfiable if and only if the original constraint (SMT) is satisfiable.

- *SLOT Backend* translates the optimized LLVM IR back into an SMT constraint. The complex structures created by the optimizer must be translated back without semantic gaps.

We have applied SLOT to the quantifier-free benchmarks for bitvectors, floating-point numbers, and their combination included in the SMT-LIB specification [3]. Our extensive evaluation demonstrates that SLOT can substantially speed up SMT solving, especially for complex constraints which would otherwise take a long time to solve. Our approach increases the number of solvable constraints by up to 20% for bitvector, 15% for floating-point, and 80% for mixed benchmarks. Moreover, SLOT is more effective than existing solvers combined: it can solve constraints for which all tested solvers time out. We also observe mean speedups above 2× for bitvector and floating-point, and as high as 3× for mixed constraints. By measuring which optimization passes contribute most to the speedup, we find that simple peephole optimizations and global value numbering optimizations are sufficient to improve solver performance.

In summary, we make the following primary contributions:

- We present an easy-to-use, solver-agnostic framework for speeding up SMT solving by translating constraints to a compiler IR and back.
- We define, prove, and implement SLOT, and show that it improves the performance of solvers on standard benchmarks.
- We measure which LLVM optimization passes contribute most to speeding up SMT formulas, giving users access to well-tested simplifications and solver developers insight into possible solver improvements.

The rest of the paper is structured as follows. Section 2 motivates SLOT with an example SMT constraint. Section 3 presents background on constraints in SMT-LIB, while Section 4 describes SLOT's translation and proves its fidelity. Section 5 describes the evaluation results, and Section 6 puts the results in context. Finally, Section 7 surveys related work, and Section 8 concludes.

2 Motivating Example

This section presents a concrete example (Figure 2) to motivate SLOT. Specifically, it takes Z3 390 seconds to solve the original formula (Figure 2a). After applying SLOT, the optimized formula (Figure 2d) can be solved almost instantly.

Input SMT constraint. Figure 2a gives an SMT formula from the SMT-LIB `QF_BV` benchmark set.[1] It checks whether multiplication can overflow (lines 3-7) when the inputs a and b are subject to a division constraint (line 8). The formula is `unsat` because the values

[1]QF_BV/challenge/multiplyOverflow.smt2

```
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)
```

(a) Original SMT-LIB constraint.

```
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
}
```

(b) Result of SLOT frontend translation.

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

(c) Result of SLOT optimization.

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

(d) Final formula after SLOT backend translation.

Figure 2: SLOT translation and optimization process.

of a that satisfy the second assertion would cause the multiplication $a \times b$ to overflow. Even though this constraint is concise and simple, Z3 takes 390 seconds to return the `unsat` result.

SLOT frontend. Figure 2b gives the result of SLOT's frontend: an LLVM function that is semantically equivalent to the SMT constraint in Figure 2a. This function returns true on an input (a, b) if and only if (a, b) satisfies the original SMT constraint. Its instructions mirror the function applications in Figure 2a. For example, `zext` is equivalent to the SMT `zero_extend` operation and `mul` is equivalent to the SMT `bvmul` function.

SLOT optimization. Figure 2c gives the result of LLVM optimization on the function in Figure 2b using all available optimization passes (`O3`). For this function, two LLVM optimization passes (`instcombine` and `gvn`) suffice to simplify away all substantive instructions, producing the function that always returns false.

SLOT backend. Finally, Figure 2d shows the result of translating Figure 2c back to an SMT constraint. It is obvious that the LLVM function always returns false, so the corresponding SMT constraint simply asserts falsity. Z3 can now trivially conclude that the constraint in Figure 2d is unsatisfiable in 0.02 seconds.

Challenges. From Figure 2, we can see that SLOT allows an SMT solver to leverage the power of existing LLVM optimizations. The key technical challenge of SLOT is bridging the semantic gap between SMT constraints and LLVM IR, *i.e.*, ensuring the input and output SMT constraints are equivalent. While multiplication and bit extension are equivalent in the two languages, SMT functions cannot always be directly mapped to LLVM IR. For example, line 10 of Figure 2b adds a check to ensure the division on line 8 of Figure 2a does not introduce division by zero, because this has undefined behavior in LLVM.

3 Preliminaries

This section gives background on the SMT problem [4] and describes the bitvector and floating-point theories of the SMT-LIB standard [3].

Definition 1 (SMT formula). *Given a* theory T *with signature Σ and interpretations I, an SMT formula ϕ is an expression made up of symbols (function applications or variables) from Σ. I is the set of maps from variables in Σ to sort-appropriate values. ϕ is satisfiable if there exists an interpretation in I that satisfies ϕ.*

Intuitively, a theory T provides definitions of *sorts* (*i.e.*, types) and functions, and an SMT formula is a set of variables and a constraint on those variables using functions from T. If there exists an assignment of the variables which fulfills the constraint, we call the formula sat; otherwise, it is unsat. The SMT-LIB standard defines eight theories and from these 29 *logics*, combinations of functions from one or more theories, possibly with extensions. All logics rely on the Core theory, which defines basic boolean operations like logic and, logic or, and equality. We restrict our discussion to the Core theory and the quantifier-free logics of bitvectors and floating-point numbers.

SMT-LIB has a sort for each width of bitvector ((_ BitVec n)), several unary and binary operations on bitvectors like bvneg and bvadd, and bitvector comparisons like bvuge. The theory of floating-point numbers defines the sorts (_ FloatingPoint e s) for integers $e, s > 1$. Operations on floating-point values follow standard IEEE-754 semantics [34], though the sizes of the exponent and significand are not limited to those defined in IEEE-754. As with the bitvector logic, there are unary and binary operations on floating-point values (many of these require the specification of one of the five rounding modes) and comparisons that yield booleans. There are also conversions from floating point to bitvectors, from bitvectors to floating-point values, and between different size floating-point values. Table 1 gives a full list of QF_BV and QF_BVFP functions.

Following the work of Kroening and Strichman [21], we summarize Table 1 using the grammar shown in Figure 3. The grammar consists of formulas (F), bitvector comparisons (B), floating-point comparisons (C), bitvector values (V), and floating-point values (W). Intuitively, formulas are expressions with boolean sort; bitvector and floating-point comparisons are expressions of boolean sort which take bitvector or floating-point expressions, respectively; and values are expressions of bitvector or floating-point sort.

4 SLOT: SMT-LLVM Optimizing Translation

This section formalizes the translation described in Section 2, presents proofs of semantics preservation for bitvectors and floating point values, and describes the under-approximation necessary for unbounded integers.

Table 1: List of functions in the bitvector and floating-point theories by type. We abbreviate bitvectors BV, floating-point values FP, and rounding modes RM. A represents any type. "*" indicates a function parameterized by integer constants, "†" indicates a function that changes bit widths.

Function sort	QF_BV **and** QF_BVFP **functions**
Bool $\longrightarrow$ Bool	not
Bool $\times$ Bool $\longrightarrow$ Bool	$\Rightarrow$, and, or, xor
$A \times A \longrightarrow$ Bool	=, distinct
Bool $\times A \times A \longrightarrow A$	ite
BV $\times$ BV $\longrightarrow$ Bool	bvule, bvsle, bvuge, bvsge, bvult, bvslt, bvugt, bvsgt
BV $\longrightarrow$ BV	bvnot, bvneg, extract*†, repeat*†, zero_extend*†, sign_extend*†, rotate_left*, rotate_right*
BV $\times$ BV $\longrightarrow$ BV	concat†, bvadd, bvsub, bvmul, bvsdiv, bvudiv, bvsrem, bvurem, bvsmod, bvand, bvor, bvnot, bvxor, bvnand, bvnor, bvxnor, bvshl, bvlshr, bvashr, bvcomp
FP $\longrightarrow$ Bool	fp.isNaN, fp.isInfinite, fp.isZero, fp.isNormal, fp.isSubnormal, fp.isNegative, fp.isPositive
FP $\times$ FP $\longrightarrow$ Bool	fp.eq, fp.lt, fp.gt, fp.leq, fp.geq
FP $\longrightarrow$ FP	fp.neg, fp.abs
RM $\times$ FP $\longrightarrow$ FP	fp.sqrt, to_fp*†, fp.roundToIntegral
FP $\times$ FP $\longrightarrow$ FP	fp.rem, fp.min, fp.max
RM $\times$ FP $\times$ FP $\longrightarrow$ FP	fp.add, fp.sub, fp.mul, fp.div
RM $\times$ FP $\times$ FP $\times$ FP $\longrightarrow$ FP	fp.fma
BV $\longrightarrow$ FP	to_fp*
BV $\times$ BV $\times$ BV $\longrightarrow$ FP	fp*
RM $\times$ BV $\longrightarrow$ FP	to_fp*†, to_fp_unsigned*†
RM $\times$ FP $\longrightarrow$ BV	fp.to_ubv*†, fp.to_sbv*†

$$
\begin{aligned}
F &:= \texttt{true} \mid \texttt{false} \mid C \mid (\texttt{not}\ F) \mid (\Rightarrow\ F\ F) \mid \\
&\quad (\texttt{and}\ F\ F) \mid (\texttt{or}\ F\ F) \mid (\texttt{xor}\ F\ F) \mid (=\ F\ F) \mid \\
&\quad (\texttt{distinct}\ F\ F) \mid (\texttt{ite}\ F\ F\ F) \\
B &:= (=\ V\ V) \mid (\texttt{distinct}\ V\ V) \mid (\texttt{bvc}\ V\ V) \\
C &:= (=\ W\ W) \mid (\texttt{distinct}\ W\ W) \mid (\texttt{fpc}\ V\ V) \mid \\
&\quad (\texttt{fp.isclass}\ W) \\
V &:= \text{Constant} \mid \text{Symbol} \mid (\texttt{ite}\ F\ V\ V) \mid (\texttt{bvop1}\ V) \mid \\
&\quad (\texttt{bvop2}\ V\ V) \mid (\texttt{fp.to\_ubv}\ W) \mid (\texttt{fp.to\_sbv}\ W) \\
W &:= \text{Constant} \mid \text{Symbol} \mid (\texttt{fp.fma}\ W\ W\ W) \mid \\
&\quad (\texttt{fpop1}\ W) \mid (\texttt{fpop2}\ W\ W) \mid (\texttt{to\_fp}\ V) \mid \\
&\quad (\texttt{to\_fp\_unsigned}\ V) \mid (\texttt{fp}\ V\ V\ V)
\end{aligned}
$$

Figure 3: The grammar of constraints in the QF_BV and QF_BVFP logics. bvc **is any of the bitvector comparisons from Table 1.** fpc **means any of the floating-point comparisons, and** class **means any of the floating-point class operations.** bvop1 **and** bvop2 **mean any of the unary and binary bitvector operations, respectively, and the same for** fpop1 **and** fpop2**.**

4.1 Overview

Given an SMT constraint C, SLOT translates each operation to an LLVM equivalent, creating an LLVM function L. It then invokes the LLVM optimizer, producing optimized function L'. Finally, it translates back into an SMT constraint C'. Intuitively, equivalence between C and C' means that their sets of satisfying assignments are equal. Equivalence between a constraint and an LLVM function means that, given a variable assignment, evaluating the constraint produces the same result as executing the LLVM function.

Algorithm 1 performs the translation from an SMT-LIB constraint C into an LLVM function L. Each function corresponds to an SMT sort, and builds the LLVM statements corresponding to

Algorithm 1: SLOT frontend translation.

```
Data: C, a constraint with function $C_{op}$, children $C_0, C_1, \ldots$
Result: L, an LLVM function
Function BuildLLVM($C$):
    if $C$ is an FP comparison then
        return BuildComparison($FP, C$);
    else if $C$ is a BV comparison then
        return BuildComparison($BV, C$);
    else if $C$ is a leaf then
        return C as boolean constant;
    else return GetLLOp($C_{op}$, BuildLLVM($C_0$), BuildLLVM($C_1$));
Function BuildComparison($T, C$):
    return GetLLOp($C_{op}$, BuildVal($T, C_0$), BuildVal($T, C_1$));
Function BuildVal($T, C$):
    if $C$ is a leaf then
        return C as a T constant;
    else if $C$ is an FP conversion then
        return GetLLOp($C_{op}$,BuildVal($BV, C_0$))
    else if $C$ is a BV conversion then
        return GetLLOp($C_{op}$,BuildVal($FP, C_0$))
    else if $C$ is ite then
        return Select($C_0$, BuildVal($T, C_1$), BuildVal($T, C_2$));
    else return GetLLOp($C_{op}$, BuildVal($T, C_0$), BuildVal($T,
      C_1$));
```

an SMT expression with that sort. The GetLLOp function represents fetching an LLVM instruction or instructions which have the same effect as the input SMT operation C_{op}.

Optimization from L to L' is performed by the LLVM optimizer. For black box style processing of SMT constraints, SLOT uses all the passes included in LLVM's O3 optimization level. However, not all optimization passes are relevant to SLOT's translation (for instance, SLOT does not introduce any memory operations). Section 5 discusses in detail which LLVM passes are most important for SLOT.

Finally, we translate L' back into an SMT constraint C' with Algorithm 2. This translation is straightforward; we proceed up the L' syntax tree and convert each instruction it to its equivalent SMT-LIB function as in Algorithm 2. Because frontend translation, optimization, and backend translation all preserve the semantics of the constraint, we can then use the satisfiability of C' as a proxy for the satisfiability of C—this property is formalized in Theorem 1.

The key challenge of translation is defining GetLLOp without introducing undefined behavior. Some functions can be translated one-to-one, but bitvector division, shifts, and floating-point comparisons have subtly different semantics in LLVM and SMT-LIB. In addition, some SMT operations have no direct LLVM equivalent and vice versa, requiring their semantics to be built from existing operations in each language.

4.2 Frontend translation

Types. Let C be an SMT constraint over variables $c_1, c_2, \ldots, c_n$. The possible sorts of a variable c_i are boolean, bitvector, and floating-point. An SMT-LIB *boolean* is equivalent to the LLVM i1 type. An n-wide *bitvector* is equivalent to the LLVM type in. The SMT-LIB floating-point sorts (_ FloatingPoint 5 11), (_ FloatingPoint 8 24), (_ FloatingPoint 11 53), and (_ FloatingPoint 15 113) are respectively equivalent to LLVM half, float, double, and fp128.

Algorithm 2: SLOT backend translation.

```
Data: L, an LLVM function
Result: C, an SMT constraint
Function BuildSMT($L$):
    $V \leftarrow L.arguments$;
    return ConvertVal($V, L.return$);
Function ConvertVal($Vars, Value$):
    if $Value$ in $Vars$ then
        return GetSMTVar($Value$);
    else if $Value$ is a constant then
        return GetSMTConst($Value$);
    else if $Value$ is icmp then
        return GetSMTBvComp(ConvertVal($Vars, Value.op(0)$),
          ConvertVal($Vars, Value.op(1)$));
    else if $Value$ is fcmp then
        return GetSMTFloatComp(ConvertVal($Vars, Value.op(0)$),
          ConvertVal($Vars, Value.op(1)$));
    else if $Value$ is an intrinsic call then
        return GetSMTIntrinsic(ConvertVal($Vars, Value.op(0)$),
          ...);
    else
        return GetSMTOp(ConvertVal($Vars, Value.op(0)$),
          ConvertVal($Vars, Value.op(1)$));
```

SMT-LIB supports floating-point values of arbitrary width (and even of arbitrary exponent and significand widths), but LLVM supports only a few fixed floating-point widths. We therefore limit our translation to the standard 16-, 32-, 64-, and 128-bit floating-point types with exponent and significand sizes listed above.

Example 1. *An SMT constraint C with variables*

```
(declare-fun a () Bool)
(declare-fun b () (_ BitVec 64))
(declare-fun c () (_ FloatingPoint 8 24))
```

is translated to an LLVM function with the following signature:

```
define i1 @C(i1 %a, i64 %b, float %c).
```

Variables and constants. During frontend translation, we give variables the same names in L as in C. It is also straightforward to translate boolean and bitvector constants, which have the same representation in LLVM as in SMT-LIB. The floating-point values ± 0, $\pm\infty$, and NaN are translated to their LLVM representations; all other floating-point values are constructed from bitvectors.

Simple operations. Many operations have the same semantics in LLVM and SMT-LIB; we list these here without detailed proof. SLOT translates these operations by simply applying the equivalent LLVM operation to the same arguments.

- The boolean functions and, or, and xor have the same names and semantics in LLVM and SMT-LIB. ($\Rightarrow$ a b) is reduced to (or (not a) b).
- Bitvector and floating-point comparisons (including = for bitvectors and fp.eq for floating-point) are equivalent to the LLVM icmp and fcmp instructions with the appropriate condition codes (*ordered* for floating-point).
- The SMT-LIB function ite is equivalent to LLVM select.
- Zero extension and sign extension are equivalent in SMT-LIB and LLVM.

- The bitvector math operations `bvadd`, `bvsub`, and `bvmul` and foating point math operations `fp.neg`, `fp.add`, `fp.sub`, `fp.mul`, `fp.div`, and `fp.rem` have the same semantics as the similarly named LLVM instructions.
- The SMT-LIB `not` and `bvnot` operations are equivalent to the LLVM `xor` instruction with the second argument having all bits set (*i.e.*, -1).
- `to_fp` on a single bitvector argument is equivalent to the LLVM `bitcast` instruction. `to_fp` on a floating-point argument is equivalent to either `fpext` or `fptrunc`, depending on the relative widths. `to_fp` on a rounding mode and a bitvector (*i.e.*, signed numeric conversion to floating-point) has the same semantics as `sitofp` in LLVM. Similarly, `to_fp_unsigned` is equivalent to `uitofp`.
- The SMT-LIB conversions `fp.to_ubv` and `fp.to_sbv` are equivlent to the LLVM instructions `fptoui` and `fptosi`, respectively.

In addition to functions with equivalent LLVM *instructions*, we express several SMT-LIB functions using LLVM *intrinsics*. These are common LLVM functions invoked using the `call ⟨intrinsic⟩` syntax. In the context of SLOT, there is no cost to using intrinsics instead of instructions, as we do not use the LLVM IR to generate a binary.

- SMT-LIB bit rotation is equivalent to LLVM's funnel shift intrinsics. For example, `((_ rotate_left` i`)` a`)` becomes `call i`n `@llvm.fshl.i`n`(i`n `%a, i`n `%a, i`n i`)`.
- Floating-point *fused multiply-add* (FMA), square root, and absolute value have the same names and semantics in SMT-LIB and LLVM (as intrinsics).
- `fp.min` and `fp.max` are equivalent to the `llvm.minnum` and `llvm.maxnum` intrinsics, respectively. These match the SMT-LIB semantics in that if one argument is `NaN`, the other argument is returned.
- The SMT-LIB floating-point class predicates like `fp.isNaN`, `fp.isInfinite`, etc. are equivalent to the `llvm.is.fpclass` intrinsic. This intrinsic takes a bitmask representing which classes to check for–each of the SMT-LIB predicates can be represented with the flags.

Division and bit shifting. There are several functions whose SMT-LIB and LLVM versions differ in subtle ways because the LLVM version has undefined behavior on some inputs. Ensuring that translation handles these cases correctly is the key challenge in implementing SLOT. In SMT-LIB, bitvector division by 0 is defined as a fixed value depending on the dividend. In LLVM, it produces a poison value, which is propagated by the optimizer through all subsequent operations. To build an equivalent series of LLVM instructions, we must add a check for this case.

Figure 4 shows the frontend translation of the four SMT-LIB bitvector division and remainder operations. In each case, SLOT adds a check comparing the divisor to zero, and then chooses either the result of an LLVM math operation or a constant as defined in the SMT-LIB standard. Signed division (Figure 4b) may produce either 1 or -1 depending on the signs of the inputs. The `bvsmod` operation is translated to a computation in terms of `urem`.

In addition to different handling of division by 0, LLVM and SMT-LIB have different semantics for bitvector shift operations. In

```
%zero = icmp eq in %b, 0
%div = udiv in %a, %b
%out = select  i1 %zero, in -1, in %div
```

(a) LLVM equivalent of (`bvudiv` a b)**.**

```
%zero = icmp eq in %b, 0
%neg = icmp slt in %a, 0
%const = select  i1 %neg, in 1, in -1
%div = sdiv in %a, %b
%out = select  i1 %zero, in %const, in %div
```

(b) LLVM equivalent of (`bvsdiv` a b)**.**

```
%zero = icmp eq in %b, 0
%rem = {u,s}rem in %a, %b
%out = select  i1 %zero, in %a, in %rem
```

(c) LLVM equivalent of (`bv{u,s}rem` a b)**.**

Figure 4: LLVM equivalents of SMT-LIB division and remainder.

```
%wide = icmp uge in %b, n
%shift = shl in %a, %b
%out = select  i1 %wide, in 0, in %shift
```

(a) LLVM equivalent of (`bvshl` a b)**.**

```
%wide = icmp uge in %b, n
%shift = lshr in %a, %b
%out = select  i1 %wide, i64 0, in %shift
```

(b) LLVM equivalent of (`bvlshr` a b)**.**

```
%wide = icmp uge in %b, n
%neg = icmp slt in %a, 0
%const = select  i1 %neg, in -1, in 0
%shift = ashr in %a, %b
%out = select  i1 %wide, in %const, in %shift
```

(c) LLVM equivalent of (`bvashr` a b)**.**

Figure 5: LLVM equivalents of SMT-LIB shift.

LLVM, shifts by the bit width or more have undefined behavior. In SMT-LIB, they always result in a bitvector of all 0s or all 1s (in the case of arithmetic shift right of a negative value). The translations of these three shift operations are shown in Figure 5.

Floating-point equality. LLVM floating-point math does not have undefined behavior like that of integers, but it does have two distinct notions of equality: `fp.eq` and "=". `fp.eq` checks for floating-point equality in the IEEE sense; it has the same semantics as LLVM's `fcmp oeq`. SMT-LIB "=", on the other hand, is a core theory operation that checks for the equality of two SMT expressions. Unlike IEEE-754, every value must be uniquely represented in SMT-LIB, so there is one `NaN` "object" which is equal to any other `NaN`. In all other cases, "=" means bitwise equality–this translation is shown in Figure 6.

Changing bit widths. There are three bitvector operations that change the bit widths of their arguments: concatenation (combining multiple bitvectors), repeat (repeating a single bitvector a constant number of times), and bit extraction. Each of these operations is translated to a sequence of LLVM instructions that simulate their semantics. Suppose we have a bitvector a of length n and want to extract the bits from i down to j (both inclusive in SMT-LIB).

```
%bca = bitcast float %a to in
%bcb = bitcast float %b to in
%nana = call i1 @llvm.is.fpclass.fn(fp %a, i32 3)
%nanb = call i1 @llvm.is.fpclass.fn(fp %b, i32 3)
%both = and i1 %nana, %nanb
%eq = icmp eq in %bca, %bcb
%out = or i1 %both, %eq
```

Figure 6: LLVM equivalent of (= a b) for floating-point values. `fp` **indicates** `half`**,** `float`**,** `double`**, or** `fp128` **with width n. The constant** `i32 3` **indicates a check for** `NaN`**.**

Intuitively, to extract this portion of a, we move the bits of interest to the right end of the bitvector (a shift by j), and then truncate to the appropriate size ($i - j + 1$).

Concatenation involves extending both arguments to the new width, shifting one into the newly-added all-zero bits of the other, and then combining with bitwise *or*. The SMT-LIB ((_ repeat n) a) operation is translated by chaining multiple concatenations. The number of repetitions is a constant parameter, and is therefore known statically. The translations of concatenation and repetition may add redundant overhead for some inputs, but any such overhead is eliminated during the optimization phase.

Floating-point construction. The SMT-LIB floating-point constructor `fp` takes a bitvector each for the sign, exponent, and significand and returns a floating point value. In LLVM, we need to concatenate (shift and bitmask) the three parts and interpret (*i.e.,* `bitcast`) the result as a floating-point value. This translation is shown in Figure 7.

Rounding modes. SMT-LIB defines five separate floating-point *rounding modes*: `roundNearestTiesToEven`, `roundNearestTiesToAway`, `roundTowardPositive`, `roundTowardNegative`, and `roundTowardZero`. These modes specify the semantics of floating-point operations like addition and subtraction when rounding is required. By default, LLVM floating-point instructions follow round to nearest with ties to even, so SLOT translates SMT function applications with this rounding mode directly to LLVM instructions. For other rounding modes, the tool must generate an LLVM call to a *constrained floating-point intrinsic* in LLVM, which in most cases allows the specification of rounding mode.

Example 2. *The SMT operation with 32-bit floating-point variables a and b*

```
(fp.add roundTowardPositive a b)
```

is translated to the following:

```
call float @llvm.experimental.constrained.fadd.f32(
    float %a, float %b, metadata !"round.upward",
    metadata !"fpexcept.ignore")
```

However, constrained floating point intrinsics do not exist for all SMT-LIB operations; in these cases, SLOT chooses the correct intrinsic based on rounding mode. For example, SMT-LIB conversion from floating-point to signed bitvecotrs `fp.to_sbv` becomes one of `constrained.roundeven`, `constrained.lround`, `llvm.ceil`, `llvm.floor`, or `fptosi`, depending on the rounding mode. Analagous measures are required for `fp.to_ubv` and `fp.roundToIntegral`.

```
%zsg = zext i1 %sign to in
%zex = zext ie %exp to in
%zsi = zext is %sig to in
%ssg = shl in %zsg, n − 1
%sex = shl in %zex, e
%right = or in %sex, %zsi
%all = or in %ssg, %right
%out= bitcast in %all to fptype
```

Figure 7: LLVM equivalent of (`fp` **sign exp sig). n is the width of the floating-point values, e is the width of the exponent, and s is the width of the significand.** `fptype` **may be any of** `half`**,** `float`**, or** `double`**.**

4.3 Backend translation

Simple operations, variables, and types. During backend translation, the straightforward operations listed in Section 4.2 can be translated just as during frontend translation. Function arguments of L' are converted to variables in C' with the same names and types. LLVM's optimizer may add or remove intermediate SSA variables in L, but only the function arguments are converted to variables in C. The optimizer may render on the the function arguments dead; in this case, it is not translated back into a variable. Therefore, the set of variables in C' is a subset of the variables in C. Bitvector and floating point types are also translated as during frontend translation. However, LLVM does not distinguish between booleans and 1-wide bitvectors, while SMT-LIB does. In most cases, this distinction is immaterial, and we treat `i1` as a boolean, but where the optimizer introduces bitvector operations on an `i1` (for instance, sign extension), we convert the argument to a bitvector, rather than a boolean.

Undefined behavior. During frontend translation, great care must be taken not to introduce undefined behavior into L. This is because the SMT versions of operations are more strictly defined than those in LLVM; in other words, the SMT bitvector division, for instance, matches the outputs of LLVM division on all inputs, but not the reverse. LLVM optimization does not introduce any undefined behavior, so, during backend translation, we need not insert any checks around operations like division and shifting. There is one LLVM operation that is undefined in SMT-LIB: bitcast conversions from floating-point values to integers. SLOT handles these conversions by introducing an extra integer variable and constraining the result of converting it to a floating-point.

Bit operations intrinsics Frontend translation produces only those intrinsics listed in Section 4.2. However, the optimizer may introduce other intrinsics which must be handled by backend translation. The `llvm.bswap` intrinsic swaps the lowest and highest *bytes* of its input, and is translated to a sequence of extract and concatenate operations representing these semantics. Similarly, the `llvm.bitreverse` intrinsic reverses all of the bits; this is also achieved by composing extraction and concatenation. Finally, the `llvm.ctpop` intrinsic counts how many bits are set (have value 1) in a bitvector; this is also achieved through several extractions followed by addition.

Math intrinsics. LLVM includes the intrinsics `umin`, `umax`, `smin`, and `smax` to take the signed minimum, unsigned maximum, signed minimum, and signed maximum, respectively, of two bitvector arguments. These intrinsics are translated into an SMT-LIB `ite` operation with the appropriate comparison. For example, a `umin` call on bitvector arguments a and b is translated to the SMT-LIB expression (`ite` (`bvult` a b) a b).

In addition, LLVM includes "saturated" math operations like `llvm.usub.sat`. These instructions prevent over- and underflow by clamping the return value to 0 if underflow would have occurred. Like the minimum and maximum operations, these intrinsics are translated to a `ite` expression. Rounding mode intrinsics produced by frontend translation must also be converted back to the corresponding SMT function with the correct rounding mode argument.

4.4 Preservation of satisfiability

We show that, for bitvector and floating-point constraints, the satisfiability of the original SMT constraint is preserved through frontend translation (Lemma 1), optimization, and backend translation (Lemma 2).

Property 1 (Optimizer semantics preservation). *Given an input LLVM function L which does not contain any undefined behavior, the result of optimizing L, call it L', has the same semantics as L. That is, for all inputs $l_1, l_2, \dots, l_n$, $L(l_1, l_2, \dots, l_n) = L'(l_1, l_2, \dots, l_n)$.*

This property of the optimizer may not always hold because the optimizer may contain bugs. But in our work, we take it as ground truth that the input and output of the optimizer are equivalent. Because we focus on relatively simple optimizations (*e.g.,* we do not deal with memory operations), compiler bugs changing our results are likely to be rare. In our testing of more than 100,000 SMT benchmarks, we have encountered no compiler bugs. We now prove that semantic preservation of SLOT.

Lemma 1 (Frontend translation). *Let C be an SMT constraint with variables $c_1, c_2, \dots, c_n$, and L be the function produced by the frontend translation of C. Then C is satisfiable if and only if there exists an input to L for which L returns true.*

Proof. ($\Rightarrow$) Assume C is satisfiable. Then there exists an assignment of the variables of C, $X = \{x_1, x_2, \dots, x_n\}$ for which C evaluates to true. Let L denote the LLVM function resulting from frontend translation. From Section 4.2, at each instruction i in L, the value produced by i is the same as the value produced by the corresponding function application in C. In particular, with the assignment X, C's outermost function application should produce `true`. This means that the last instruction in L must return `true`.

($\Leftarrow$) Assume that there is an input $X = \{x_1, x_2, \dots, x_n\}$ such that $L(X)$ is `true`, and consider whether the assignment of the values X to the variables of C satisfies C. By the translations in Section 4.2, for each instruction i, the equivalent function application in C yields the same value. In particular, we assume that the last instruction in L returns `true`; this corresponds to the final result of evaluating C, so the assignment X must satisfy the constraint C. □

Lemma 2 (Backend translation). *Let L be an LLVM function over integer (bitvector) types with n arguments and let C be the SMT constraint resulting from performing backend translation on L. Then, C is satisfiable if and only if there exists a set of inputs $l_1, l_2, \dots, l_n$ such that $L(l_1, l_2, \dots, l_n)$ returns* `true`.

Proof. ($\Rightarrow$) Assume that there exists an input $l_1, l_2, \dots, l_n$ on which L returns `true`. Let the set of values of the internal SSA variables of L under the given input be $v_1, v_2, \dots, v_l$, and call $Y = \{l_1, l_2, \dots, l_n, v_1, v_2, \dots, v_l\}$ the set of all variables from L. At each instruction i in L, the corresponding function application in C gives the same value as i. In particular, because L returns `true`, the last instruction, and so the result of evaluating C, must be `true`, which means that the assignment Y satisfies the constraint C.

($\Leftarrow$) Assume that C is satisfiable. This means that there exists a set of variables $Y = \{y_1, y_2, \dots, y_k\}$ for which C produces `true`. Now take the subset of Y, which corresponds to the input variables of L. At each instruction in L, we know the value must be the same as the corresponding SMT function application. In particular, the outermost function of C corresponds to L's return instruction, so since the assignment Y caused C to evaluate to `true`, L must also return `true`. □

Theorem 1 (Preservation of satisfiability). *Given an SMT constraint C on floating points and bitvectors, the new constraint C' produced by SLOT is satisfiable if and only if C is satisfiable.*

Proof. ($\Rightarrow$) Let L be the LLVM function produced by frontend translation of C, L' be the result of optimizing L, and C' be the result of conducting a backend translation of L'. Assume C is satisfiable. Then by Lemma 1, there exists an input on which L returns `true`. Moreover, from Section 4.2, we know that L contains no undefined behavior. Therefore, by Property 1, there is an input to L' such that L' also returns `true`. But then, by Lemma 2, C' is satisfiable.

($\Leftarrow$) Assume that C' is satisfiable. Then by Lemma 2, there exists an input on which L' returns `true`. Again, we know that L has no undefined behavior by the lemmas in Section 4.2, so by Property 1, there is an input to L for which L returns `true`. But then, by Lemma 1, C is satisfiable. □

Theorem 1 means that the sequence of translation, optimization, and translation described in this section produces a new constraint that has the same satisfiability as the original. Moreover, because of the construction of the translation, SLOT also preserves models between C and C'. That is, if C is satisfiable, an assignment that satisfies C' directly gives an assignment that satisfies the original constraint—we just ignore the extra variables introduced by the translation and optimization process. The theoretical guarantee of Theorem 1 gives us a practical, solver-agnostic tool for preprocessing and optimizing SMT constraints.

5 Evaluation

We evaluate SLOT by applying it to the SMT-LIB benchmark suites for the subject theories [3]. We highlight our most important results as follows.

- SLOT increases the number of solvable formulas at specified timeouts by up to 23% for bitvector-only benchmarks, 14% for floating-point-only benchmarks, and 80% for mixed benchmarks, allowing the solving of all but one `QF_BV` benchmark within 600 seconds.
- On average, SLOT slows down the smallest benchmarks but speeds up the largest benchmarks. Geometric mean speedups are up to 2.7× for bitvector-only benchmarks and over 3× for floating-point benchmarks.

Table 2: Timeout improvement results produced by SLOT. Each column denotes a different time limit with the total number of original unknown formulas ("Total"), the number improved ("Imp."), and the percentage ("%"). The "All" rows denote the number of formulas for which all solvers timed out, but at least one of the solvers produced a solution after SLOT was applied.

		600			300			120			60			30		
Benchmark	Solver	Total	Imp.	%	Total	Imp.	%	Total	Imp.	%	Total	Imp.	%	Total	Imp.	%
QF_FP	Z3	129	11	8.5	146	12	8.2	161	10	6.2	189	10	5.3	199	6	3.0
	CVC5	58	8	13.8	74	7	9.5	90	7	7.8	110	8	7.3	124	6	4.8
	All	55	5	9.1	74	7	9.5	90	7	7.8	109	7	6.4	123	5	4.1
QF_BVFP	Z3	19	6	31.6	28	11	39.3	39	7	18.0	58	7	12.1	95	20	21.0
	CVC5	6	4	66.7	6	4	66.7	8	3	37.5	14	3	21.4	35	13	37.1
	All	5	4	80.0	5	3	60.0	7	2	28.6	13	2	15.4	28	9	32.1
QF_BV	Z3	2747	481	17.5	3217	457	14.2	3954	639	16.2	4503	666	14.8	5248	593	11.3
	CVC5	2081	233	11.2	2325	294	12.7	2846	383	13.5	3866	742	19.2	4633	829	17.9
	Boolector	1691	242	14.3	2012	302	15.0	2528	320	12.7	3087	467	15.1	3884	499	12.9
	All	757	99	13.1	886	107	12.1	1115	113	10.1	1477	223	15.1	2138	488	22.8

- Most of SLOT's speedup is the result of just a few simple LLVM optimization passes like `instcombine`. Our approach shows which optimizations are "missing" from SMT solvers and allows the effort involved in developing these passes to be instantly available in the SMT context.

5.1 Experimental setup

Given a solver, a benchmark, and a timeout T^*, we follow a three-step process to test the effectiveness of SLOT. First, we measure how long it takes for the solver to conclude either `sat` or `unsat` for the benchmark; call this T_{pre}. Then, we apply SLOT to the benchmark, producing a new SMT constraint; we call the time it takes to do this T_{SLOT}. Finally, we measure how long the solver takes to solve the optimized benchmark, T_{post}. For a fair comparison, we must offset the overhead of running SLOT against the speedup achieved. Thus, a formula has been improved if $T_{\text{SLOT}} + T_{post} < T_{pre}$. This is the *SLOT-only* result. In addition, we adopt the portfolio methodology [41]; by running SLOT optimization in parallel with a solver, a user can simply take whichever result is produced first. When discussing speedups, we report this (*i.e.*, $\min\{T_{pre}, T_{\text{SLOT}} + T_{post}\}$) as the *portfolio* result. We report all proportional speedups as geometric means, reducing the impact of large outliers. We answer the following research questions:

- **RQ1: How many more formulas can be solved?** Given a time limit, how many formulas from benchmark sets can SLOT convert from unknown to either `sat` or `unsat`?
- **RQ2: How much faster can formulas be solved?** What is the proportional speedup produced by SLOT for constraints with low and high original solving times?
- **RQ3: Which LLVM optimization passes contribute?** Which optimization strategies in LLVM are most effective at simplifying SMT formulas beyond the capabilities of existing solvers?

Implementation. We have implement SLOT in about 2,500 lines of C++ and make it publicly available on Github[2]. We use Z3's built-in parser for the SMT-LIB language and the standard LLVM C++ API, but provide input and output in the standard SMT-LIB format for use with solvers other than Z3. Frontend and backend translation is carried out as described in Section 4. SLOT has been tested with LLVM version 16.0.0.

[2]https://github.com/mikekben/SLOT

Benchmarks. We use as test cases the standard solver benchmarks provided by the SMT-LIB developers for floating-point numbers (`QF_FP`, $n = 40,407$), bitvectors (`QF_BV`, $n = 46,191$), and mixed floating-point and bitvector constraints (`QF_BVFP`, $n = 17,249$) [3]. As discussed in Section 4, we restrict floating-point variables to the standard 16-, 32-, 64-, and 128-bit widths exclude any constraints which contain variable rounding modes. These limits are minimal: only 26 benchmarks from `QF_FP` (all for variable rounding mode), and 128 `QF_BVFP` benchmarks (89 for unsupported widths and 39 for variable rounding modes) are excluded, amounting to just 0.2% of all mixed and floating-point benchmarks.

Solvers. We test with the state-of-the-art general SMT solvers Z3 and CVC5 used in prior literature [40, 43]. For the bitvector-only benchmarks, we also test with Boolector [35], a solver specifically optimized for bitvectors [42]. SLOT has been tested with Z3 version 4.12.1, CVC5 version 1.0.5, and Boolector version 3.2.2.

Testing environment. All experiments are performed on a server with two AMD EPYC 7402 CPUs and 512GB RAM, running Ubuntu 20.04. We test with timeouts between 30 and 600 seconds, in line with those used in applications for translation validation [25] (zero to five minutes) and symbolic execution (between five and 129 solver calls within one hour) [12]. Finally, when measuring speedups, we count solver and SLOT timeouts as 600-second contributions.

5.2 RQ1: How many more formulas can be solved?

Table 2 shows the number of constraints which are changed from unknown to solved by SLOT for each of the three benchmark sets. The total column denotes the number of unknown benchmarks at each timeout, and the improved column ("Imp.") gives the number of constraints, from those, which can be solved after SLOT is applied. The results include SLOT's running time, *i.e.,* we report a benchmark as improved only if $T_{post} + T_{\text{SLOT}} < T^*$. Since solvers are typically run with a fixed timeout, *e.g.*, during symbolic execution [12], the proportion of constraints which move from timeout to solved at fixed values of T^* represents an improvement for users.

SLOT is most effective at speeding up mixed constraints; it allows all but one mixed benchmark to be solved within 600 seconds and reduces the number of unsolvable constraints by about one third at all time limits. renders solvable roughly 10% of timeout floating-point constraints, and 15%0-20% of bitvector benchmarks. The results are comparable for each of the tested solvers, showing that 's speedup is not solver-specific.

Most importantly, SLOT not only improves each solver's performance, but also does better than all solvers combined. The "All" rows in Table 2 show the number of benchmarks for which all of the solvers timed out and the number which became possible to solve with at least one of the solvers. The improvements in these rows show that outperforms even a portfolio of existing solvers, decreasing the number of unknown constraints by as much as 22% for small bitvector benchmarks.

5.3 RQ2: How much faster can formulas be solved?

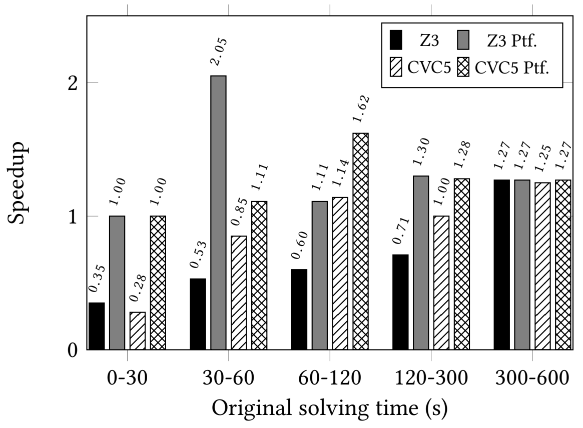
Figure 8 shows the mean speedups observed for each benchmark set. Values below one indicate a slowdown. For the smallest benchmarks, SLOT slows down solving, often substantially. However, while the proportional slowdown is large, the absolute slowdown is typically small, and occurs because the overhead of translating outweighs the cost of simply solving the benchmark (*i.e.*, $T_{\text{SLOT}} > C_{pre}$). For example, one benchmark[3] with Z3 is sped up from 0.06 seconds to 0.02 seconds, but SLOT takes 0.24 seconds to translate and optimize it.

The effect reverses for more complex constraints: for constraints that take longer than 300 seconds, we improve mean solving time by more than 1.25× for floating-point, about 1.6× for mixed, and between 1.4× and 1.7× for bitvector benchmarks. The portfolio methodology yields even greater running time improvements, in the range of 3× for `QF_BVFP` and 2× for `QF_BV`. Even small constraints below 60 seconds initial running time see appreciable speedups under all solvers with the portfolio method. The difference between the SLOT-only and portfolio results exists because the dramatic speedup of some constraints is offset by a slowing down of others.

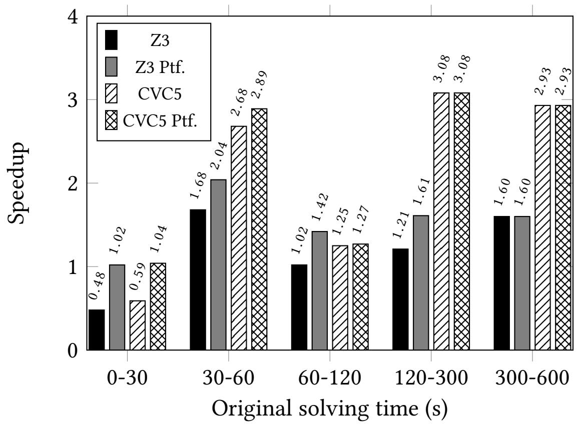
5.4 RQ3: Which LLVM optimization passes contribute?

To understand why SLOT produces performance improvements, we investigate which LLVM passes contribute most to the underlying results. The structure of SMT constraints means that most LLVM optimization passes are irrelevant to SLOT. Translated SMT constraints differ from most programs in that:

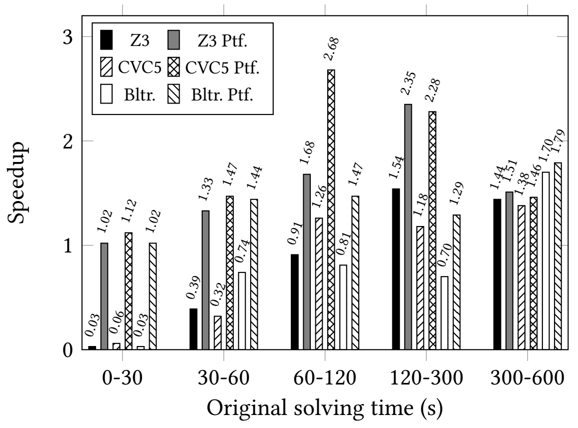
- They perform no memory operations–SMT variables are directly translated into LLVM function arguments.
- They have only one function; this maintains the equivalence definitions described in Section 4.
- The single function has only one basic block (*i.e.*, there is no branching). This is a consequence of the nature of SMT



(a) `QF_FP`



(b) `QF_BVFP`



(c) `QF_BV`

Figure 8: Geometric mean speedup from original constraint to optimized constraint produced by SLOT for each benchmark set under Z3, CVC5, and Boolector (for `QF_BV`). Constraints are grouped into ranges of original solving time along the x-axis. All measurements include T_{SLOT}.

[3] QF_BV/Sage2/bench_7588.smt2

constraints; the only branch–like operation is `ite`, which is translated to an LLVM `select` instruction.

The majority of LLVM's 58 optimization passes affect only memory operations (7), are interprocedural (10), or optimize branching (17). An additional 16 passes do not apply to translated SMT constraints for a variety of other reasons (they are architecture-specific, they optimize debug information, etc.). We also exclude `bb-vectorize` because vectorization introduces substantial translation overhead while providing no benefit in the SMT context. This leaves eight passes that are relevant to SLOT: `instcombine` (regular and aggressive), `instsimplify`, dead code elimination (regular and aggressive), global value numbering, `reassociate`, and sparse conditional constant propagation (SCCP). SLOT runs these passes in the same order in which they are performed during LLVM `O3` optimization.

Table 3 shows how many benchmarks each optimization pass affects; a benchmark is counted for a pass if pass application caused any change to the LLVM IR function. Agressive dead code elimination did not change any constraints, and aggressive `instcombine` only changed a few; this may be because, for the SMT context, their features are usually handled by the non-aggressive versions. The most effective passes are `instcombine`, which changes almost every constraint, `reassociate`, and global value numbering (for `QF_BV`). Notably, many more passes change bitvector benchmarks than floating-point; this is a combined result of lower structural complexity in the `QF_FP` and `QF_BVFP` benchmark sets and greater difficulty in optimizing floating-point operations.

Table 4 shows the mean speedup observed for benchmarks which were and were not affected by each optimization pass for `QF_BV` with initial timeouts above 30 seconds; the results for floating-point and mixed benchmarks are comparable, but have small sample size. The results confirm that `instcombine` speeds up benchmarks the most, followed by global value numbering and `reassociate`.

The `instcombine` pass consists mostly of simple peephole optimizations, which shows that solver performance could benefit from simple theory-specific reasoning which is already implemented in compilers. Global value numbering and reassociation also contribute substantially; while some solver heuristics already eliminate common subexpressions, more advanced implementations of these algorithms in LLVM provide further benefit. Our results show that the extensive effort expended to perfect compiler optimizations indeed provides benefit beyond that available in existing SMT solvers. Using SLOT, solver users can benefit from that effort without deep knowledge of solver implementation.

6 Discussion

Compiler optimization vs. SMT simplification. Because the purpose of compiler optimizations is to reduce the number of processor instructions, some operations which are "simpler" in LLVM may not provide any advantage in SMT. For example, bitvector multiplication by 2^n is equivalent to shifting left by n; however, Z3 takes much longer to solve constraints involving shifts. On one benchmark[4] for example, Z3 takes less than a second if doubling is expressed as $a + a$ or $2 \times a$, but does not finish within 24 hours if it is expressed as shift left by one.

[4] QF_BV/20190311-bv-term-small-rw-Noetzli/bv-term-small-rw_1244.smt2

Table 3: Percentage of benchmarks affected by each optimization pass.

Pass	QF_FP	QF_BVFP	QF_BV
instcombine	99%	100%	78%
reassociate	78%	57%	26%
gvn	<1%	<1%	44%
sccp	0%	<1%	17%
dce	0%	<1%	17%
instsimplify	0%	<1%	16%
aggressive-instcombine	0%	0%	<1%
adce	0%	0%	0%

Table 4: Mean speedups for benchmarks which are and are not affected by each pass from the `QF_BV` benchmark set with initial solving time above 30 seconds under Z3. The spread is the difference in mean speedup between benchmarks which are affected by the pass and those which are not.

Pass	Count	Speedup without	Speedup with	Spread
instcombine	4,031	1.0×	1.83×	0.83
gvn	3,816	1.08×	1.85×	0.77
reassociate	2,168	1.35×	2.02×	0.67
sccp	1,360	1.51×	1.86×	0.35
instsimplify	1,562	1.53×	1.75×	0.22
dce	1,705	1.56×	1.68×	0.12
aggressive-instcombine	8	1.6×	1.22×	-0.38

In our implementation of SLOT, we provide a flag to force backend translation to generate multiplication, rather than shift, where possible. However, there may be more complex analogous examples arising from the fundamentally different purpose of LLVM. So while compiler optimization unlocks logic not present in existing SMT solvers, it acts more as a sieve than as a magic bullet. Running SLOT and a solver as a portfolio allows solver heuristics and SLOT each to shine where they perform best, each doing well on some benchmarks but slowing down on others. Only those benchmarks that neither can handle slip through the sieve.

SLOT overhead. The main driver of the proportional slowdown for small constraints is the cost of running SLOT. In contrast to a solver, which just needs to parse the constraint, SLOT must parse, translate, optimize, translate again, and then write. The performance of SLOT's algorithms is roughly linear in the size of the AST of the original constraint, while SMT solving has unpredictable, possibly exponential performance. This means that the proportion of $T_{\text{SLOT}} + T_{\text{post}}$ contributed by SLOT generally decreases as T_{pre} increases, as shown in Table 5.

The moderate increase for `QF_BV` benchmarks above 300 seconds is a result of SLOT timeouts; these are counted as 100% contribution in Table 5. Frontend translation makes up about 60% of the running time of SLOT for bitvectors, and about one third for mixed and floating-point constraints while optimization contributes between

Table 5: $T_{\text{SLOT}}/(T_{\text{SLOT}} + T_{\text{post}})$ (as a percentage) for bitvector and floating-point benchmarks.

Time interval (T_{pre} using Z3)	QF_FP	QF_BV	QF_BV
0-30	32.21%	22.17%	47.29%
30-60	0.02%	0.16%	3.50%
60-120	0.01%	0.20%	1.49%
120-300	0.01%	0.09%	1.7%
300-600	0.02%	0.01%	2.49%

7% and 10%. Backend translation takes longer for constraints including floating-point numbers (about 60%) because floating point intrinsics require additional steps to be translated back to SMT-LIB.

Other SMT Theories. Program analysis tools make use of just a few SMT theories: bitvectors [25], floating point [24], and more recently, strings [6]. SLOT improves the performance of solvers on the theories relevant to these applications, and we leave to future work the extension of SLOT's general method to other theories like real numbers of use outside software engineering [22]. While the optimization process used in SLOT may provide benefit outside bitvectors and floating-point numbers, applying translation and optimization to other theories would require either a new conception of semantics preservation or substantial under-approximation, since compiler IRs are not expressive enough to model unbounded computation. The limitations imposed by under-approximating may also negate any benefit provided by compiler optimizations.

7 Related Work

SMT constraint transformation. Existing work presents a number of strategies for simplifying SMT constraints. Dillig *et al.* [17] introduce a solver-internal constraint simplification algorithm that preserves satisfiability. Reynolds *et al.* [37, 38] introduce simplifying transformations for unbounded string constraints. Transformations of SMT formulas are also employed to test solvers. StringFuzz [9] focuses on generating well-formed formulas, but also provides some transformations on string constraints. STORM [27] transforms boolean formulas to perform blackbox testing. Bugariu *et al.* [11] introduce constant assignment and term synthesis as transformations for string constraints. Sparrow [43] and YinYang [40] expand transformations to real numbers and integers. The transformations performed by these tools are designed to test solvers; our approach uses a related method to speed up solving instead.

Speeding up SMT solving. Most work on improving SMT solver performance focuses on algorithms to be implemented within a solver. In addition to Z3, CVC5, and Boolector, such work has taken the form of new solvers like MathSAT [14], Bitwuzla [31], and Yices [19]. Early work on improving solver performance used symmetry to reduce the constraint-solving search space [16]. More recently, Niemetz *et al.* introduced syntax-guided quantifier instantiation to speed up solving for quantified constraints [32]. For particular theories, Z3str3 speeds up solving of string constraints [5], and Berzish *et al.* introduce new methods for solving constraints involving regular expressions [6]. Sadhak [29] combines CVC4 with fuzzing techniques for uninterpreted functions to improve performance. FastSMT [1] uses a neural network to find better ways to combine existing solver heuristics, thereby speeding up solving. MBA-Solver [42] departs from solver-specific approaches by pre-processing bitvector constraints involving alternating bitwise and arithmetic operations. Our approach is most similar to MBA-Solver, as SLOT uses pre-processing rather than solver-internal improvements. However, it differs in that we apply the broad range of optimizations performed by LLVM, including floating-point transformations. We harness an existing source of optimizations as a black box rather than hand-crafting one for the SMT problem.

The constraint–code nexus. Work on symbolic execution and translation validation has used SMT constraints to represent the semantics of LLVM programs. KLEE [12] converts LLVM IR programs into SMT formulas that encode symbolic execution constraints; many symbolic execution tools are built on the LLVM-SMT core provided by KLEE [36]. Alive and its progeny [25, 26] generate SMT constraints from LLVM instructions to verify the optimizations performed by the LLVM optimizer, which Lee *et al.* [23] expands to LLVM's memory model. LifeJacket [33] and Alive-FP [28] use SMT formulas to verify floating-point computation. VeRA [10] also translates C++ code to SMT constraints for program verification, and faces some engineering challenges analogous to SLOT. Constraints in the formal refinement-based B-method have also been translated to SMT-LIB [20, 39]. These approaches use SMT solvers to reason about programs and compilers. SLOT does the opposite, using a compiler to reason about SMT problems and exactly preserving constraint semantics instead of solving analysis constraints.

Optimizations outside compilers. Dong *et al.* [18] apply compiler optimizations like those used in SLOT directly to KLEE. They find that those optimizations can slow down symbolic execution because the optimizer alters branching structure in a manner which complicates symbolic execution. LEO [13] attempts to remedy this limitation by using machine learning to choose which optimization passes to apply. SLOT's results differ because it operates at the level of constraints, not programs. This allows SLOT to work in contexts outside symbolic execution, and also to avoid analysis-frustrating branching optimizations. Steno [30] translates declarative queries into imperative code to speed up operations over collections. SLOT, on the other hand uses both front- and backend translation with the aim of simplifying the constraint rather than transforming it into executable code.

8 Conclusion

This paper has presented a general pre-processing tool, SLOT, which allows solver users to apply compiler optimizations to SMT constraints as a black-box. SLOT practically improves solvers' performance on standard benchmarks and increases the number of solvable constraints at fixed time limits. Furthermore, the speedup is achieved using only the simplest compiler optimization passes, giving solver developers insight into possible improvements to solver tactics.

Acknowledgments

The authors thank the anonymous reviewers for their feedback on earlier drafts of this paper. This work was supported, in part, by the United States National Science Foundation (NSF) under grants

No. 1917924, No. 2114627, and No. 2237440; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

References

[1] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. 2018. Learning to Solve SMT Formulas. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 10338–10349.

[2] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*. Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[4] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*. Vol. 31. Chapter 11, 305–343. https://doi.org/10.1007/s00165-019-00486-z

[5] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. IEEE, 55–59. https://doi.org/10.23919/FMCAD.2017.8102241

[6] Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. 2021. An SMT Solver for Regular Expressions and Linear Arithmetic over String Length. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*. Springer, 289–312. https://doi.org/10.1007/978-3-030-81688-9_14

[7] Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. 2023. Satisfiability Modulo Custom Theories in Z3. In *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13881)*. Springer, 91–105. https://doi.org/10.1007/978-3-031-24950-1_5

[8] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 643–659.

[9] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *Computer Aided Verification - 30th International Conference, CAV 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*. Springer, 45–51. https://doi.org/10.1007/978-3-319-96142-2_6

[10] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a verified range analysis for JavaScript JITs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 135–150. https://doi.org/10.1145/3385412.3385968

[11] Alexandra Bugariu and Peter Müller. 2020. Automatically testing string solvers. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1459–1470. https://doi.org/10.1145/3377811.3380398

[12] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 209–224.

[13] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to Accelerate Symbolic Execution via Code Transformation. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs, Vol. 109)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:27. https://doi.org/10.4230/LIPIcs.ECOOP.2018.6

[14] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*. Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

[15] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[16] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. 2011. Exploiting Symmetry in SMT Problems. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6803)*. Springer, 222–236. https://doi.org/10.1007/978-3-642-22438-6_18

[17] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Small Formulas for Large Programs: On-Line Constraint Simplification in Scalable Static Analysis. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6337)*. Springer, 236–252. https://doi.org/10.1007/978-3-642-15769-1_15

[18] Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. 2015. Studying the influence of standard compiler optimizations on symbolic execution. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 205–215. https://doi.org/10.1109/ISSRE.2015.7381814

[19] Bruno Dutertre. 2014. Yices 2.2. In *Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*. Springer, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

[20] Sebastian Krings and Michael Leuschel. 2016. SMT Solvers for Validation of B and Event-B Models. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9681)*. Springer, 361–375. https://doi.org/10.1007/978-3-319-33693-0_23

[21] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures - An Algorithmic Point of View*. Springer. https://doi.org/10.1007/978-3-540-74105-3

[22] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2014. Proving Non-termination Using Max-SMT. In *Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*. Springer, 779–796. https://doi.org/10.1007/978-3-319-08867-9_52

[23] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P. Lopes. 2021. An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*. Springer, 752–776. https://doi.org/10.1007/978-3-030-81688-9_35

[24] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. 2017. Floating-point symbolic execution: a case study in n-version programming. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 601–612. https://doi.org/10.1109/ASE.2017.8115670

[25] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. ACM, 65–79. https://doi.org/10.1145/3453483.3454030

[26] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 22–32. https://doi.org/10.1145/2737924.2737965

[27] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 701–712. https://doi.org/10.1145/3368089.3409763

[28] David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*. Springer, 317–337. https://doi.org/10.1007/978-3-662-53413-7_16

[29] Sujit Kumar Muduli and Subhajit Roy. 2022. Satisfiability modulo fuzzing: a synergistic combination of SMT solving and fuzzing. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1236–1263. https://doi.org/10.1145/3563332

[30] Derek Gordon Murray, Michael Isard, and Yuan Yu. 2011. Steno: automatic optimization of declarative queries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 121–131. https://doi.org/10.1145/1993498.1993513

[31] Aina Niemetz and Mathias Preiner. 2020. Bitwuzla at the SMT-COMP 2020. *CoRR* abs/2006.01621 (2020). arXiv:2006.01621

[32] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. 2021. Syntax-Guided Quantifier Instantiation. In *Tools and Algorithms for*

the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12652). Springer, 145–163. https://doi.org/10.1007/978-3-030-72013-1_8

[33] Andres Nötzli and Fraser Brown. 2016. LifeJacket: verifying precise floating-point optimizations in LLVM. (2016), 24–29. https://doi.org/10.1145/2931021.2931024

[34] Institute of Electrical and Electronics Engineers. 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

[35] Mathias Preiner, Aina Niemetz, and Armin Biere. 2017. Counterexample-Guided Model Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205).* 264–280. https://doi.org/10.1007/978-3-662-54577-5_15

[36] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* USENIX Association, 49–64.

[37] Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019. High-Level Abstractions for Simplifying Extended String Constraints in SMT. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562).* Springer, 23–42. https://doi.org/10.1007/978-3-030-25543-5_2

[38] Andrew Reynolds, Maverick Woo, Clark W. Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427).* Springer, 453–474. https://doi.org/10.1007/978-3-319-63390-9_24

[39] Joshua Schmidt and Michael Leuschel. 2022. SMT solving for the validation of B and Event-B models. *Int. J. Softw. Tools Technol. Transf.* 24, 6 (2022), 1043–1077. https://doi.org/10.1007/s10009-022-00682-y

[40] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020.* ACM, 718–730. https://doi.org/10.1145/3385412.3385985

[41] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. 2009. A Concurrent Portfolio Approach to SMT Solving. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643).* Springer, 715–720. https://doi.org/10.1007/978-3-642-02658-4_60

[42] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. 2021. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* ACM, 651–664. https://doi.org/10.1145/3453483.3454068

[43] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Skeletal approximation enumeration for SMT solver testing. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021.* ACM, 1141–1153. https://doi.org/10.1145/3468264.3468540

Received 2023-03-02; accepted 2023-07-27