Stochastic Local Search for Solving Floating-Point Constraints*

Shaobo He, Marek Baranowski, and Zvonimir Rakamarić

School of Computing, University of Utah, Salt Lake City, USA
{shaobo,baranows,zvonimir}@cs.utah.edu

Abstract. We present OL1V3R, a solver for the SMT floating-point theory that is based on stochastic local search (SLS). We adapt for OL1V3R the key ingredients of related work on leveraging SLS to solve the SMT fixed-sized bit-vector theory, and confirm its effectiveness by comparing it with mature solvers. Finally, we discuss the limitations of OL1V3R and propose solutions to make it more powerful.

1 Introduction

Numeric computations realized by floating-point arithmetic have become ubiquitous. For example, machine learning applications, often implemented in floating-point arithmetic, are used everywhere. However, unlike its exact real arithmetic counterpart, floating-point arithmetic exhibits unintuitive properties that complicate developers' reasoning about programs. For example, rounding of floating-point arithmetic breaks properties that would otherwise hold for exact real arithmetic (e.g., associativity), resulting in unexpected bugs.

Software verifiers based on *satisfiability modulo theories* (SMT) solvers have been successfully used to help developers (semi-)automatically check correctness of programs. However, the current successful use cases are limited to integer programs. The SMT floating-point theory (QF_FP) [2,14] is relatively new compared to the theories used to model integer programs. Moreover, state-of-the-art decision procedures for QF_FP are based on bit-blasting, which converts SMT formulas into Boolean circuits solved using highly-efficient SAT solvers. While bit-blasting can often effectively solve even large formulas in the theory of bit-vectors (QF_BV), it typically scales poorly in the presence of more complicated operations such as multiplication. Moreover, Boolean circuits derived from bit-blasting the QF_FP operations are typically much larger than those produced for the QF_BV ones, which makes bit-blasting for QF_FP even more brittle.

As a result, search-based incomplete solvers, such as XSat [6], goSAT [9], and JFS [10, 11], emerged as solutions for the scalability issues of bit-blasting. The basic idea of such solvers is to cast the problem of solving QF_FP constraints to an optimization problem, which is then solved using off-the-shelf optimizers. Such

* Supported in part by the National Science Foundation (NSF) awards CCF 1552975 and CCF 1704715.

solvers, although demonstrating large speed-ups on certain benchmarks over using bit-blasting, suffer from stability and scalability issues due to relying on external optimization engines not tailored for the particular types of optimization problems they generate. For example, XSat often returns incorrect answers on SMT-LIB [16] benchmarks [11]. Moreover, to the best of our knowledge, none of the existing search-based solvers support arbitrary precision floating-point numbers as allowed by the theory specification.

In this paper, we present our attempt to build a stable search-based QF_FP solver. Our solver, OL1V3R,[1] is based on stochastic local search (SLS) [8], which was previously implemented in the qfbv-sls tactic of Z3 [4] to solve QF_BV constraints [5]. OL1V3R supports common operations specified in the QF_FP theory, including custom-sized floating-points.[2] We evaluate it on a set of SMT-LIB benchmarks and our preliminary results are encouraging: despite using software-emulated arithmetic, OL1V3R achieves performance comparable to mature SMT solvers such as Z3 and MathSAT [3]. Finally, we discuss the lessons we learned and propose directions for improvements.

2 Approach

2.1 Input Grammar

The SMT floating-point theory [2, 14] mimics the IEEE standard 754-2008 and also defines conversions between the floating-point and other sorts such as reals. Similar to the qfbv-sls tactic of Z3, we first convert an input formula into its negation normal form with the following grammar:

$$\begin{aligned}
\langle \mathit{formula} \rangle &::= (\wedge \langle \mathit{lexpr} \rangle^{*}) \\
\langle \mathit{lexpr} \rangle &::= (\wedge \langle \mathit{lexpr} \rangle^{*}) \mid (\vee \langle \mathit{lexpr} \rangle^{*}) \mid \langle \mathit{atom} \rangle \mid \neg \langle \mathit{atom} \rangle \\
\langle \mathit{atom} \rangle &::= \top \mid \bot \mid \langle \mathit{id} \rangle \mid (\langle \mathit{pred} \rangle \langle \mathit{nlexpr} \rangle^{*})
\end{aligned}$$

Here, an atom is either a logical constant, Boolean literal, or predicate over non-logical expressions. Unlike qfbv-sls, we do not reduce predicates into a minimal set, such as by desugaring $x \leq y$ into $x < y \vee x = y$, because reductions like this are not trivial for floating-point predicates due to the existence of NaNs.

2.2 Search Algorithm

We use a simple search algorithm in Fig. 1 to iteratively search for an assignment to variables α that maximizes an objective function which we call the score function. The algorithm takes as input a formula F that follows the grammar we introduced in the previous section. During the initialization (function *initialize*),

[1] We made OL1V3R publicly available at `https://github.com/soarlab/OL1V3R`.

[2] The floating-point formats supported by OL1V3R depend on the capabilities of the underlying "bigfloat" library it employs to evaluate floating-point arithmetic.

```
function SOLVE($F$)
    $\alpha \leftarrow initialize()$
    $i \leftarrow 1$
    for $i \leq maxStep$ do
        if $satisfies(F, \alpha)$ then return $sat$
        else
            $A \leftarrow selectAssertion(F, \alpha)$
            $\alpha \leftarrow selectMove(F, A, \alpha)$
        end if
        $i \leftarrow i + 1$
    end for
    return $unknown$
end function
```

```
function SELECTMOVE($F, A, \alpha$)
    $V \leftarrow getVars(A)$
    if $uniform(0, 1) \leq p$ then
        return $randomWalk(V, \alpha)$
    else
        $\alpha' \leftarrow getBestMove(F, V, \alpha)$
        if $score(F, \alpha') \leq score(F, \alpha)$ then
            return $random(\alpha)$
        else
            return $\alpha'$
        end if
    end if
end function
```

Fig. 1. Pseudocode of the OL1V3R's algorithm. Function SOLVE performs top-level search, whereas SELECTMOVE invokes the core components (score computation, neighbor selection, randomization). Constants *maxStep* and *p* are input parameters of the algorithm.

we assign $+zero$ to each floating-point variable and *true* to each Boolean variable. When the number of search steps exceeds a preset limit *maxStep*, our algorithm returns *unknown* and terminates. If the algorithm finds an assignment that satisfies a formula (function *satisfies*), it returns *sat*. (We describe how we implement *satisfies* in Sec. 3.) Otherwise, a move (i.e., a mutation to the current assignment) is selected to continue the search. We adopt the heuristic from the qfbv-sls tactic that chooses the candidate variables to mutate from the assertion that is not satisfied and has the highest score (function *selectAssertion*). The selected assertion is passed into function *selectMove* that generates the next assignment.

In *selectMove*, we first attempt to perform a *random walk* with a probability p (defaults to 0.001), where a neighbor is randomly chosen (function *randomWalk*). Then, the neighbor of the variables reachable from the assertion (function *getVars*) that improves the score of the formula the most (function *getBestMove*) is chosen to continue the search. If there does not exist an improving neighbor, a randomized assignment is used for the next step. This algorithm is akin to the more sophisticated one proposed in Fröhlich et al. [5], and we discuss the potentially useful advanced heuristics in Sec 4. Our approach is sound (provided the semantics of the SMT floating-point theory are correctly modeled), whereas it is incomplete since it cannot prove that a formula is unsatisfiable. In the rest of this section, we describe the key components of our algorithm.

Score Computation Score function drives the search, and in Fig. 2 we define it recursively for logical expressions. The score of a satisfying logical expression is 1, which is the maximum score, and otherwise it is in the range $[0, 1)$. Currently, we treat unary predicates, such as *fp.isNaN*, as Boolean variables. Although some of these predicates (e.g., *fp.isInfinite*) describe qualities of floating-point

$$
\begin{aligned}
s(\wedge e_1 \dots e_n, \alpha) &= \frac{1}{n}\sum_{i=1}^{n} s(e_i, \alpha) \\
s(\vee e_1 \dots e_n, \alpha) &= \mathit{max}(s(e_1, \alpha) \dots s(e_n, \alpha)) \\
s(\mathit{fp.lt}\ e_1\ e_2, \alpha) &= \begin{cases} 1 & \mathit{fp.lt}\ e_1|_\alpha\ e_2|_\alpha \\ 0 & e_1|_\alpha = \text{NaN} \vee e_2|_\alpha = \text{NaN} \\ c(1 - \frac{|\mathit{fpPos}(e_1|_\alpha) - \mathit{fpPos}(e_2|_\alpha)|+1}{2^n}) & \text{otherwise} \end{cases} \\
s(\neg(\mathit{fp.lt}\ e_1\ e_2), \alpha) &= \begin{cases} 1 & \neg(\mathit{fp.lt}\ e_1|_\alpha\ e_2|_\alpha) \\ c(1 - \frac{|\mathit{fpPos}(e_1|_\alpha) - \mathit{fpPos}(e_2|_\alpha)|}{2^n}) & \text{otherwise} \end{cases} \\
s(\mathit{atom}, \alpha) &= \begin{cases} 1 & \mathit{atom}|_\alpha \\ 0 & \text{otherwise} \end{cases} \\
s(\neg \mathit{atom}, \alpha) &= 1 - s(\mathit{atom}, \alpha)
\end{aligned}
$$

Fig. 2. Definition of the score function s. Here, $e|_\alpha$ evaluates an expression e using an assignment α, and $c \in (0, 1)$ is a parameter that scales the score of an assignment that is not satisfying (defaults to 0.5). In the last two equations, *atom* refers only to Boolean variables and unary predicates, and not relational operators such as *fp.eq*, *fp.lt*, and *fp.gt*. For relational operators, we only show the floating-point less-than predicate and its negation; the score for other predicates is computed similarly.

numbers and thus should have designated score function definitions, we leave those for future work.

The qfbv-sls tactic defines the score function of a relational operator $b_1 \bowtie b_2$ using the distance between two integers $\mathit{bv2nat}(b_1)$ and $\mathit{bv2nat}(b_2)$, where *bv2nat* maps a bit-vector to its unsigned integer value. Similarly, we define our score function of a floating-point relational operator $x_1 \bowtie x_2$ using the distance between two integers $\mathit{fpPos}(x_1)$ and $\mathit{fpPos}(x_2)$, where *fpPos* is defined as follows:

$$
\mathit{fpPos}(x) = \begin{cases} \mathit{bv2nat}(\mathit{fp2bv}(x)) & \mathit{fp.isPositive}(x) \\ 2^{bw-1} - \mathit{bv2nat}(\mathit{fp2bv}(x)) & \mathit{fp.isNegative}(x) \end{cases}
$$

Here, function *fp2bv* converts a non-NaN floating-point number x to its bit representation, and bw is the bit-width of x. For example, $\mathit{fpPos}(+\mathit{zero}) = 0$ and $\mathit{fpPos}(-\mathit{zero}) = 0$. Also, *fpPos* maps the maximum negative floating-point number to -1. The predicate $x < 0.0$ has higher score when $x = 0.5$ as opposed to $x = 1.0$ since $fpPos(0.5)$ is less than $fpPos(1.0)$. Note that function *fpPos* is not defined for NaNs, and the score rules in Fig. 2 ensure that a NaN cannot be passed as an argument to it.

Our score definition of equality is different from the qfbv-sls tactic and Niemetz et al. [13], which rely on the Hamming distance between two bit-vectors. In the case of floating-points, using the Hamming distance typically leads to worse performance than using our definition based on floating-point number *positions* as captured by *fpPos*. This is expected because floating-point numbers by definition represent numerical values, whereas bit-vectors are not necessarily interpreted as integers.

Neighborhood Relation The neighborhood relation, which is used in functions *randomWalk* and *getBestMove* in Fig. 1, maps a candidate assignment into the set of its neighbors. The neighbors of an assignment with respect to a set of variables are the union of all the neighbors of each variable. The core neighbors of a floating-point number x that has a unique bit representation are all floating-point numbers that differ from x by exactly one bit in their bit representations, i.e., they are obtained by flipping a bit in $\mathit{fp2bv}(x)$. For NaN, we assign a randomly generated floating-point number as its neighbor. In addition to the core bit-flipping neighbors, we also include floating-points obtained by adding ± 1 *unit in the last place* into the neighborhood relation.

Randomization We empirically observed that prioritizing special floating-point numbers, such as ±infinity, during the search exhibits better performance. Hence, the randomization of floating-point values used in OL1V3R (function *random* in Fig. 1) generates special numbers with high probability (default of 0.8), and otherwise it selects a floating-point number uniformly from the set of floating-point numbers.

3 Implementation and Experiments

We implemented OL1V3R in Racket. We use its *math/bigfloat* [17] module to evaluate floating-point expressions (function *satisfies* in Fig. 1), which supports any significand bit-width greater than 2, but only a fixed-size 31-bit exponent. Hence, our implementation checks for overflows and underflows of all floating-point operations with exponent bit-widths less than 31, and handles them as special cases to ensure the soundness of the evaluation. The bigfloat module binds to the GNU MPFR library [12], which emulates arbitrary precision floating-point arithmetic in software. This means that performing arithmetic operations over bigfloats is much slower than over common floating-point types with hardware support. Moreover, we compute the score of a logical expression (once we evaluate its floating-point sub-expressions) using rationals, which gives us precise scores but also causes a slowdown. We discuss potential improvements of these design choices in Sec 4.

The benchmarks we use for evaluation are a subset of the non-incremental SMT-LIB QF_FP benchmarks [15] obtained as follows. First, we exclude benchmarks in folder *wintersteiger* as they are mostly trivial regressions used to test correctness rather than performance. Then, we run MathSAT and Z3 on the left-over benchmarks using a 15 minutes time limit, and select only those on which neither tool reports unsat. Finally, all selected benchmarks are preprocessed using the *jfs-opt* tool shipped with JFS because OL1V3R only partially supports the SMT-LIB2 syntax. Our benchmark selection strategy allows us to focus the

Table 1. Comparison between OL1V3R and other solvers. Columns **Sat** and **Unsat** show the numbers of benchmarks where a tool returns sat and unsat, respectively. Column **Unknown** shows the number of benchmarks where a tool fails to provide any results due to out-of-memory or crashes. Column **Timeout** shows the number of benchmarks that time out. Column **Other** shows the number of benchmarks where a tool returns sat while OL1V3R times out; column **OL1V3R** shows the opposite.

Tool	Sat	Unsat	Unknown	Timeout	Other	OL1V3R
OL1V3R	113	0	2	82	–	–
MathSAT	125	1	7	64	15	5
Z3	88	0	10	99	3	30
JFS	113	0	0	84	5	7

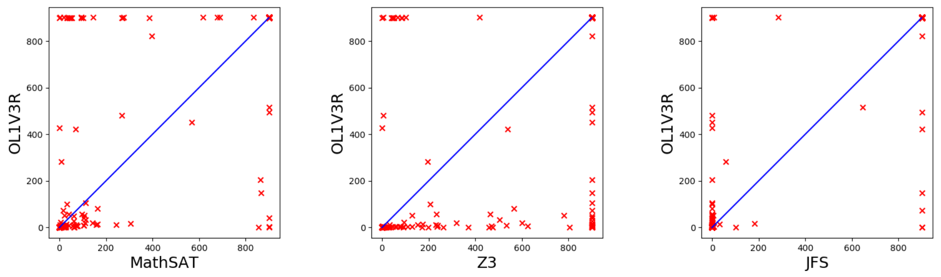

Fig. 3. Comparison of the runtimes of OL1V3R with other solvers.

evaluation on benchmarks that are either satisfiable or possible to be satisfiable since OL1V3R, like other search-based solvers, can only provide sound results for satisfiable formulas. For unsatisfiable formulas, OL1V3R keeps searching for a solution until it hits the time limit, which is also the behavior of JFS.

We evaluate OL1V3R by comparing it with two state-of-the-art SMT solvers, MathSAT [3] (version 5.5.4) and Z3 [4] (version 4.8.4), as well as one search-based solver that uses coverage-guided fuzzing, JFS [10] (commit 2322167). We exclude XSat [6] and goSAT [9] because previous evaluations show that their performance is inferior to JFS [11]. We leverage *benchexec* [1] to obtain reproducible and rigorous benchmarking. The time and memory limits are 900s and 4GB, respectively; we allocate one CPU core per benchmark. We run the experiments on a d820 node of the Utah Emulab [18] cluster with 4 Intel Xeon E5-4620 CPUs and 128GB DDR3 RAM running Ubuntu 16.04.

Table 1 shows the results of running each tool on the selected benchmarks. We do not observe any inconsistencies between the results returned by the solvers. (MathSAT reports unsat for one benchmark on which it previously timed out because the preprocessing step tends to improve its performance.) In terms of benchmarks solved, MathSAT demonstrates the best performance and Z3 the worst, while JFS and OL1V3R are comparable. In terms of runtimes, OL1V3R is comparable to MathSAT, and also typically better than Z3 and slower than JFS, as shown by the scatter plots in Fig. 3. However, the runtimes of JFS have an extreme distribution: it either solves a benchmark quickly or times out. Most benchmarks that JFS solves OL1V3R also solves within a reasonable amount of time (around 3 minutes).

A deeper analysis reveals that both JFS and OL1V3R perform better on benchmarks that contain complex arithmetic such as division and square root. This observation coincides with the JFS' authors' observation that JFS can be complementary to Z3 or MathSAT [11]. Moreover, OL1V3R demonstrates better performance on small benchmarks that permit few models. We believe this can be attributed to the OL1V3R's distance-sensitive score functions. Finally, OL1V3R emulates floating-point arithmetic in software during the evaluation of floating-point expressions in the score computation. On the other hand, JFS evaluates floating-point computations using the underlying hardware. Hence, we expect the performance gap to be easily reduced if we also leverage the floating-point processing hardware (see Sec. 4).

4 Limitations and Proposed Solutions

Evaluation of the score functions can be more efficient. For example, most use cases of floating-point arithmetic are restricted to single/double precision floating-point representations, and those also have fast hardware support on almost all platforms. Hence, we could implement support for these representations as a special case to leverage the underlying hardware. To overcome the potential, albeit unlikely, soundness issues related to the underlying hardware, we can validate the models returned by OL1V3R by either switching back to the software-emulated arithmetic or using sound solvers such as MathSAT and Z3. Moreover, scores can be computed using double-precision floating-point arithmetic as opposed to the expensive rational arithmetic because our experience indicates that double-precision is almost always precise enough for this purpose. Note that this would not compromise the soundness of the algorithm since the computed scores are only used to guide the search. Finally, we can leverage multi-core parallelism to evaluate the scores using a map-reduce paradigm.

Our search strategy suffers from scalability issues on certain benchmarks and could be improved in several ways. The move selection policy is best-improvement, which implies that the score of every neighbor must be computed. The complexity of finding a neighbor that improves the score of the chosen assertion the most is $O(|V|bw)$, where $|V|$ is the number of variables and bw is the maximum bit-width of the used floating-point representations. Our experimental results show that some assertions contain a large number of variables, thus making move selection extremely slow. Hence, we propose using the first-improvement policy, as opposed to the current best-improvement, when the number of variables is above a chosen limit. The intuition is that first-improvement progresses faster whereas best-improvement spends most of its time in computing and comparing scores. However, first-improvement is highly sensitive to the order of neighbors. For example, flipping an exponent bit changes the score much more dramatically than flipping a significand bit. Therefore, we argue that variable neighborhood search [7] could be beneficial not only for the proposed first-improvement policy but also the default best-improvement one.

To be more specific, neighborhood relations are refined to subgroups by sign-bit, exponent bits, and significand bits. This observation is in accordance with the experiments described by Fröhlich et al. [5]. In essence, the chosen search algorithm should capture the fact that the bit representations of floating-point numbers are structured. Furthermore, the scores of assertions being weighted equally leads to frequent violations of assertions that could be easily satisfied, such as intervals of variables, wasting a lot of cycles that should be spent in searching for solutions that satisfy "hard" assertions. This motivates us to adopt in the future the heuristic in qfbv-sls that adds weights to assertions and dynamically adjust them according to the frequency at which an assertion is satisfied.

Like other search-based solvers, OL1V3R has difficulties in handling benchmarks that contain equalities, which are often produced by software verifiers. The reason is obvious: satisfying equalities often requires many more search steps than satisfying other comparison operators, such as *fp.lt*. A straightforward solution that might be effective in many situations is to eliminate equalities whenever possible. For example, if two variables are asserted to be equal, then all the occurrences of one variable can be replaced with the other.

Acknowledgements We thank Dan Liew for answering our questions related to his work on JFS.

References

1. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer (STTT) **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
2. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: Proceedings of the IEEE International Symposium on Computer Arithmetic (ARITH). pp. 160–167 (2015). https://doi.org/10.1109/ARITH.2015.26
3. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT solver. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 93–107 (2013). https://doi.org/10.1007/978-3-642-36742-7_7
4. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
5. Fröhlich, A., Biere, A., Wintersteiger, C.M., Hamadi, Y.: Stochastic local search for satisfiability modulo theories. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI). pp. 1136–1143 (2015)
6. Fu, Z., Su, Z.: XSat: A fast floating-point satisfiability solver. In: Proceeding of the International Conference on Computer Aided Verification (CAV). pp. 187–209 (2016). https://doi.org/10.1007/978-3-319-41540-6_11

7. Hansen, P., Mladenović, N., MorenoPérez, J.A.: Variable neighbourhood search: Methods and applications. Annals of Operations Research **175**(1), 367–407 (2010). https://doi.org/10.1007/s10479-009-0657-6
8. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Elsevier (2004)
9. Khadra, M.A.B., Stoffel, D., Kunz, W.: goSAT: Floating-point satisfiability as global optimization. In: Proceedings of the Conference on Formal Methods in Computer Aided Design (FMCAD). pp. 11–14 (2017). https://doi.org/10.23919/FMCAD.2017.8102235
10. Liew, D.S.: Constraint solver based on coverage-guided fuzzing. `https://github.com/delcypher/jfs`
11. Liew, D.S.: Symbolic Execution of Verification Languages and Floating-Point Code. Ph.D. thesis, Imperial College London (2018)
12. The GNU MPFR library. `https://www.mpfr.org`
13. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. Formal Methods in System Design (FMSD) **51**(3), 608–636 (2017). https://doi.org/10.1007/s10703-017-0295-6
14. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: Informal Proceedings of the International Workshop on Satisfiability Modulo Theories (SMT) (2010)
15. SMT-LIB benchmarks in the QF_FP theory. `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP`
16. SMT-LIB: The satisfiability modulo theories library. `http://smtlib.cs.uiowa.edu`
17. Toronto, N.: Arbitrary-precision floating-point numbers (Bigfloats). `https://docs.racket-lang.org/math/bigfloat.html`
18. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI). pp. 255–270 (2002). https://doi.org/10.1145/844128.844152