SMT Theory Arbitrage: Approximating Unbounded Constraints using Bounded Theories

BENJAMIN MIKEK, Georgia Institute of Technology, USA
QIRUN ZHANG, Georgia Institute of Technology, USA

SMT solvers are foundational tools for reasoning about constraints in practical problems both within and outside program analysis. Faster SMT solving improves the performance of practical tools and expands the set of tractable problems. Existing approaches to improving solver performance either focus on general algorithms applied below the level of individual theories, or focus on optimizations within a single theory. Unbounded constraints in which the number of possible variable values is infinite, such as real numbers and integers, pose a particularly difficult challenge for solvers. Bounded constraints in which the set of possible values is finite, such as bitvectors and floating-point numbers, on the other hand, are decidable and have been the subject of extensive performance improvement efforts.

This paper introduces a theory arbitrage: we transform unbounded constraints, which are often expensive to solve, into bounded constraints, which are typically cheaper to solve. By converting unbounded problems into bounded ones, theory arbitrage takes advantage of better performance on bounded constraints and unlocks optimization techniques that only apply to bounded theories. The transformation is achieved by harnessing a novel abstract interpretation strategy to infer bounds. The bounded transformed constraint is then an underapproximation of the semantics of the unbounded original. We realize our method for the theories of integers and real numbers with a practical tool (STAUB). Our evaluation demonstrates that theory arbitrage alone can speed up individual constraints by orders of magnitude and achieve up to a 1.4× speedup on average across nonlinear integer benchmarks. Furthermore, it enables the use of the recent compiler optimization-based technique SLOT for unbounded SMT theories, unlocking a further speedup of up to 3×. Finally, we incorporate STAUB into a practical termination proving tool and observe an overall 9% improvement in performance.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: SMT, constraint solving, bound inference, abstract interpretation

ACM Reference Format:
Benjamin Mikek and Qirun Zhang. 2024. SMT Theory Arbitrage: Approximating Unbounded Constraints using Bounded Theories. *Proc. ACM Program. Lang.* 8, PLDI, Article 157 (June 2024), 26 pages. https://doi.org/10.1145/3656387

1 INTRODUCTION

Many problems relevant to program analysis are expressed with Satisfiability Modulo Theories (SMT) constraints and solved using SMT solvers. SMT solvers take as input constraints over some base theory and either provide a variable assignment that satisfies the constraint or prove that the constraint cannot be satisfied. SMT solving is central to program analysis applications such as symbolic execution [16, 47], program synthesis [12, 40], and program verification [10, 45]. The performance of these practical tools is directly tied to the performance of solvers; when solvers can handle problems quickly, the tools perform better.

Authors' addresses: Benjamin Mikek, bmikek@gatech.edu, Georgia Institute of Technology, Atlanta, GA, USA; Qirun Zhang, qrzhang@gatech.edu, Georgia Institute of Technology, Atlanta, GA, USA.

ACM 2475-1421/2024/6-ART157
https://doi.org/10.1145/3656387

Modern SMT solvers operate on constraints over both *bounded* and *unbounded* theories. We call a theory bounded if every sort it defines includes only finitely many values; we call a theory unbounded if it has at least one infinite sort. Unbounded theories include those of integers ($\mathbb{Z}$) and real numbers ($\mathbb{R}$), while bounded theories include those of bitvectors and floating-point numbers. Even though unbounded SMT theories represent theoretical mathematics and not machine arithmetic, they have seen wide use in program analysis applications. For example, unbounded real arithmetic is useful in modeling automata [19], while integer arithmetic has been used for termination proving [37], to encode the constant multiplication problem [48], and in developing better answer set programming techniques [65]. Unbounded theories also have applications outside computer science, including for geometry [43, 46] and in economics [54], among other fields.

Solving unbounded SMT constraints poses challenges from both practical and theoretical perspectives. Some theory-specific reasoning is symbolic [31, 58], but SMT solvers still face the challenge of modeling arithmetic unbounded in both magnitude and precision while being implemented in traditional programming languages where all datatypes are bounded. Thus, as the needs for precision and magnitude grow, so must the data structures used to represent values [15]. From a theoretical standpoint, the search space for unbounded theories is also infinite, compared to the finite search space for bounded theories. The satisfiability of nonlinear integer arithmetic constraints is even undecidable [24]. Both the theoretical complexity of the problem and the engineering issues faced by solver developers make it difficult to improve performance for unbounded theories.

SMT solvers contain extensive logic to handle bounded constraints. The SMT-LIB [4] theory of bitvectors represents integer values with two's complement semantics, while the theory of floating-point numbers represents rational values with IEEE-754 [38] semantics. These theories are used to reason about values like the 16-, 32-, and 64-bit integers and single- and double-precision floating-point values offered in most C-like languages. The limit on the number of possible values of bitvector and floating-point variables makes their constraints decidable and simplifies reasoning. However, bound imposition alone is not enough; as we show in Section 2, adding variable bounds on an integer constraint does not improve, and may even hinder, performance. The key benefit of bounded theories is that existing work has introduced many practical theory-specific techniques to improve performance. For instance, the structure of the bitvector theory admits many optimizing rewriting rules [61, 69] that improve the performance of counterexample-guided solving [62]. Bitvector and floating-point constraints have also been simplified using compiler optimizations [50]. The cumulative effect of decidability, boundedness, and existing solving techniques for bounded theories often gives solvers superior performance on bounded constraints. In our experiments, for example, it takes the SMT solver Z3 [25] between 1.8× and 5.5× longer on average to solve a nonlinear integer constraint than a bitvector one with equivalent operations.

This work proposes a novel approach, *SMT theory arbitrage*, to speed solving for unbounded theories. Theory arbitrage converts an original constraint S in an expensive-to-solve unbounded theory into a constraint S' with the same semantics, but in a bounded theory which is cheaper to reason about. The conversion is possible because the unbounded theories of integers and real numbers have bounded counterparts: bitvectors and floating-point numbers, respectively. The approach has three main advantages: it produces constraints that are bounded, linear, and decidable, and consequently simpler to solve; it unlocks existing simplification strategies, both within and outside solvers; and it is solver-agnostic and can be applied to any SMT-LIB-compliant solver.

The key challenge is how to convert to a bounded theory while maintaining correctness. As we show in Section 3, bound imposition faces a tradeoff: large bounds are more likely to be sufficient to correctly represent unbounded semantics but are slower to reason about, while small bounds result in faster solving but may not be sufficient to accurately model the unbounded computation. We balance these competing interests in two steps. First, we analyze the input SMT constraint

to find a bound which is sufficient with high probability, but not too large. Second, we introduce an underapproximation to guarantee run-through correctness, even if the selected bounds were insufficient. The first step is realized with a novel and general abstract interpretation strategy to infer bounds on the original SMT constraint. This strategy uses an abstract domain representing bit widths and is adaptable to the different notions of boundedness in integer and real number problems. The second step guarantees correctness through underapproximation. If the new constraint S' is unsatisfiable, we revert to the original constraint, providing no performance improvement. On the other hand, if S' is satisfiable, we check that it shares a satisfying assignment with S, and if it does, produce a significant performance improvement. This strategy guarantees overall correctness while speeding up solving: a subset of input constraints is dramatically sped up while performance for others does not change, leading to significant speedups on average over all constraints.

SMT theory arbitrage differs from existing approaches to improving solver performance because it transforms from one theory to another. Past work has focused on improving the theory-independent foundations of solvers [26, 29], or on heuristics and simplifications for single theories both within solvers [6–8] and during pre-processing [50, 69]. These approaches provide guarantees of correctness. In the first case, though, they cannot exploit all of the structure in an underlying problem, and in the latter case, they do not handle unbounded theories. Our approach converts from one theory to another to take advantage of performance differences. Some solvers already make use of under- and over-approximations, but our approach uses it to narrow the set of problems that can be handled to the correct ones, rather than using it to directly solve a constraint. Moreover, by transforming unbounded constraints into bounded ones, we unlock further existing optimizations that are specific to bounded theories.

We realize theory arbitrage as a practical tool for **S**MT **T**heory **A**rbitrage from **U**nbounded to **B**ounded constraints (STAUB). Our implementation speeds up SMT solving for integers (to bitvectors) and real numbers (to floating-point numbers)—other unbounded theories like strings and arrays lack a bounded counterpart in SMT-LIB. We perform an extensive three-part evaluation. First, we perform tests on the standard set of SMT benchmarks for linear and nonlinear quantifier-free integer and real arithmetic (SMT-LIB's `QF_LIA`, `QF_NIA`, `QF_LRA`, and `QF_NRA` logics) and find that STAUB provides mean speedups up to 1.4× while rendering tractable hundreds of constraints for which state-of-the-art solvers time out. Second, we evaluate the performance of STAUB in combination with prior work on speeding up the solving of bounded SMT constraints [50], and find that STAUB's transformation unlocks an additional 3× improvement in performance. Finally, we evaluate STAUB under pessimistic conditions with constraints generated by the client termination proving analysis Ultimate Automizer [36]. We find that it achieves a performance improvement of 9%, even in an application where most constraints are unsatisfiable.

In summary, we make the following contributions.

- We introduce a novel strategy, *SMT theory arbitrage*, which underapproximatively transforms unbounded SMT constraints into bounded ones, thereby speeding up solving.
- We realize our framework for the two unbounded SMT-LIB theories of integer and real arithmetic, introducing along the way an abstract interpretation strategy for bound inference that can handle each theory's different notion of boundedness.
- We show that our approach provides substantial performance improvements over state-of-the-art solvers and speeds up a client analysis.

The paper is structured as follows. Section 2 motivates SMT theory arbitrage. Section 3 formally states the problem, while Section 4 describes our conversion strategy, including bound inference through abstract interpretation. Section 5 presents our empirical evaluation, while Section 6 provides a discussion of our findings, Section 7 describes related work, and Section 8 concludes.

```
1 (declare-fun x () Int)
2 (declare-fun y () Int)
3 (declare-fun z () Int)
4 (assert (= (+ (* x x x) (* y y y) (* z z z)) 855))
5 (check-sat)
```

(a) An example unbounded nonlinear integer constraint taken from the SMT-LIB benchmark set (`QF_NIA/20220315-MathProblems/STC_0855.smt2`).

```
1 (declare-fun x () (_ BitVec 12))
2 (declare-fun y () (_ BitVec 12))
3 (declare-fun z () (_ BitVec 12))
4 (assert (not (bvsmulo x x)))
5 ...
6 (assert (= (bvadd (bvmul x x x) (bvmul y y y) (bvmul z z z)) (_ bv855 12)))
7 (check-sat)
```

(b) Transformation of the constraint into the bounded theory of bitvectors, with width 12. Note that additional assertions prohibiting overflow, in the form of that on Line 4, have been omitted for brevity.

```
1 (declare-fun x () Int)
2 (declare-fun y () Int)
3 (declare-fun z () Int)
4 (assert (and (<= x 2047) (>= x (- 2048))))
5 (assert (and (<= y 2047) (>= y (- 2048))))
6 (assert (and (<= z 2047) (>= z (- 2048))))
7 (assert (= (+ (* x x x) (* y y y) (* z z z)) 855))
8 (check-sat)
```

(c) Original constraint in the unbounded theory of integers with bounds imposed as additional constraints on the variables x, y, and z (Lines 4-6). The bounds match the 12-bit width in Figure 1b.

Fig. 1. Example translation of an unbounded integer benchmark to the bounded theory of bitvectors.

2 MOTIVATING EXAMPLE

This section provides a concrete example that demonstrates the benefits of theory arbitrage on a nonlinear integer SMT constraint.

Input SMT Constraint. Figure 1a gives a nonlinear integer constraint from the SMT-LIB benchmark set. The example checks whether three cubes can sum to the constant 855. It takes Z3 (version 4.12.3) 27.7 seconds to conclude that the constraint is satisfiable. One satisfying assignment is $x = 7, y = 8, z = 0$.

SMT Theory Arbitrage. Theory arbitrage refers to the conversation from the expensive integer constraint in Figure 1a to its bounded counterpart bitvector constraint in Figure 1b. The translation is achieved by traversing the constraint's syntax tree once. Variables are changed to the 12-bit bitvector type, and constants like 855 are mapped to their equivalent bitvector values. Integer operations like "+" and "*" are converted to their bitvector equivalents, *i.e.*, `bvadd` and `bvmul`. Additional constraints, shown on Line 4, are inserted to prevent overflow, preserving the semantics of the unbounded constraint. Z3 can solve the constraint in Figure 1b in just 0.1 seconds, a speedup by orders of magnitude.

The Benefit of a Bounded Theory. The performance improvement achieved by theory arbitrage derives from solver tactics, which are specific to bounded theories. Bound imposition unlocks these tactics but does not always speed up solving in and of itself. For example, Figure 1c shows the same constraint still in the unbounded theory of integers, but with bounds corresponding to the width 12. This constraint takes 26.3 seconds to solve, nearly the same as the original.

Challenges. The conversion from Figure 1a to Figure 1b produces an equivalent constraint on 12-bit bitvectors. However, choosing a proper width is challenging. If we select a standard 64-bit width, the constraint takes longer to solve, as shown in Figure 2. On the other hand, if we select an 8-bit width, the result of the multiplication $7 * 7 * 7$ cannot be represented, and the constraint becomes unsatisfiable. Bound selection thus faces a tradeoff; Section 4 introduces an abstract interpretation to infer widths and balance the competing interests of speed and correctness.

Underapproximation. While it is possible to pick a width wide enough to maintain the semantics of the specific constraint in Figure 1a, it is not possible to select a sufficient width in every case. We must, therefore, underapproximate the solution to the original constraint. If the transformed constraint is unsatisfiable, then it may be that the original constraint was `unsat`, or it may be that the chosen bounds were not sufficient. The two cases are indistinguishable, so we revert to the original constraint and provide no performance improvement. In the satisfiable case, as in Figure 1b, we can compare the satisfying assignment of the bounded constraint to the original to verify that the selected bounds were sufficient. This verification also handles any *semantic differences* between the bounded and unbounded theories, like integer overflow or floating-point rounding.

3 PRELIMINARIES

3.1 Definitions and Problem Statement

An SMT constraint is a first-order logical formula over one or more theories. While in the general case, the set of possible theories and the sorts they define is infinite, we focus on SMT as defined by the SMT-LIB standard [4]. Existing literature includes extensive formal descriptions of the SMT problem [5, 44]; here, we provide a brief overview relevant to our theory arbitrage approach.

Definition 3.1 (SMT Theory). An SMT theory is a pair $T = (\Sigma, A)$ where Σ is a signature consisting of sort symbols $S_1, S_2, \ldots$ and function symbols $F_1, F_2, \ldots$ and A is a class of Σ-interpretations.

A theory limits the set of possible interpretations of symbols in the signature Σ. Intuitively, a theory in SMT-LIB provides a set of *sorts* (*i.e.*, types) and a set of functions over values from those sorts. With a theory in hand, we can now define formulas.

Definition 3.2 (SMT Formula). Given a theory $T = (\Sigma, A)$ with signature Σ and interpretations A, an SMT formula ϕ is an expression made up of symbols (function applications, variables, or constants) from Σ. ϕ is satisfiable in T, or T-satisfiable, if there exists an interpretation in A that satisfies ϕ.

SMT formulas are a conjunction of several individual constraints; for simplicity of notation in this paper, we refer to constraints, as our strategy can be applied equally to individual assertions and to formulas that conjoin many constraints. In practice, the question of satisfiability for constraints is one of variable assignments. A constraint has a set of variables $v_1, v_2, \ldots$, each with a particular sort from T. If there exists an assignment $v_1 = x_1, v_2 = x_2, \ldots$ of the variables for which evaluating the constraint produces `true`, then it is satisfiable. If no such assignment exists, it is unsatisfiable.

The SMT-LIB standard defines eight theories: the core theory, arrays, bitvectors, floating-point numbers, integers, real numbers, integer/real conversions, and strings. The core theory defines the boolean sort and the standard set of boolean operations. It also defines equality and inequality comparisons, which apply to all sorts. From the functions defined in these eight theories, SMT-LIB also defines 29 *logics*; a logic is a combination of functions and sorts from one or more theories that form a coherent whole and for which specific solver strategies may apply [4]. Since we define satisfiability in terms of a boolean result, all logics rely on the core theory.

For constraints in some theories, there are infinitely many different possible assignments of each variable. Other theories have a finite (though typically large) set of possible values for each variable. We state this distinction formally in Definitions 3.3 and 3.4.

Definition 3.3 (Bounded Theory). A theory $T = (\Sigma, A)$ is bounded if for all $S \in \Sigma$, there exits $N \in \mathbb{N}$ such that $|S| < N$.

Definition 3.4 (Unbounded Theory). A theory $T = (\Sigma, A)$ is unbounded if it is not bounded, *i.e.*, if there exists $S \in \Sigma$ such that S has infinite cardinality.

According to Definitions 3.3 and 3.4, the theory of bitvectors is bounded because each bitvector sort allows exactly 2^n possible values, where n is the bit width. Floating-point numbers are bounded for the same reason. The *core theory* of boolean operations is also bounded because there are only two possible boolean values. On the other hand, the integer sort represents any member of $\mathbb{Z}$ while the sort of reals represents values from $\mathbb{R}$. These two sorts can take any of infinitely many values, so the theories of integer and real arithmetic are unbounded according to Definition 3.4. The theories of strings and arrays are also unbounded since arrays and strings can be of any length.

Our conversion targets are the theory of bitvectors, which represents machine integers with a fixed width, and floating-point numbers, which represent IEEE-754 floating-point values. The theory of floating-point numbers allows an arbitrary choice of exponent and significand widths, beyond those used in most programming languages. For ease of notation, we adopt Z3's notion of a *kind*; this is a higher-level sort used to associate similar sorts [25]. In particular, all bitvector sorts are of the bitvector kind, while all floating-point sorts are of the floating-point kind.

The goal of theory arbitrage is to speed up solving over the unbounded theories of integers and reals by converting them to bounded theories. We state the problem formally as follows.

> Given an unbounded SMT theory T and a constraint C over T, impose bounds on C and construct a new constraint C' over some theory T' such that
> - T' is a bounded theory; and
> - C' and C are equisatisfiable.

Transforming from C to C' requires the imposition of a bound on a constraint which, *a priori*, lacked one. Neither the structure of the SMT problem nor the theories defined in SMT-LIB provide a clear source of such bounds. The key challenge for theory arbitrage is, therefore, how to impose a bound on an unbounded constraint without modifying its semantics.

3.2 Imposing Bounds

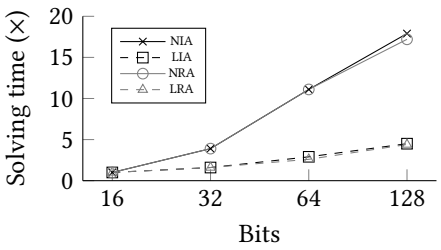
This section briefly discusses theoretical bounds. Unfortunately, we find that satisfying assignments to unbounded constraints are not subject to any practically useful theoretical bounds. We then demonstrate that there is a tradeoff between picking large bounds, which preserve semantics but are slow to solve, and small bounds, which are faster to solve but less likely to be sufficient to represent constraint semantics. Section 4 introduces a bound inference technique based on abstract interpretation to balance this tradeoff and overcome the lack of theoretical guidance.

Integer Arithmetic. The satisfiability of linear integer arithmetic constraints is decidable, and the values of variables in a satisfying assignment are bounded. The theoretical bound on the solutions to a linear integer problem is $2n\,(ma)^{2m+1}$, where n is the number of variables, m is the number of inequalities, and a is the largest constant [59].

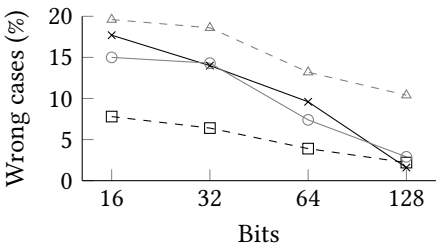
Unfortunately, this theoretical bound on solutions grows exponentially in the number of assertions (inequalities) and, therefore, quickly becomes impractically large, even for relatively small numbers of assertions. Useful SMT problem instances, including those in the SMT-LIB benchmark

Table 1. Summary of theoretical results for unbounded SMT theories.

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



(a) Geometric mean solving time, relative to 16 bits for each logic.



(b) Percentage of constraints with a different satisfiability result from the unbounded original.

Fig. 2. Performance and semantics preservation for naive transformation with a fixed width for nonlinear and linear integer arithmetic (NIA, LIA), and real arithmetic (NRA, LRA). Timeout is 60 seconds.

set, run to thousands of inequalities. Moreover, the bound only applies to *linear* integer arithmetic, as its proof arises from a linear algebraic formulation of the problem. By contrast, solutions to nonlinear integer arithmetic constraints, where multiplication of variables is permitted, have no bounds. Indeed, the question of satisfiability is undecidable in the nonlinear case [24].

Real Arithmetic. Both the linear and nonlinear fragments of real arithmetic are decidable. However, their satisfying assignments have no theoretical bounds [66]. Real arithmetic also presents an additional challenge: solutions may be unbounded not only in magnitude but also in precision. In terms of theory arbitrage, this means that representing a real value with a bounded floating-point number may be incorrect for two distinct reasons: the floating-point value may not have the capacity to represent the magnitude of the real value, or the real number may require (possibly infinitely many) more significant digits than are available in the floating-point context.

Table 1 gives a summary of the theoretical results for the unbounded SMT-LIB theories of integers and real numbers. Only for one logic is there even a theoretical bound on satisfying assignments, and this bound is too large to be of practical use. The lack of theoretical bounds means that it is impossible to represent the semantics of an unbounded constraint in a bounded theory with perfect fidelity—approximation is required.

The Bound Selection Tradeoff. Even though no bound is sufficient for every constraint over integer or real arithmetic in theory, most satisfying assignments and intermediate computations fit within reasonably large bounds in practice. The bigger the bounds selected, the more likely they are to be sufficient to correctly represent an unbounded computation; this phenomenon is quantified in Figure 2b. This militates in favor of always selecting large bounds. However, there is a countervailing factor: larger bounds slow down solving. This effect is shown in Figure 2a and is well-known in the literature as a tactic to speed up both programs and solvers [17, 42]. Thus, any implementation of theory arbitrage must balance the need for bounds large enough to faithfully represent unbounded semantics against the need for bounds small enough for solving to run quickly. In Section 4, we introduce a novel abstract interpretation strategy to achieve this balance.

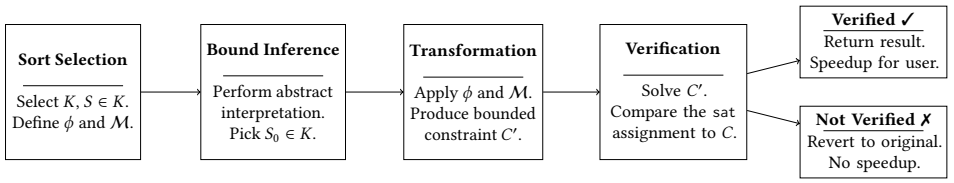


Fig. 3. Overview of the theory arbitrage approach.

4 REALIZING THEORY ARBITRAGE

Our transformation methodology consists of four steps, as summarized in Figure 3. Given a constraint in an unbounded theory, we first select a bounded sort whose domain is a subset of the original unbounded sort and define mappings between functions and constants. Next, we pick bounds. We do this by conducting abstract interpretation to find a likely upper bound on values in the constraint. Then, we translate the constraint to the newly selected theory. Finally, we solve the transformed constraint and verify whether its solution also satisfies the original. If it does, we provide the result to the user; otherwise, we revert to the original constraint.

4.1 Sort Selection

Consider an unbounded SMT theory T; according to Definition 3.4, T has at least one sort S such that $|S|$ is infinite. We select a bounded theory T' with a sort kind K that approximates S, modulo bounds. T' should also contain functions over sorts in K which have equivalent or nearly equivalent semantics to functions over S in T. The notion of sort correspondence is formalized in Definition 4.1.

Definition 4.1 (Sort Correspondence). Let S be an unbounded sort. A *sort correspondence* is a tuple $(S, K, \phi, \mathcal{M})$, where S is a unbounded sort, K is a kind, ϕ is a partial function from S to members of K, and $\mathcal{M}$ is a mapping between functions such that

(i) K is countable and for all $S' \in K$, S' is finite;
(ii) ϕ is a partial surjection for every member of K, *i.e.*, for all $S' \in K$, for all $v \in S'$, $\phi^{-1}(v) \in S$;
(iii) For any pair of sorts $S'_0, S'_1 \in K$, if $S'_0 \subseteq S'_1$, then $\phi^{-1}(S'_0) \subseteq \phi^{-1}(S'_1)$;
(iv) $\mathcal{M}$ is an injective mapping from functions over S to functions over members of K; that is, for all functions $f : S^i \to S^j$, there exists $S' \in K$ such that $\mathcal{M}(f) : S'^i \to S'^j$.

For theories with multiple unbounded sorts, we could in principle generate a sort correspondence for each one. However, in the integer and real cases, there is only one unbounded sort per theory. We expect the mapping $\mathcal{M}$ to produce functions with the same purpose. For instance, in real numbers, we have $\mathcal{M}(+) = \texttt{fp.add}$. However, Definition 4.1 does not require that $\mathcal{M}$ exactly preserve semantics. In fact, we explicitly allow some *semantic differences*, as formalized in Definition 4.2.

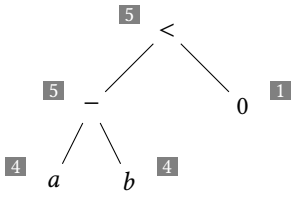
Definition 4.2 (Semantic Difference). Given a sort correspondence $(S, K, \phi, \mathcal{M})$, a *semantic difference* is a pair (f, v) such that

$$\phi\left(f\left(v\right)\right) \neq \mathcal{M}\left(f\right)\left(\phi\left(v\right)\right).$$

Definition 4.2 is required to capture slight differences in the semantics of operations in bounded and unbounded theories; semantic differences occur in small numbers in both the integer-bitvector and real-floating-point sort correspondences because of integer overflow and floating-point rounding. For example, `fp.add` and + have the same effect on most inputs, but `fp.add` may produce slightly different results from regular addition because of floating-point rounding. Semantic differences are not fatal to our transformation strategy, but they do affect the underapproximation approach: the more semantic differences there are, the less likely it is that the translated constraint can be verified. $\mathcal{M}$ should therefore be constructed to minimize the number of semantic differences.

```
1 (declare-fun a () Int)
2 (declare-fun b () Int)
3 (assert (>= a 15))
4 (assert (< (- a b) 0))
5 (check-sat)
```

(a) Simple SMT constraint with satisfying assignment $a = 15$, $b = 16$. The width needed to represent the largest constant 15 is four, which is insufficient to represent the satisfying assignment.



(b) AST of the assertion on Line 4. Boxed numbers represent the width for each value, and are propagated upwards. For variables, we assume the width of the largest constant, four.

Fig. 4. An illustrative example of bound inference on an integer constraint.

In the integer case, let S be the sort of (unbounded) integers and K the kind of bitvectors; the members of K are the sorts corresponding to each bitvector width. We define ϕ to be the function that converts an integer to its binary representation. Bitvectors clearly meet condition (i) of Definition 4.1. In addition, ϕ is a partial surjection by construction, fulfilling property (ii), and for any fixed width sort S_n, the inverse image of S_n under ϕ is a subset of S_{n+1}, fulfilling property (iii). $\mathcal{M}$ maps $*$ to `bvmul`, + to `bvadd`, < to `bvslt`, *etc.* There is one source of semantic differences for integers: overflow in the bitvector theory.

In the real number case, S is the sort of real numbers, and K is the kind of floating-point values in SMT-LIB—as in the integer case, K fulfills property (i) of Definition 4.1. The function ϕ maps real fractional values into their floating-point representation according to IEEE-754. ϕ is partial because there are some real values (irrational values like π or e, for instance) that cannot be mapped to floating-point values. Every floating-point value, though, does have a corresponding real value, fulfilling property (ii).[1] Property (iii) also holds, since a wider floating-point sort can always represent values from a narrower floating-point sort. The definition of $\mathcal{M}$ is intuitive; $\mathcal{M}(-) =$ `fp.sub`, $\mathcal{M}(/) =$ `fp.div`, *etc.* Semantic differences arise from floating-point rounding.

4.2 Bound Inference via Abstract Interpretation

In this subsection, we first provide an example to guide intuition about bound inference, and then formalize our analysis as an abstract interpretation. The concrete domain is the semantics of the constraint, while the abstract domain is the set of widths needed to represent values in the constraint. Our abstract interpretation strategy allows STAUB to select bounds for an input constraint, which balances the tradeoff between large and small bounds.

Example. Figure 4 shows a simple constraint to provide intuition about our bound inference strategy. To translate it to a bounded constraint, we must pick a bit width. The largest constant is 15, which requires four bits; however, the satisfying assignment includes $b = 16$, so choosing a width of four is not sufficient. Instead, we analyze the constraint with an abstract semantics representing the maximum width of values at each syntax tree node, as shown in Figure 4b. We use the largest constant width as an assumption for the variables a and b. At each AST node, we propagate the width up, changing it according to the semantics of each operation. For instance, the result of subtraction may require one more bit than the inputs, so we take the maximum child width (four) and add one to produce a width of five. This process is repeated for <, which simply takes the maximum widths of its children. The result is the width five, which is sufficient to represent the satisfying assignment. We now formalize our analysis as an abstract interpretation.

[1]We define $\phi(0) = +0$ in floating-point semantics, and in practice require that $\phi^{-1}(-0) = 0$. Any computation that produces one of the three pathological floating-point values NaN, $+\infty$, and $-\infty$ is treated as a semantic difference.

Abstract Semantics for Integers. The concrete domain is the power set of integers with no bounds, along with booleans: $\mathbb{C} = \mathcal{P}(\mathbb{Z} \cup \{\texttt{true}, \texttt{false}\})$. The operations on $\mathbb{C}$ are as defined in SMT-LIB, including the standard mathematical functions for addition, subtraction, multiplication, *etc.* As usual, the ordering on $\mathbb{C}$ is the subset relation $\subseteq$. We define the abstract domain to be the set of positive integers $\mathbb{A} = \mathbb{Z}^+$, with the normal arithmetic ordering $\leq$. Intuitively, members of $\mathbb{A}$ represent bit widths; that is, $a \in \mathbb{A}$ represents all integers that can be represented in binary form with a or fewer digits.

The abstraction function $\alpha_i : \mathbb{C} \to \mathbb{A}$ maps a set of integers to the width needed to represent the largest member in binary as in Equation 1. Width is based on the absolute value, with one additional bit to represent negative values in two's complement. Since $\mathbb{C}$ contains booleans, we extend the definition of absolute value by fixing $|c| = 1$ if c is a boolean value.

$$\alpha_i(C) = \left\lceil \left(\log_2 10\right) \cdot \left(\max\left\{|c| : c \in C\right\}\right) \right\rceil + 1 \tag{1}$$

The concretization function $\gamma_i : \mathbb{A} \to \mathbb{C}$ maps a width to the set of integers which can be represented with that width, as shown in Equation 2. The output always includes both boolean values.

$$\gamma_i(a) = \left[-2^{a-1}, 2^{a-1} - 1\right] \cup \{\texttt{true}, \texttt{false}\} \tag{2}$$

Lemma 4.3. *α_i and γ_i form a Galois connection.*

Proof. ($\Rightarrow$) Let $a \in \mathbb{A}$ and $C \in \mathbb{C}$ be arbitrary and suppose that $\alpha_i(C) \leq a$. From the construction of α_i, every single member of C can be represented in $\alpha_i(C)$ or fewer bits or is a boolean. Since $\alpha_i(C) \leq a$, every member of C can be represented by a or fewer bits. Because $\gamma_i(a)$ is the set of values that can be represented in a bits, including `true` and `false`, we have that $C \subseteq \gamma_i(a)$.

($\Leftarrow$) Again let $a \in \mathbb{A}$ and $C \in \mathbb{C}$ be arbitrary. Suppose that $C \subseteq \gamma_i(a)$. Since $\gamma_i(a)$ is the set of all integers that can be represented in a bits or fewer, union booleans, every value in C, including the maximum (in magnitude) value, can be represented in a or fewer bits. Thus $\alpha_i(C) \leq a$. □

Finally, we must define the semantics of each concrete integer function in the abstract domain $\mathbb{A}$. The abstract semantics of each type of expression is given in Figure 5a. Constants are given their actual width, while variables are assigned an unknown variable width x. Expressions that produce a boolean, like `and`, `or`, and `xor`, simply propagate the maximum widths of their arguments. Each integer operation has defined semantics; for example, multiplication can involve adding together the widths of its operands, while addition can only produce a result one bit wider than its inputs. We have constructed the abstract semantics given in Figure 5a so that they preserve order in the abstract domain.

Abstract Semantics for Real Numbers. For real constraints, the concrete domain is the power set of real numbers union with booleans, $\mathbb{C} = \mathcal{P}(\mathbb{R} \cup \{\texttt{true}, \texttt{false}\})$. The operations on $\mathbb{C}$ are those defined in SMT-LIB, as in the integer case. The ordering on $\mathbb{C}$ is the subset relation $\subseteq$. We define the abstract domain to be the set of pairs of positive integers $\mathbb{A} = \mathbb{Z}^+ \times (\mathbb{Z}^+ \cup \{\infty\})$. We define a partial ordering $\preceq$ on $\mathbb{A}$ as follows: for $(m_1, p_1), (m_2, p_2) \in \mathbb{A}$,

$$(m_1, p_1) \preceq (m_2, p_2) \iff m_1 \leq m_2 \wedge (p_2 = \infty \vee p_1 \leq p_2)\,. \tag{3}$$

Equation 3 is not the standard lexicographical order; one pair precedes another only if *both* of its elements are smaller than the elements of the other. Intuitively, members of $\mathbb{A}$ represent pairs of magnitude and precision, and we therefore denote them with (m, p). The first element represents the number of binary digits required to represent the magnitude of a real value, *i.e.*, the number of bits needed to represent the next largest integer. The second element represents the number of binary significant figures required to exactly represent a real value.

$$
\begin{aligned}
[\![n]\!] &= \lceil |n| \, (log_2 10) \rceil + 1 \\
[\![\text{var}]\!] &= x \\
[\![(\texttt{ite}\ b\ E_1\ E_2)]\!] &= \max\left([\![E_1]\!], [\![E_2]\!]\right) \\
[\![(\texttt{abs}\ E_1)]\!], [\![(\texttt{-}\ E_1)]\!] &= [\![E_1]\!] \\
[\![(\texttt{+}\ E_1\ E_2)]\!], [\![(\texttt{-}\ E_1\ E_2)]\!] &= \max\left([\![E_1]\!], [\![E_2]\!]\right) + 1 \\
[\![(\texttt{*}\ E_1\ E_2)]\!] &= [\![E_1]\!] + [\![E_2]\!] \\
[\![(\texttt{/}\ E_1\ E_2)]\!] &= \max\left([\![E_1]\!], [\![E_2]\!]\right) \\
[\![(\texttt{mod}\ E_1\ E_2)]\!] &= [\![E_1]\!] \\
[\![(\texttt{boolop}\ E_1\ E_2)]\!] &= \max\left([\![E_1]\!], [\![E_2]\!]\right)
\end{aligned}
$$

(a) Abstract semantics for integer operations.

$$
\begin{aligned}
[\![n]\!] &= (\lceil |n| \, (log_2 10) \rceil + 1, \text{dig}(n)) \\
[\![\text{var}]\!] &= (x_m, x_p) \\
[\![(\texttt{ite}\ b\ E_1\ E_2)]\!] &= \left(\max\left([\![E_1]\!]_m, [\![E_2]\!]_m\right), \right. \\
&\qquad \left. \max\left([\![E_1]\!]_p, [\![E_2]\!]_p\right)\right) \\
[\![(\texttt{-}\ E_1)]\!] &= [\![E_1]\!] \\
[\![(\texttt{+}\ E_1\ E_2)]\!], [\![(\texttt{-}\ E_1\ E_2)]\!] &= \left(\max\left([\![E_1]\!]_m, [\![E_2]\!]_m\right) + 1, \right. \\
&\qquad \left. \max\left([\![E_1]\!]_p, [\![E_2]\!]_p\right)\right) \\
[\![(\texttt{*}\ E_1\ E_2)]\!] &= \left([\![E_1]\!]_m + [\![E_2]\!]_m, [\![E_1]\!]_p + [\![E_2]\!]_p\right) \\
[\![(\texttt{/}\ E_1\ E_2)]\!] &= \left(\max\left([\![E_1]\!]_m, [\![E_2]\!]_m\right), \infty\right) \\
[\![(\texttt{boolop}\ E_1\ E_2)]\!] &= \left(\max\left([\![E_1]\!]_m, [\![E_2]\!]_m\right), \right. \\
&\qquad \left. \max\left([\![E_1]\!]_p, [\![E_2]\!]_p\right)\right)
\end{aligned}
$$

(b) Abstract semantics for real number operations.

Fig. 5. Abstract semantics for the integer and real number domains. "boolop" represents any of and, or, xor, etc. For real numbers, $[\![E]\!]_m$ denotes the magnitude element of $[\![E]\!]$ while $[\![E]\!]_p$ denotes the precision element.

The real abstraction function $\alpha_r : \mathbb{C} \to \mathbb{A}$ maps a set of real values to a pair representing the greatest magnitude and greatest precision required to represent any value in the set; the formal definition is given in Equation 4, where $\text{dig}(c)$ is the minimum number of binary significant digits needed to exactly represent a real number c; *i.e.*, $\text{dig}(c) = \min\left\{d \in \mathbb{Z}^+ : \left(2^d c\right) \in \mathbb{Z}\right\}$. Note that for irrational numbers, $\text{dig}(c) = \infty$. For boolean c, we set $|c| = 1$ and $\text{dig}(c) = 0$.

$$\alpha_r(C) = \left(\left\lceil \left(\log_2 10\right) \cdot \left(\max\left\{\lceil |c| \rceil : c \in C\right\}\right)\right\rceil + 1, \max\left\{\text{dig}(c) : c \in C\right\}\right) \tag{4}$$

The concretization function $\gamma_r : \mathbb{A} \to \mathbb{C}$ maps a magnitude-precision pair to the set of real values of lower or equal magnitude and no greater precision, as shown in Equation 5.

$$\gamma_i\left((m, p)\right) = \left\{v \in \left[-2^{m-1}, 2^{m-1} - 1\right] : \left(2^p v\right) \in \mathbb{Z} \vee p = \infty\right\} \cup \{\texttt{true}, \texttt{false}\} \tag{5}$$

Lemma 4.4. *α_r and γ_r form a Galois connection.*

Proof. ($\Rightarrow$) Let $(m, p) \in \mathbb{A}$ and $C \in \mathbb{C}$ be arbitrary. Suppose that $\alpha_r(C) \preceq (m, p)$. Then (1) every element of C has a magnitude no greater than 2^m (including booleans); and (2) every element of C requires no more than p bits to be represented exactly (where p may be infinite). For any element $c \in C$, from (1) it holds that $c \in \left[-2^{m-1}, 2^{m-1} - 1\right]$, and from (2) it holds that either $2^p c$ is an integer or $p = \infty$. But these are exactly the conditions given in Equation 5, therefore $C \subseteq \gamma_i\left((m, p)\right)$.

($\Leftarrow$) Let $(m, p) \in \mathbb{A}$ and $C \in \mathbb{C}$ be arbitrary and suppose $C \subseteq \gamma_i((m, p))$. According to Equation 5, every element $c \in C$ is a boolean, or is in the interval $\left[-2^{m-1}, 2^{m-1} - 1\right]$ with either $p = \infty$ or $2^p c \in \mathbb{Z}$. Thus, the magnitude of $\alpha_r(C)$ is less than or equal to m. Moreover, either $p = \infty$ or the element of C requiring the most significant digits can be represented exactly with p digits or fewer. In the first case, by the definition of the ordering $\preceq$, $\alpha_r(C) \preceq (m, p)$ since $p = \infty$. In the second case, we have that the precision portion of $\alpha_r(C)$ is less than or equal to p and so $\alpha_r(C) \preceq (m, p)$. □

Figure 5b gives the abstract semantics of each real number function in SMT-LIB. Constants are given their actual width and precision, while variables are assigned an unknown variable magnitude and precision (x_m, x_p). Because there are no irrational constants in SMT-LIB, constants never result in an infinite precision value. As for integers, the semantics of Figure 5b preserve order in the abstract domain.

Soundness and Implications. The abstract semantics for both integers and real numbers allow us to perform abstract interpretation on constraints over real numbers and integers in SMT-LIB. Since both γ_i, α_i and γ_r, α_r form Galois connections, the abstract interpretation is sound [20]. This leads to the following theorem.

Theorem 4.5 (Soundness of Abstract Interpretation). *Let S be a constraint and $[\![S]\!]$, parameterized by a variable x, be the result of conducting abstract interpretation on S. If S is satisfiable, then all satisfying assignments of S require no more than $[\![S]\!]$ width (or $[\![S]\!]_m, [\![S]\!]_p$ width and precision in the real case) to be represented. Moreover, if evaluating S on a satisfying assignment, the result of every intermediate operation in S has the same property.*

In other words, the soundness of our abstract interpretation implies that any satisfying assignment of S can be found using only the number of bits given by $[\![S]\!]$. If S is `sat`, then we can find its satisfying assignment with computations bounded by the inferred number of bits. The contrapositive means that if no assignment of variables to satisfy S can be found within the bounds given by $[\![S]\!]$, then S is `unsat`. The only drawback is that the result of abstract interpretation is symbolic in the width assumption value x.

Unfortunately, there is no choice of x which is sound for all constraints; indeed, the theoretical unboundedness result implies that no such x exists. Our choice of x, therefore, leads to underapproximation. In practice, we let x be the width of the largest constant present in the constraint, plus one bit. Use of the largest constant is a common strategy in program analysis [52]. For integers, it may be that this x is not large enough; in this case, $[\![S]\!]$ will produce a width that underapproximates the width needed to solve the constraint. Real numbers are subject to the same problem, but there is also a second hurdle: abstract interpretation may produce an infinite precision value. In practice, we must bound $[\![S]\!]_p$ to some reasonable value, and in our implementation of STAUB we achieve this by modifying the abstract semantics of the division operation so that $[\![(/\ E_1\ E_2)]\!] = \left([\![E_1]\!]_m + [\![E_2]\!]_m, [\![E_1]\!]_p + [\![E_2]\!]_p\right)$. This assumption results in further underapproximation in practice.

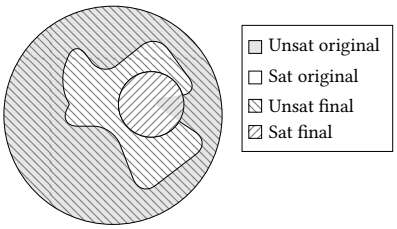
Implementation. We implement our analysis as a custom abstract interpretation in order to integrate it with constraint parsing, thus maximizing efficiency. We use a simple syntax tree traversal because this aligns with the properties of SMT-LIB. In particular, SMT-LIB is declarative, with simple syntax trees consisting of variables, constants, and pure function applications. Since there are no variable assignments, the abstract domains for both integers and real numbers are inherently relational, and because each branch of the syntax tree is independent, the analysis is modular as well. This gives rise to a simple analysis that traverses a constraint's syntax tree, applying the abstraction function to values and the abstract semantics given in Figure 5 to each function application. Runtime is critical for theory arbitrage since the abstract interpretation must be performed live for each constraint, rather than being run once statically for a program whose performance it does not affect as in more traditional practical applications. In Section 6.3, we describe why the performance of abstract interpretation is critical in STAUB's implementation and discuss the possibility of alternative abstract interpretation strategies.

4.3 Constraint Transformation

With a theory correspondence and bounds, we can convert a constraint over an unbounded theory into a bounded one. For each variable, we simply map its sort to the sort $S' \in K$ identified via abstract interpretation. For instance, if $[\![S]\!] = 12$ for an integer constraint, we translate it to the 12-bit bitvector type. We convert each constant to the equivalent value in the new sort using the definitions of ϕ for integers and real numbers, a straightforward process.

Bounded Result	sat		unsat
Assignment matches?	Yes	No	
Conclusion	sat	None	None
Effect	Speedup	Revert	Revert

(a) Table of possible cases.



(b) Visualization of possible assignments and each case.

Fig. 6. Summary of possible cases after STAUB application. The diagram shows the space of all possible solutions, with a subset satisfying the original constraint in white. The inner circle represents assignments that satisfy the bounded constraint after STAUB is applied.

We next turn to the mapping $\mathcal{M}$. There are six mathematical operations on integers: absolute value, negation, addition, subtraction, multiplication, and division. Each has a corresponding bitvector operation: `abs`, `bvneg`, `bvadd`, `bvsub`, `bvmul`, and `bvsdiv`. There are also six comparison operations: $=$, $\neq$, $\leq$, $<$, $\geq$, and $>$. Equality and inequality are identical for integers and bitvectors, and the four comparators are mapped to `bvsle`, `bvslt`, `bvsge`, `bvsgt`. In the real number case, there are five functions—the same as for integers, but without absolute value. These are mapped to `fp.neg`, `fp.add`, `fp.sub`, `fp.mul`, and `fp.div`. The comparison functions are analogous to the integer case.

Each operation in both the integer and floating-point cases has some semantic differences; that is, some cases where the result of the function application differs between the bounded and unbounded theories. In the integer case, this occurs because of overflow semantics in the theory of bitvectors. In the floating-point case, it occurs because of IEEE-754 floating-point rounding. For integers, we add constraints to prohibit overflow by invoking SMT-LIB's `bvaddo`, `bvsubo`, *etc.* predicates.[2] For real numbers, there is no way to prevent existing solvers from using floating-point rounding semantics.

4.4 Verification

Because the choice of x creates an underapproximation, the constraint C' produced by the translation step may not be equisatisfiable with the original constraint C. To prevent uncertainty about correctness, we add a verification step to ensure that the bounds produced by abstract interpretation are sufficient for a particular constraint. We conduct this verification according to the portfolio approach to solving [68]: we run STAUB and a solver in parallel. If the solver finishes first, we take its result and do not change performance. If STAUB finishes first, on the other hand, there are three possible cases, shown in Figure 6. In particular, if STAUB produced a `sat` result, we *verify* that the satisfying assignment also satisfies the original unbounded constraint.

(1) **The transformed constraint is** `unsat`**.** The bounded constraint may be `unsat` because the original constraint was or because the selected bounds were insufficient. The two cases cannot be distinguished, so we default to the original constraint and provide no speedup.
(2) **The transformed constraint is** `sat`**, and its satisfying assignment also satisfies the original constraint.** In this case, we return the satisfying assignment to the user, providing a substantial speedup.
(3) **The transformed constraint is** `sat`**, but its satisfying assignment does not satisfy the original constraint.** This circumstance arises as the result of semantic differences like floating-point rounding; the solver may have found an assignment that relies on a semantic difference to produce a `sat` result. We default to the original constraint.

[2]The overflow predicates are proposals for the next version of the SMT-LIB standard [30]. However, both Z3 and CVC5 already implement them.

The cumulative effect of our underapproximation approach is to dramatically speed up a subset of constraints while providing no benefit on other constraints. In practice, our theory arbitrage achieves large speedups on a large proportion of satisfiable constraints while providing no improvement on unsatisfiable constraints. In the next section, we evaluate the benefits of the approach, including the precision of our abstract interpretation bound inference strategy.

5 EVALUATION

We evaluate our theory arbitrage approach extensively on standard SMT benchmarks, in combination with the bitvector and floating-point constraint optimizer SLOT [50], and with a practical program analysis tool, the Ultimate Automizer, that uses SMT solving to prove termination [36]. We summarize our findings as follows.

- Theory arbitrage speeds up average solving by up to 1.4×, and makes hundreds of unknown constraints tractable. Our abstract interpretation approach outperforms a fixed choice of width. The improvement is particularly strong for nonlinear integer constraints; some linear integer and real constraints are sped up, but mean speedups are lower for these logics.
- Beyond its own speedup, STAUB unlocks further optimizations available for bounded theories, achieving an *additional* 2×-3× speedup.
- Our approach to improving solver performance can benefit a client application even under pessimistic circumstances; STAUB speeds up constraint solving for the Ultimate Automizer by about 9%.

5.1 Experimental Setup

Given an SMT constraint C over an unbounded theory, suppose it takes a solver T_{pre} time to solve the constraint. A solver user experiences an improvement if we can provide a correct solution to C in less than T_{pre} time. STAUB is not a solver, but a pre-processor compatible with any SMT-LIB–compliant solver. Thus, we must consider first the time it takes to transform the unbounded constraint into a bounded one with STAUB (T_{trans}), the time it takes a solver to solve the bounded constraint (T_{post}), and the time it takes to verify the bounded solution (T_{check}).

A performance improvement occurs in one of two ways. First, if a solver produced an unknown or timeout result on a constraint and STAUB succeeds in producing a satisfying assignment, this gives a user a result for a previously unknown constraint; we call such a case a *tractability improvement*. Second, a user experiences a performance improvement if the entire final running time is less than the initial: $T_{pre} > T_{trans} + T_{post} + T_{check}$. We measure the magnitude of the speedup as a ratio, *i.e.*, $\alpha = \frac{T_{pre}}{T_{trans}+T_{post}+T_{check}}$ and focus on α on average over many constraints following portfolio methodology [68]. The use of portfolio methodology is a key part of STAUB's structure because it is essential to handling `unsat` constraints that are not known to be unsatisfiable initially. While a portfolio approach ensures that no cases are slowed down, it comes with the drawback that two cores are required. Some constraints are sped up by STAUB while others are fastest in original form, for instance, taking advantage of numeric algorithms for real numbers.

With measurement definitions in mind, we ask the following three research questions.

- **RQ1: Does theory arbitrage improve solver performance?** Specifically, how many tractability improvements occur, and what is the mean value of α over benchmarks?
- **RQ2: Does STAUB unlock speedups for bounded constraints?** Specifically, does STAUB enable the use of an existing optimization for bounded constraints to increase α further?
- **RQ3: How much does STAUB improve a client analysis?** Specifically, does the performance improvement provided by STAUB translate to better performance of a state-of-the-art termination proving tool?

Table 2. Count of tractability improvements for each logic and solver, including comparison of STAUB to fixed width choices. The last column represents cases that could not be solved by *either* solver originally but could be solved by *at least one* solver after theory arbitrage.

	Z3			CVC5			Z3 ∩ CVC5		
	8-bit	16-bit	STAUB	8-bit	16-bit	STAUB	8-bit	16-bit	STAUB
NIA	184	63	305	2,594	1,464	3,241	148	47	278
LIA	27	9	67	63	26	110	24	8	63
NRA	0	3	1	4	13	6	0	1	0
LRA	0	0	0	0	0	0	0	0	0

Implementation. We have implemented our approach in C++ as an end-to-end tool enabling parsing of input constraints, abstract interpretation, output of transformed constraints, and underapproximation checking. We use Z3's parser for the SMT-LIB language and embed solving and underapproximation checking with Z3 in STAUB. We also provide a terminal flag to allow the output of SMT-LIB bounded constraints for use with other solvers.

Benchmarks. We use as test cases the standard solver benchmarks provided with SMT-LIB for the unbounded theories [4]. There are four sets of benchmarks: linear integer arithmetic (`QF_LIA`, 13,224 constraints), nonlinear integer arithmetic (`QF_NIA`, 25,358 constraints), linear real arithmetic (`QF_LRA`, 1,753 constraints), and nonlinear real arithmetic (`QF_NRA`, 12,134 constraints). We evaluate using the quantifier-free versions of each logic and leave the extension of STAUB to quantified logics to future work. For our evaluation with the Ultimate Automizer [36], we use 97 C programs taken from the SV-COMP benchmark set for termination proving [9].

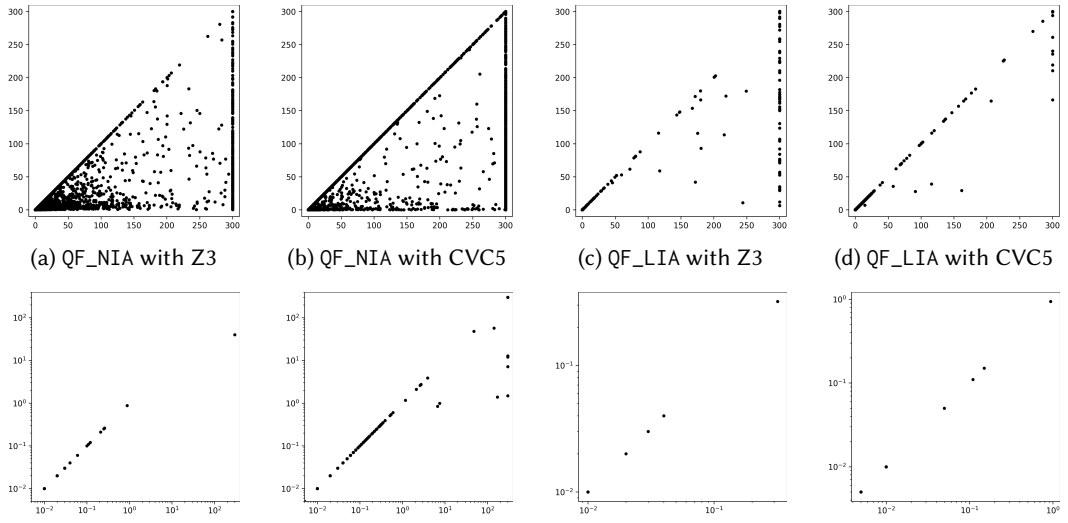
Solvers. We select Z3 and CVC5 [3] for testing because they are the two state-of-the-art general SMT solvers most widely used in existing work [16, 67]. For RQ2, we use SLOT [50], an open-source tool that speeds up the solving of bitvector and floating-point constraints by applying compiler optimizations. For RQ3, we evaluate using the open-source Ultimate Automizer, a state-of-the-art tool for proving termination proving [36]. For all solvers and tools, we test with the latest available development versions as of October 2023.

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 up to 300 seconds, in line with those used in prior work [16, 49, 50]. When measuring speedups, we report the geometric mean, and count solver and STAUB timeouts as 300-second contributions.

5.2 RQ1: Does theory arbitrage improve solver performance?

To answer the performance question, we investigate tractability improvements and proportional speedups. We also compare STAUB's abstract interpretation-based width inference to a static choice of fixed widths.

Tractability Improvements. Table 2 shows the number of tractability improvement cases for each solver and each logic at 300 seconds; the "STAUB" columns contain the overall result. Theory arbitrage performs by far the best for nonlinear integers, allowing CVC5 to solve thousands of constraints for which it originally timed out. Z3 also sees hundreds of traceability improvements. We achieve more modest tractability improvements for linear integer and nonlinear real arithmetic. There are no tractability improvements for linear real arithmetic; this is an effect of a large number of semantic difference cases and the small size of the benchmark set, which contains only 251 constraints that originally time out.



(a) QF_NIA with Z3 (b) QF_NIA with CVC5 (c) QF_LIA with Z3 (d) QF_LIA with CVC5

(e) QF_NRA with Z3 (f) QF_NRA with CVC5 (g) QF_LRA with Z3 (h) QF_LRA with CVC5

Fig. 7. Plots of final versus initial solving time for each solver on each benchmark set. The x-axis is the original solving time, and the y-axis is the solving time after STAUB is applied. Points below the diagonal represent reductions in solving time, while values along the line $x = 300$ represent tractability improvements. Figures 7e-7h are plotted with a logarithmic scale since most constraints have very small initial solving time. All measurements include T_{trans} and T_{check}.

Proportional Speedups. Figure 7 gives plots of the performance of benchmark constraints before and after theory arbitrage is applied. Points along the diagonal represent no change in performance, points below the diagonal represent a performance improvement, and points along the right edge of the graph represent tractability improvements. Our use of portfolio methodology [68] means that no cases experience worse performance (above the diagonal). STAUB is again most effective on nonlinear integer arithmetic, with cases roughly randomly distributed at various levels of improvement. STAUB produces more traceability improvements under CVC5, but also more cases with no improvement compared to Z3. A similar pattern, though with fewer cases, emerges for linear integers. Nonlinear real arithmetic is substantially sped up for a few dozen constraints under CVC5, but in general, most of the verifiable constraints in real arithmetic have short initial solving times.

Table 3 shows the results in numerical form, giving the geometric mean speedups achieved by STAUB in the column labeled "STAUB Verified Speedup". For all logics and solvers but QF_LRA, STAUB achieves large speedups on verified cases; as much as 5.5× for QF_NIA and up to 14× for a small number of nonlinear real constraints. The "Overall Speedup" column gives the mean speedup over the entire benchmark set—both verified cases, which are sped up substantially, and unsat and semantic difference cases which experience no improvement. For instance, in the first row, a 1.82× speedup on 8,121 of 25,358 cases translates to a 1.21× speedup overall. We break out constraints by intervals of initial solving time because the cost of running STAUB is large relative to T_{pre} for small constraints, affecting proportional but not absolute speedups. The results show that STAUB is most effective for QF_NIA, but still speeds up linear integer and nonlinear real constraints. In addition, Table 3 demonstrates that performance improvements are not solver-specific, as improvements occur for both Z3 and CVC5.

Table 3. Geometric mean speedups for each logic for various original solving time intervals. Each "Verified Cases" column gives the number of constraints where the satisfying assignment after theory arbitrage could be verified. "Verified Speedup" is the geometric mean speedup for verified cases, and "Overall Speedup" is the geometric mean speedup for the entire benchmark set, including cases that could not be verified. The "SLOT" column gives the overall speedup after application of SLOT (Section 5.3). All measurements include T_{trans} and T_{check}.

Logic	Solver	T_pre	Count	Fixed 8-bit			Fixed 16-bit			STAUB			SLOT
				Verified Cases	Verified Speedup	Overall Speedup	Verified Cases	Verified Speedup	Overall Speedup	Verified Cases	Verified Speedup	Overall Speedup	Overall Speedup
NIA	Z3	0-300	25,358	7,434	1.204	1.056	5,094	1.53	1.089	8,121	1.820	1.211	1.480
		1-300	18,632	4,870	1.302	1.071	2,760	2.145	1.12	5,565	2.215	1.268	1.659
		60-300	7,963	490	3.232	1.075	204	15.114	1.072	798	4.857	1.172	1.461
		180-300	6,654	233	4.393	1.053	79	10.113	1.028	393	5.482	1.106	1.252
	CVC5	0-300	25,358	7,434	3.248	1.412	5,094	3.264	1.268	8,121	1.998	1.248	2.763
		1-300	21,607	5,787	4.483	1.495	3,383	5.900	1.320	6,320	2.370	1.287	3.225
		60-300	14,224	3,130	11.194	1.702	1,755	21.383	1.459	3,809	3.386	1.386	4.263
		180-300	12,849	2,769	12.249	1.716	1,558	24.542	1.474	3,423	3.498	1.396	4.387
LIA	Z3	0-300	13,224	2,208	1.011	1.002	3,648	1.003	1.001	4,400	1.014	1.005	1.008
		1-300	4,273	95	1.276	1.005	64	1.174	1.002	171	1.426	1.014	1.024
		60-300	2,147	46	1.653	1.011	14	20.83	1.005	91	1.942	1.029	1.047
		180-300	1,408	34	1.965	1.016	9	3.131	1.007	75	2.157	1.042	1.067
	CVC5	0-300	13,224	2,208	1.246	1.037	3,648	1.100	1.027	4,400	1.002	1.001	1.013
		1-300	6,608	282	2.509	1.040	531	1.320	1.023	818	1.009	1.001	1.026
		60-300	3,969	90	2.601	1.022	47	3.569	1.015	140	1.044	1.002	1.027
		180-300	3,650	67	3.121	1.021	30	5.217	1.014	116	1.018	1.001	1.025
NRA	Z3	0-300	12,133	613	1.002	1.000	994	1.005	1.000	988	1.002	1.000	1.000
		1-300	2,001	1	4.047	1.001	6	2.193	1.002	1	7.517	1.001	1.000
		60-300	1,233	1	4.047	1.001	5	2.566	1.004	1	7.517	1.002	1.000
		180-300	1,190	0	-	-	3	3.569	1.003	1	7.517	1.002	1.000
	CVC5	0-300	12,134	613	1.031	1.002	994	1.036	1.003	988	1.026	1.002	1.001
		1-300	2,189	14	3.841	1.009	28	3.473	1.016	16	4.847	1.012	1.007
		60-300	1,138	4	33.738	1.012	13	14.068	1.031	8	14.075	1.019	1.013
		180-300	1,006	4	33.738	1.014	13	14.068	1.035	6	13.085	1.015	1.011
LRA	Z3	0-300	1,753	35	1.000	1.000	106	1.000	1.000	54	1.000	1.000	1.000
		1-300	700	0	-	-	3	1.000	1.000	0	-	-	-
		60-300	262	0	-	-	0	-	-	0	-	-	-
		180-300	172	0	-	-	0	-	-	0	-	-	-
	CVC5	0-300	1,753	35	1.000	1.000	106	1.000	1.000	54	1.000	1.000	1.000
		1-300	832	0	-	-	10	1.000	1.000	0	-	-	-
		60-300	446	0	-	-	0	-	-	0	-	-	-
		180-300	276	0	-	-	0	-	-	0	-	-	-

Effectiveness of Width Inference. We also conduct an ablation study to evaluate the effectiveness of our abstract interpretation strategy. The average width computed by STAUB's abstract interpretation is 13.1 bits, so we compare STAUB to both a smaller standard fixed bitwidth (8 bits) and a larger standard width (16 bits). The columns labeled "Fixed 8-bit" and "Fixed 16-bit" in Table 3 give the results. The data show that, especially for Z3, our abstract interpretation method substantially outperforms both constraint-independent fixed choices of width. For instance, for nonlinear integer arithmetic under Z3, STAUB achieves a speedup of 1.211× versus just 1.056× for a fixed choice of 8 bits and 1.089× for 16 bits. For CVC5, there is no substantial difference, and STAUB performs about the same or slightly worse than the fixed bit width choices. The fact that there are more verified cases with 8 bits than with 16 bits is at first surprising, as it differs from the tradeoff described in Figure 2. This occurs because the higher running time of 16-bit constraints means that more of them time out. Constraints which time out cannot be verified by STAUB, decreasing the number of verified cases relative to the 8 bit width. Table 2 confirms this and shows that our width inference strategy provides the highest number of tractability improvements, followed by a fixed choice of 8 bits and finally fixed 16 bits. Overall, the results show that width inference via abstract interpretation with STAUB is far more effective than either a larger or smaller fixed width choice under Z3, and performs no worse under CVC5.

Table 4. Results for applying STAUB to the Ultimate Automizer client analysis.

Benchmarks	97
Verified cases	8
Tractability improvements	1
Mean speedup for verified cases	2.93×
Overall mean speedup	1.093×

5.3 RQ2: Does STAUB unlock speedups for bounded constraints?

Theory arbitrage not only speeds up solving, but also allows existing optimization strategies for bounded constraints to be applied to unbounded ones. We investigate this advantage of STAUB by combining it with SLOT, a recent tool that simplifies bitvector and floating-point constraints using compiler optimization [50]. Without theory arbitrage, SLOT could not be applied to unbounded constraints. The "SLOT" columns in Table 3 show the mean speedups achieved by chaining together theory arbitrage and SLOT. In most cases, SLOT speeds up constraints by an additional factor of 2× or 3×, and for a few nonlinear integer constraints, achieves orders of magnitude speedups. We also test the STAUB-SLOT combination for real arithmetic; however, since SLOT only supports the standard 16-, 32-, 64-, and 128-bit floating-point values, we must "round up" the widths STAUB selects to apply SLOT. The higher widths negate any benefit of SLOT's optimization.

5.4 RQ3: How much does STAUB improve a client analysis?

Finally, we investigate to what extent STAUB can improve a practical client analysis tool. We choose the Ultimate Automizer, a termination proving tool that has recently competed in the program analysis competition SV-COMP [36]. The tool generates a constraint that encodes the termination problem for a program under study and calls an existing solver to obtain a solution or `unsat` result. The Ultimate Automizer is a *pessimistic* use case for STAUB because the majority of the constraints it generates are `unsat`, and therefore cannot be verified. Nevertheless, we find that STAUB provides a speedup on average. We run the constraints produced by the Ultimate Automizer using STAUB and compare the results to Z3. There are 931 C programs in the SV-COMP benchmark set related to termination proving [9]. However, the Ultimate Automizer generates constraints with arrays, which STAUB does not support, for most of these constraints. We, therefore, limit our evaluation to the 97 programs for which the Ultimate Automizer does not produce array constraints. On average, STAUB can speed up performance by 9%, as summarized in Table 4. Because it is a pessimistic use case, the performance improvement in contexts with a greater proportion of satisfiable constraints, like those included in the SMT-LIB benchmark set, is likely to be greater.

6 DISCUSSION

6.1 Bound Inference Overhead

The results presented in Section 5 show that STAUB can practically speed up constraint solving for unbounded constraints, and every result presented includes the running time of STAUB, *i.e.*, T_{trans} and T_{check}. In our evaluation, we find that for the vast majority of large constraints, $T_{trans} \ll T_{post}$ and that T_{check} is *de minimis*. Bound inference has linear runtime in the size of the constraint's AST; so does translation. Solving, even after transformation to a bounded theory, on the other hand, has highly unpredictable, and, in the worst case, exponential performance. Since solvers are highly optimized, T_{trans} may reach or exceed T_{post} for small constraints, which drags down proportional speedups. As T_{pre} and consequently T_{post} grow, though, the impact of translation on performance decreases. This phenomenon is why proportional speedups with STAUB (and SLOT) in Table 3 are higher for larger values of T_{pre}.

6.2 Bound Selection and Refinement

The abstract interpretation approach described in Section 4 provides a general method for inferring bounds and yields substantial performance improvements. Future work may explore improvements to the bound selection process. One possible improvement is to engage in iterative bound refinement, where bounds are selected, tested, and then either decreased or increased repeatedly. Such an approach may be able to produce better bounds; however, checking whether the bounds are too large or too small likely requires solving a constraint, which has unpredictable running time and may exceed the cost of solving the original constraint.

Bounds could also be refined by selecting widths on a per-variable basis; this could be effected, for instance, by using multiple variable assumptions $x_0, x_1, \ldots$. While this may provide performance benefits, it would also introduce width tracking overhead to comply with SMT-LIB semantics. For instance, addition of bitvectors requires that both arguments be of the same width, so performing translation correctly in the context of multiple widths would require bit extension or truncation of one of the parameters. The addition of an extension or truncation operation may make the constraint harder to solve because it would result in expensive mixed bitwise and arithmetic operations [69]. Moreover, the analysis required to determine which variables to extend or truncate would slow down the bound inference process.

Finally, it may be possible to improve bound selection by introducing domain knowledge. The theoretical unboundedness results [24, 66] mean that this is not possible for all SMT constraints, but it may be possible for fragments of an unbounded logic. STAUB already supports user specification of a fixed width, but it could be extended to allow users to specify a custom formula relating width to the constraint contents. For instance, in a constraint representing a scheduling problem [13], a user may know a bound on how precisely time steps are measured. Such an approach would increase the burden on users, but could increase the number of cases where the result can be verified. Verification might still be required, though, as a result of semantic differences.

6.3 Abstract Interpretation Strategies

The simplicity of the abstract interpretation strategy described in Section 4.2 arises from SMT-LIB's simplicity relative to conventional programming languages. Future work, though, could use a different abstract interpretation strategy to achieve further improvements. For example, iterative bound refinement would introduce a loop and may allow STAUB to use new abstract domains and include widening operators as in the seminal work on abstract interpretation [20]. Future work could also use an iterative forward-backward abstract interpretation strategy [27, 71] to perform bound refinement, where the forwards analysis computes bounds, and the backwards analysis computes an assumption value like x.

A set of multiple ordered abstract domains with a suitable ordering can also provide performance benefits [1]; in the theory arbitrage context, this may better cover the precision portion of real number constraints. More complex logical connectives, which have been used to analyze the relationship between variables in other contexts [39], could also reduce the need for the assumption value x, though it couldn't eliminate assumptions entirely. The cost of more complex abstract interpretation strategies is worse performance, and in theory arbitrage, bound inference must be carried out for every constraint and its runtime offset against the improvement in solving time. This differs from traditional uses of abstract interpretation for imperative programs, in which the analysis must be run only one time and does not affect the performance of the program being analyzed [21]. Performance concerns are therefore greater for STAUB than for traditional programs, and the benefits of a more sophisticated abstract interpretation strategy would have to be weighed against its performance costs.

6.4 Extension to Other Theories

STAUB's theory arbitrage approach handles the two common unbounded theories of integers and real numbers. However, apart from the implementation details of the abstraction and concretization functions, our approach is general and could be applied to any given pair of unbounded and bounded theories. SMT-LIB does not contain any other such pairs, but it does include two additional theories which meet Definition 3.4: arrays and strings [4]. For both theories, length is an unbounded integer. An abstract interpretation approach analogous to that presented in Section 4 could be used to infer bounds on the length of arrays or strings; for instance, concatenation can result in the sum of the lengths of its arguments, while a substring operation reduces or does not change string length. We leave to future work the question of whether a theory of fixed-size arrays could improve solver performance, merely noting that if the answer is in the affirmative, then theory arbitrage could be applied to arrays or strings as well. Future work could also explore the application of our bound inference strategy to constraints that are themselves bounded, for instance, to reduce the width of bitvector constraints. Existing work suggests that even drastic reductions in the width of bitvectors may improve performance with only a small accuracy cost [42], which may render the abstract interpretation approach less useful for already–bounded constraints. Finally, it may be possible to extend a STAUB-like strategy to existential and universal quantifiers. The major challenge with applications to quantified theories is that there is no clear abstract semantics for quantifiers.

7 RELATED WORK

7.1 Speeding up SMT Solving

Existing work on improving solver performance has focused on strategies implemented within particular solvers. Such work sometimes takes the form of new solvers, including the seminal work on Z3 [25], and newer solvers like Bitwuzla [55], MathSAT [18], and Yices [28]. In addition, recent approaches have improved algorithms for existing solvers, for instance, improving performance on string constraints [7] or regular expressions [8]. Trident speeds up solving on bitvector constraints by propagating bitvector information into the SAT solving backend of an SMT solver [70].

Other theory-agnostic innovations have included the introduction of syntax-guided quantifier instantiation [56] and the combination of existing solvers with fuzzing [53]. Recently, attention has turned to using machine learning to speed up solving. FastSMT [2] uses a neural network to select heuristics within a solver, while MachSMT [64] uses machine learning to choose among different solvers Our approach contributes to the literature by offering a solver-agnostic approach to pre-processing constraints which transforms constraints from one theory into another, rather than developing new, or more systematically choosing among existing, solver heuristics.

7.2 Constraint Pre-Processing

While most prior work has focused on solver-internal improvements, some novel approaches match STAUB in pre-processing constraints before feeding them to a solver. MBA-Solver [69] takes as input bitvector constraints and simplifies them with re-writing rules focused on mixed bitwise and arithmetic operations. SLOT [50] pre-processes bitvector and floating-point constraints by applying compiler optimizations. Bjørner and Fazekas introduce new methods to detect when pre-processing steps can be applied to newly-added constraints during incremental solving [11]. Nötzli *et al.* introduce a syntax-guided synthesis approach to developing new pre-processing rules, and implement the approach within CVC4 [57]. While STAUB shares the pre-processing approach with these works, it differs in that it transforms from one theory to another, rather than within one theory. Moreover, we introduce an underapproximate-then-check strategy which differs from existing works that rely on exact equivalence for all constraints.

7.3 Inference and Imposition of Bounds

Prior work has explored the problem of deducing bounds for constraints and programs. FPTuner [17] analyzes floating-point computation in programs and uses Taylor expansions to deduce "bounds" (limits on precision) on floating-point variables, thereby reducing bitwidths and improving performance. Jonáš and Strejček introduce a width reduction approach for bitvectors without changing from one theory to another [42]. Bitblasting is an approach that converts floating-point constraints to bitvector form, simplifying reasoning. Bitblasting is implemented in existing solvers [3, 25], and other work has developed new bitblasting transformations for use within solvers [14]. Molly [60] expands beyond bit-blasting to recover satisfying assignments from approximate models of floating-point arithmetic in any approximating theory. Int-blasting [74] is a technique for modeling bitvector computations using the theory of integers. Our approach goes in the other direction, using bitvectors to model unbounded integer computation; moreover, we do not introduce any theory extensions and simply translate them to a bounded theory. Both approaches can produce speedups in part because portfolio methodology allows STAUB to take the best of optimizations for both bounded and unbounded theories.

Our bound inference approach is also related to existing work on analyzing numerical approximations. Fluctuat and its family of tools use abstract interpretation to bound the possible error of machine arithmetic which models real-world phenomena with floating-point values [32–35]. Coward *et al.* integrate e-graphs with abstract interpretation for the same purpose: finding bounds on the precision of floating-point computation [22]. The Rosa compiler converts real computation to floating-point values for machine arithmetic [23], but it requires users to encode bounds in the input language. In the context of SMT constraints, no such information is available, creating the need for STAUB's underapproximation and verification steps.

7.4 Approximation for SMT Solving

Existing work has also explored the use of approximation to improve solver performance. UppSAT [72] introduces an underapproximation and overapproximation strategy for constraints in the theory of floating-point numbers. Zeljić *et al.* also introduce an approximation framework for simplifying constraints over any theory without the requirement that approximations be either overapproximations or underapproximations [73]. Binary decision diagram (BDD)-based solvers have also been extended with approximations that model just a few bits of a bitvector constraint [41]. Alternating over- and under-approximations are also useful for bitvector solving [63]. Our approach follows the underapproximation strategy; however, we use underapproximation in order to obtain the benefits of solving over a bounded theory, instead of using it to directly simplify a constraint within the same theory.

8 CONCLUSION

This work has presented SMT theory arbitrage, a novel method to speed up SMT solving by transforming constraints from unbounded theories, which are expensive to solve, to bounded theories, which are cheaper to solve. To effect the translation, we introduce an abstract interpretation-based width inference strategy which strikes a balance between between large bounds which are more likely to be correct but slower to reason about, and small bounds, which are the opposite. We implement our approach as STAUB, a practical tool to speed up solving. Theory arbitrage speeds up solving by orders of magnitude on individual constraints, and up to 1.4× on average. STAUB also unlocks an additional 2×—3× speedup by allowing the application of existing optimization techniques for bounded theories. The speedup is portable across solvers, and can improve the performance on a client analysis' constraints by 9%.

DATA AVAILABILITY

The data and implementation referenced in this paper have been persistently archived [51]. The latest version of STAUB is also made publicly available on Github.[3]

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback on earlier drafts of this paper. The work described in this paper was supported, in part, by the United States National Science Foundation (NSF) under grants 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] Vincenzo Arceri, Martina Olliaro, Agostino Cortesi, and Pietro Ferrara. 2022. Relational String Abstract Domains. In *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13182)*. Springer, 20–42. https://doi.org/10.1007/978-3-030-94583-1_2

[2] Mislav Balunović, Pavol Bielik, and Martin Vechev. 2018. Learning to solve SMT formulas. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 10338–10349.

[3] 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

[4] 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`.

[5] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*. Springer International Publishing, 305–343. https://doi.org/10.1007/978-3-319-10575-8

[6] Murphy Berzish, Joel D. Day, Vijay Ganesh, Mitja Kulczynski, Florin Manea, Federico Mora, and Dirk Nowotka. 2023. Towards more efficient methods for solving regular-expression heavy string constraints. *Theoretical Computer Science* 943 (2023), 50–72. https://doi.org/10.1016/j.tcs.2022.12.009

[7] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: a string solver with theory-aware heuristics. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design* (Vienna, Austria) *(FMCAD '17)*. FMCAD Inc, Austin, Texas, 55–59.

[8] 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*. Springer International Publishing, Cham, 289–312. https://doi.org/10.1007/978-3-030-81688-9_14

[9] Dirk Beyer. 2022. Progress on Software Verification: SV-COMP 2022. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, Cham, 375–402. https://doi.org/10.1007/978-3-030-99527-0_20

[10] Dirk Beyer, Matthias Dangl, and Philipp Wendler. 2018. A Unifying View on SMT-Based Software Verification. *Journal of Automated Reasoning* 60, 3 (2018), 299–335. https://doi.org/10.1007/s10817-017-9432-6

[11] Nikolaj Bjørner and Katalin Fazekas. 2023. On Incremental Pre-processing for SMT. In *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14132)*. Springer, 41–60. https://doi.org/10.1007/978-3-031-38499-8_3

[12] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 643–659.

[13] Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. 2020. SMT encodings for Resource-Constrained Project Scheduling Problems. *Computers & Industrial Engineering* 149 (2020), 106777. https://doi.org/10.1016/j.cie.2020.106777

[14] Martin Brain, Florian Schanda, and Youcheng Sun. 2019. Building Better Bit-Blasting for Floating-Point Problems. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as*

[3] https://github.com/mikekben/STAUB.

Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11427). Springer, 79–98. https://doi.org/10.1007/978-3-030-17462-0_5

[15] Richard P. Brent and Paul Zimmermann. 2010. *Modern Computer Arithmetic*. Cambridge University Press. https://doi.org/10.1017/CBO9780511921698

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[17] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-Point Mixed-Precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 300–315. https://doi.org/10.1145/3009837.3009846

[18] 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

[19] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. 2012. A quantifier-free SMT encoding of non-linear hybrid automata. In *Formal Methods in Computer-Aided Design, FMCAD 2012*. IEEE, 187–195.

[20] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/567752.567778

[21] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4435)*. Springer, 272–300. https://doi.org/10.1007/978-3-540-77505-8_23

[22] Samuel Coward, George A. Constantinides, and Theo Drane. 2023. Combining E-Graphs with Abstract Interpretation. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Orlando, FL, USA) *(SOAP 2023)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3589250.3596144

[23] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/2535838.2535874

[24] Martin Davis, Yuri Matijasevič, and Julia Robinson. 1976. Hilbert's tenth problem: Diophantine equations: positive aspects of a negative solution. In *Mathematical developments arising from Hilbert problems (Proceedings of Symposia in Pure Mathematics, Vol. XXVIII)*. American Mathematical Society, Providence, RI, 323–378. (loose erratum).

[25] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[26] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. 2011. Exploiting Symmetry in SMT Problems. In *Automated Deduction – CADE-23*. Springer, Berlin, Heidelberg, 222–236. https://doi.org/10.1007/978-3-642-22438-6_18

[27] Aleksandar S. Dimovski. 2023. Error Invariants for Fault Localization via Abstract Interpretation. In *Static Analysis: 30th International Symposium, SAS 2023, Cascais, Portugal, October 22–24, 2023, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 190–211. https://doi.org/10.1007/978-3-031-44245-2_10

[28] 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

[29] Bruno Dutertre and Leonardo de Moura. 2006. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification* (Seattle, WA) *(CAV'06)*. Springer-Verlag, Berlin, Heidelberg, 81–94. https://doi.org/10.1007/11817963_11

[30] Pascal Fontaine. 2022. SMT-LIB Google Group: FixedSizeBitVectors, QF_BV, overflows. https://groups.google.com/u/0/g/smt-lib/c/J4D99wT0aKI. Accessed: 2024-04-04.

[31] Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2013. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *Automated Deduction – CADE-24*. Springer, Berlin, Heidelberg, 208–214. https://doi.org/10.1007/978-3-642-38574-2_14

[32] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2010. A Logical Product Approach to Zonotope Intersection. (2010), 212–226. https://doi.org/10.1007/978-3-642-14295-6_22

[33] Eric Goubault. 2013. Static Analysis by Abstract Interpretation of Numerical Programs and Systems, and FLUCTUAT. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7935)*. Springer, 1–3. https://doi.org/10.1007/978-3-642-38856-9_1

[34] Eric Goubault and Sylvie Putot. 2006. Static Analysis of Numerical Algorithms. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4134)*. Springer, 18–34. https://doi.org/10.1007/11823230_3

[35] Eric Goubault and Sylvie Putot. 2011. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*. Springer, 232–247. https://doi.org/10.1007/978-3-642-18275-4_17

[36] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski. 2023. Ultimate Automizer and the CommuHash Normal Form - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2023 (Lecture Notes in Computer Science)*. 577–581. https://doi.org/10.1007/978-3-031-30820-8_39

[37] Jera Hensel, Constantin Mensendiek, and Jürgen Giesl. 2022. AProVE: Non-Termination Witnesses for C Programs - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022*. Springer, 403–407. https://doi.org/10.1007/978-3-030-99527-0_21

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

[39] Hugo Illous, Matthieu Lemerre, and Xavier Rival. 2017. A Relational Shape Abstract Domain. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10227)*. 212–229. https://doi.org/10.1007/978-3-319-57288-8_15

[40] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[41] Martin Jonás and Jan Strejcek. 2018. Abstraction of Bit-Vector Operations for BDD-Based SMT Solvers. In *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11187)*. Springer, 273–291. https://doi.org/10.1007/978-3-030-02508-3_15

[42] Martin Jonás and Jan Strejcek. 2020. Speeding up Quantified Bit-Vector SMT Solvers by Bit-Width Reductions and Extensions. In *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12178)*. Springer, 378–393. https://doi.org/10.1007/978-3-030-51825-7_27

[43] Dejan Jovanovic and Leonardo Mendonça de Moura. 2012. Solving Non-linear Arithmetic. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7364)*. Springer, 339–354. https://doi.org/10.1007/978-3-642-31365-3_27

[44] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures - An Algorithmic Point of View*. Springer, Chapter 6. https://doi.org/10.1007/978-3-540-74105-3

[45] 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

[46] Jan Leike and Matthias Heizmann. 2018. Geometric Nontermination Arguments. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10806)*. Springer, 266–283. https://doi.org/10.1007/978-3-319-89963-3_16

[47] 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* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 601–612.

[48] Nuno P. Lopes, Levent Aksoy, Vasco M. Manquinho, and José Monteiro. 2010. Optimally Solving the MCM Problem Using Pseudo-Boolean Satisfiability. *CoRR* abs/1011.2685 (2010). arXiv:1011.2685 http://arxiv.org/abs/1011.2685

[49] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[50] 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* (,

San Francisco, CA, USA,) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1177–1189. https://doi.org/10.1145/3611643.3616357

[51] Benjamin Mikek and Qirun Zhang. 2024. *STAUB: PLDI Artifact Evaluation.* https://doi.org/10.5281/zenodo.10895770

[52] Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[53] 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

[54] Casey B Mulligan. 2016. *Automated Economic Reasoning with Quantifier Elimination.* Working Paper 22922. National Bureau of Economic Research. https://doi.org/10.3386/w22922

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

[56] 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

[57] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark W. Barrett, and Cesare Tinelli. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11628)*. Springer, 279–297. https://doi.org/10.1007/978-3-030-24258-9_20

[58] Pierluigi Nuzzo, Alberto Puggelli, Sanjit A. Seshia, and Alberto Sangiovanni-Vincentelli. 2010. CalCS: SMT Solving for Non-Linear Convex Constraints. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Lugano, Switzerland) *(FMCAD '10)*. FMCAD Inc, Austin, Texas, 71–80.

[59] Christos H. Papadimitriou. 1981. On the complexity of integer programming. *J. ACM* 28, 4 (Oct 1981), 765–768. https://doi.org/10.1145/322276.322287

[60] Jaideep Ramachandran and Thomas Wahl. 2016. Integrating Proxy Theories and Numeric Model Lifting for Floating-Point Arithmetic. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, Texas, 153–160.

[61] Andrew Reynolds, Haniel Barbosa, Cesare Tinelli, Aina Niemetz, Andres Noetzli, Mathias Preiner, and Clark Barrett. 2018. Rewrites for SMT Solvers Using Syntax-Guided Enumeration. http://homepage.divms.uiowa.edu/~ajreynol/pres-smt2018.pdf

[62] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*. Springer, 198–216. https://doi.org/10.1007/978-3-319-21668-3_12

[63] 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

[64] Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. 2023. Algorithm selection for SMT: MachSMT: machine learning driven algorithm selection for SMT solvers. *Int. J. Softw. Tools Technol. Transf.* 25, 2 (feb 2023), 219–239. https://doi.org/10.1007/s10009-023-00696-0

[65] Da Shen and Yuliya Lierler. 2018. SMT-Based Constraint Answer Set Solver EZSMT+ for Non-Tight Programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*. AAAI Press, 67–71.

[66] Alfred Tarski. 1998. A Decision Method for Elementary Algebra and Geometry. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer Vienna, 24–84.

[67] 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

[68] 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

[69] 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

[70] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast bit-vector satisfiability. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 38–50. https://doi.org/10.1145/3395363.3397378

[71] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI, Article 174 (jun 2023), 25 pages. https://doi.org/10.1145/3591288

[72] Aleksandar Zeljic, Peter Backeman, Christoph M. Wintersteiger, and Philipp Rümmer. 2018. Exploring Approximations for Floating-Point Arithmetic Using UppSAT. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10900)*. Springer, 246–262. https://doi.org/10.1007/978-3-319-94205-6_17

[73] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. 2017. An Approximation Framework for Solvers and Decision Procedures. *J. Autom. Reason.* 58, 1 (2017), 127–147. https://doi.org/10.1007/S10817-016-9393-1

[74] Yoni Zohar, Ahmed Irfan, Makai Mann, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. 2022. Bit-Precise Reasoning via Int-Blasting. In *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13182)*. Springer, 496–518. https://doi.org/10.1007/978-3-030-94583-1_24

Received 2023-11-16; accepted 2024-03-31