Decision Heuristics in MCSat

Thomas Hader[1], Ahmed Irfan[2](✉)[0000-0001-7791-9021], and
Stéphane Graham-Lengrand[2][0000-0002-2112-7284]

[1] TU Wien, Vienna, Austria
thomas.hader@tuwien.ac.at
[2] SRI International, Menlo Park, CA, USA
{ahmed.irfan,stephane.graham-lengrand}@sri.com

Abstract. The Model Constructing Satisfiability (MCSat) approach to Satisfiability Modulo Theories (SMT) has demonstrated strong performance when handling complex theories such as nonlinear arithmetic. Despite being in development for over a decade, there has been limited research on the heuristics utilized by MCSat solvers as in Yices2. In this paper, we discuss the decision heuristics employed in the MCSat approach of Yices2 and empirically show their significance on QF_NRA and QF_NIA benchmarks. Additionally, we propose new ideas to enhance these heuristics by leveraging theory-specific reasoning and drawing inspiration from recent advancements in SAT solvers. Our new version of the MCSat Yices2 solver not only solves more nonlinear arithmetic benchmarks than before but is also more efficient compared to other leading SMT solvers.

Keywords: SAT · SMT · MCSat · Decision heuristics.

1 Introduction

Satisfiability modulo theory (SMT) is the backbone for countless verification and synthesis techniques that require expressive logical theories like real/integer arithmetic [4,28]. Modern SMT solvers rely on the combination of Boolean level reasoning with theory-specific methods through the Conflict-Driven Clause Learning with theory support (CDCL(T)) paradigm [30] or the Model Constructing Satisfiability (MCSat) approach [23,26]. The former employs off-the-shelf Boolean satisfiability (SAT) solvers as its core reasoning engines that aims to enumerate Boolean assignments, which are subsequently checked for theory compliance by independent theory algorithms. The latter lifts the Boolean-level Conflict-Driven Clause Learning (CDCL) algorithm to the theory level, incrementally building theory assignments alongside Boolean assignments, enabling a closer integration of theory reasoning in the Boolean search process. This approach is particularly effective for handling complex arithmetic theories, e.g. nonlinear arithmetic.

During the last couple of decades, SAT solving has gained impressive performance improvements by various techniques [7], many of which are arguably related to heuristics. These improvements of SAT solvers had positive impact on SMT solvers. For CDCL(T) solvers, performance gains in SAT engines translates to improvements in the propositional core of the SMT solver [18]. However, as theory engines are decoupled from SAT solving, improvements in SAT do not directly translate to better theory engines in general (except for the theories that can be reduced to SAT, e.g. bitvectors [16]). In contrast, MCSat-based SMT engines perform theory reasoning similarly to how SAT solvers handle Boolean reasoning, making them more adaptable to heuristics from SAT solving with minimal modification.

A family of such heuristics are *decision heuristics*. Whenever a decision is to be made to continue the search, the decision heuristic selects a yet unassigned variable and a value to assign it to. Those decisions have a fundamental influence on how the search proceeds. In SAT solving, decision heuristics have had a substantial part in the success of modern solvers [7]. By tuning decision heuristics in MCSat, we show that the performance of MCSat solver can be improved without any changes to the actual theory solving techniques.

Contributions. In this work we present (i) a detailed summary on how well-established decision heuristic techniques from SAT can be adapted to the MCSat-based SMT approach (Table 1) and (ii) solutions for additional challenges and opportunities that arise for domains beyond the Boolean domain. We also (iii) show detailed experimental results on the performance implications for different heuristics in the MCSat engine in Yices2 on the SMT logics QF_NRA and QF_NIA–benchmarks taken from the SMT-LIB [3].

Related Work. The MCSat implementation in Yices2 that powers non-linear integer reasoning [22] has already been utilizing some heuristics discussed in this work (particularly exponential Variable State Independent Decaying Sum (VSIDS) (EVSIDS) and value cache). However, the specifics were not discussed and also no empirical evaluation of their impact was provided in [22]. This paper addresses this gap. Besides Yices2, different MCSat variable selection heuristics in the SMT-RAT [14,1] solver have been presented in [29] for the theory of non-linear reals.

Table 1: Established SAT technologies and their MCSat version

SAT	MCSat	Section
VSIDS	Theory-based VSIDS	3.1
Reason side bumping	Boolean Scaling	
Phase Saving	Value Cache	3.2
Target Phases	Target Cache	
Rephasing	Recaching	

The work of [18] proposed an enhanced interface from CDCL(T)-based solvers to off-the-shelf SAT solvers, which improved the performance of cvc5 [2,33] by tightly integrating the state-of-the-art SAT solver CaDiCaL.

MathSAT5 [13,12,11], which is a CDCL(T)-based SMT solver, allows for the use of different pluggable SAT solvers as the core SAT solver. However, the authors reported in [13] that there were no significant performance improvements from using external pluggable SAT solvers.

2 MCSat Overview

MCSat applies CDCL-like mechanisms to perform theory reasoning either as a dedicated theory solver (e.g., for non-linear real arithmetic: Z3/nlsat) or as a fully-fledged stand-alone engine (Yices2, SMT-RAT) that is capable of handling multiple theories. The MCSat architecture consists of a *core solver*, an *assignment trail*, and *plugins* for theory reasoning.

Core solver. The core solver explicitly and incrementally constructs models with Boolean and first-order variable assignments – maintained in the assignment trail – while maintaining the invariant that none of the constraints evaluate to false. Fig. 1a shows high-level pseudo-code for the MCSat search procedure. The core solver propagates the trail information by calling the propagation function of each plugin (line 3). If a conflict is found during propagation, it checks if there is any decision to backtrack over (line 9). If so, it learns a lemma, back-jumps (line 11) and continues the loop. If not, it returns UNSAT (line 13). One of the key steps in MCSat is performing conflict analysis when a plugin detects a conflicting state. The lemmas learned via conflict analysis are based on theory-specific explanations, provided by the plugins, of conflicts and propagations. Note that Fig. 1a shows the restarting (lines 4-5) and variable scoring (line 10) mechanisms, which we will discuss in the next section. Furthermore, like SAT solvers, MCSat also performs clause database cleaning, which is not shown in the code for simplicity.

Plugins. They provide assignments for decisions, perform propagations, detect conflicts, and produce explanations. In modern MCSat engines, propositional reasoning is handled like any other theory by a dedicated plugin. When the core solver asserts a formula during search, each plugin (incl. the Boolean plugin) scans the formula and reports to the core solver all sub-terms that are *relevant* to the plugin's theory, namely those sub-terms that "appear as variables" to the theory, a.k.a. *theory variables* (actual variables or terms whose head symbols are not in the theory's signature), and such that a value assignment to those sub-terms would uniquely determine the truth value of the formula according to the theory. Relevant terms are then treated as MCSat variables — the core solver can decide or propagate value assignments for them. Plugins typically keep a set of feasible values for their theory variables. Whenever one of these sets gets empty, a conflict is raised and the plugin provides a conflict *explanation clause* that excludes the current trail and may even contain new terms.

```
int mcsat_solve():
  while (true):
    if (propagate()):
      if (restarting())
        restart()
      else if (!decide()):
        return SAT
    else:
      if (explain()):
        bump_vars()
        backjump()
      else:
        return UNSAT
```

(a) `mcsat_solve` method

```
bool decide():
  variable var = vsids_pop_unassigned()
  if (var != null):
    feasible = false
    if (has_value_cache(var)):
      feasible = try_value_cache(var)
    if (!feasible):
      pick_new_value(var)
  return (var != null)
```

(b) `decide` method

Fig. 1: MCSat search pseudo-code

Trail. The *trail* is the key data-structure in MCSat. It holds value assignments for relevant terms, functioning as a partial model during the search process (and turning into a complete model when the search concludes SAT). A term t can be *evaluated* (or is *evaluable*) in the trail M if t has an assignment in M, or if all closest relevant sub-terms of t have been assigned values in M. Evaluation-consistency is maintained in the trail, ensuring that no term evaluates to different values within it. Assignments on the trail can either be *propagated* or *decided*.

Example 1. Assume a search problem with integer ($\mathbb{Z}$) variables x, y, and z and boolean ($\mathbb{B}$) terms in the input formula $\mathcal{F}$.

$$\mathcal{F} \;=\; (\neg(x \geq 1) \vee (xy = 1)) \;\wedge\; (\neg(xy = 1) \vee (x + 2yz > 0)) \;\wedge\; (z^2 > 1)$$

A possible trail is $M = [\![(z^2 > 1) \mapsto \top, x \mapsto 1, (x \geq 1) \mapsto \top, (xy = 1) \mapsto \top, y \mapsto 1]\!]$. It consists of a $\mathbb{B}$-propagation, a $\mathbb{Z}$-decision, a $\mathbb{Z}$-propagation, a $\mathbb{B}$-propagation, and another $\mathbb{Z}$-propagation, respectively. The choice to select x was made arbitrary; any variable – Boolean or integer – which was not already on the trail could have been chosen. A different choice, say z, or a different value, say -1, would have led to different propagations and the search in a different direction.

Decision Heuristic. The purpose of this heuristic is to pick (i) a variable to decide, and (ii) a value to assign to the variable. The former is part of the MCSat core as the decision is made over *relevant terms* of all involved theories. Theory plugins can, nevertheless, influence the heuristic by increasing the weight of a variable and make it more likely to be picked. This process is called *bumping* a variable. Selecting a value is done by the relevant theory plugin. In Example 1, the variable x was picked by the central selection heuristic and the value 1 was chosen by the dedicated plugin for integer reasoning. As we will show in the following section, decision heuristics have a crucial impact on solver performance.

3 MCSat Decision Heuristics

In this section, we discuss the decision heuristic for variable and value selection that have been implemented in the MCSat scheme of the Yices2 SMT solver. The importance of these heuristics is demonstrated through experiments, which have not been previously explored in earlier Yices2 MCSat papers [22,19,20,21]. Additionally, we introduce new ideas on how to further improve the heuristics by incorporating theory-specific reasoning and insights from the SAT community. Finally, we evaluate the effectiveness of each proposed change through our experiments. For the experiments,[1] we used the benchmarks of quantifier-free logics of nonlinear real and integer arithmetic, namely QF_NRA and QF_NIA, from the SMT-LIB [3] release 2024 [32].

3.1 Variable Selection Heuristics

In SAT solving, heuristics for selecting the next variable usually follow the principle that variables which often appear in learned clauses are central to the problem structure and, thus, should be assigned early. This principle is reflected in the VSIDS variable selection heuristic, which was originally presented in the SAT solver Chaff [25]. An adaption named EVSIDS, originally implemented in the MiniSAT solver [17], is implemented in many state-of-the-art SAT solvers (e.g. CaDiCaL [5]) and has been crucial for their high efficiency [8,7]. Modern MCSat engines (e.g. Yices2, SMT-RAT) have adapted a variant of the VSIDS variable selection heuristic [23,29]. While in the SAT community other variable selection heuristics – like variable-move-to-front (VMTF) – have been proposed in the meantime [34,8] and modern solvers use heuristics-switching strategies on the fly [5], EVSIDS remains the state-of-the-art in MCSat engines.

EVSIDS Heuristic [Baseline]. In EVSIDS each variable x_i gets an activity score s_i. Whenever x_i gets bumped, s_i is increased by g^n where $g > 1$ and n is an integer that is increased at each conflict. At a decision, the variable with the highest score (efficiently determined using a priority heap), is chosen. In SAT solving, variables are bumped when they appear in conflict resolutions and different bumping strategies have been proposed. For further details we refer to relevant publications in the area of SAT solving [17,8].

In the MCSat implementation of Yices2, variables are bumped whenever they occur in a conflict resolution, i.e. either in the conflict clause or a resolution step. All variables are bumped once for each term they occur in, as long as the term is not assumed to be false on the trail by Boolean reasoning. Note that, in general, this favors non-propositional variables as they occur in many different propositional terms. Previous research on the SMT-RAT solver in the QF_NRA theory indicates that this behavior is beneficial for performance [29].

[1] The experiments were conducted on a 96-core 2.3 GHz AMD-CPU server running Ubuntu 24.04.1 LTS. Timeout per instance 5 minutes; memory limit 8 GB.

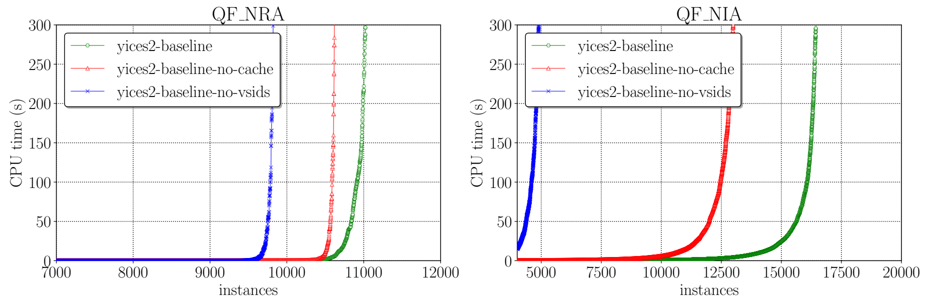

Fig. 2: Evaluation of state-of-the-art heuristics: EVSIDS and value cache

Example 2. Given a trail M and a conflict resolution containing $C_1 = x^2y + 1 > 0$, $C_2 = x + z \geq 0$, and $C_3 = y^4 > 0$. Assume M contains a Boolean assignment mapping C_2 to false. Then all related terms C_1, C_2, and C_3 are bumped (once) and the variables x and y are bumped 1 and 2 times, respectively.

Theory bumping. Theory plugins can influence the search by bumping variables using additional, theory-based heuristics. The following examples of theory-based bumping are implemented in Yices2:

– Whenever the *real or finite field plugins* detect a new term, they bump all variables according to their polynomial degree.
– The *bit-vector plugin* bumps (certain) variables of new terms once.
– The plugin for *uninterpreted functions and arrays* bumps sub-terms of conflicts in the equality graph.

What performance benefit does EVSIDS provide in MCSat? We have evaluated the influence of EVSIDS with theory bumping against the baseline without EVSIDS, i.e. using a random, fixed variable order. The results are shown in Fig. 2. They clearly show that all EVSIDS is giving a huge performance boost over the fixed variable ordering.

Extending EVSIDS with Boolean Scaling [New Heuristic]. We could confirm that EVSIDS is a leading technique for dynamic variable selection [29]. Although we did not recreate all experiments presented, the reader can refer to [29] for a further evaluation on different variable ordering. However, the uniform bumping strategy in Yices2 can potentially reduce the chances of selecting a Boolean variable as the next decision. To mitigate this scenario, we introduce the *Boolean scaling* constant to increase the bumping of Boolean terms. It builds upon another idea from SAT solving, where state-of-the-art solvers put emphasis on literals of resolved clauses, instead of the conflict clause directly [5]. This idea was first presented in [24] by accounting for the *reason side rate*. In Yices2, whenever the conflict clause C is resolved with another clause D, all literals in $D \setminus C$ are bumped by the Boolean scaling factor. We use a factor of 20 for the Boolean scaling. The results comparing different Boolean scaling are in Section 3.3.

```
mcsat_solve():
  while (true):
    if (propagate()):
      if (restarting()):
        clear_hints()
        update_cache()
        restart()
      else:
        if (recaching())
          recache()
        if (!decide())
          return SAT
    else:
      if (explain()):
        bump_vars()
        backjump()
        clear_hints()
        update_cache()
      else:
        return UNSAT;
```

(a) Modified `mcsat_solve` method

```
1  bool decide():
2    variable var = hints_pop_unassigned()
3    if (var != null):
4      var = vsids_pop_unassigned()
5    if (var != null):
6      feasible = false;
7      if (has_target_cache(var)):
8        feasible = try_target_cache(var)
9      else if (has_value_cache(var)):
10       feasible = try_value_cache(var)
11     if (!feasible):
12       pick_new_value(var)
13   return (var != null);
```

(b) Modified `decide` method

Fig. 3: Modified MCSat search pseudo-code (italicized and green colored)

Theory-Guided Variable Selection [New Heuristic]. While theory plugins can influence EVSIDS scoring with theory bumping, there are situations in the search where bypassing EVSIDS and directly suggesting a next variable is beneficial. Increasing the score of a variable substantially to ensure its selection is not wise, as this would destroy scoring for later decisions. We introduce a mechanism that enables theory plugins to suggest the next decision variable, so called *variable hints.* Hinted variables have precedence over EVSIDS and are guaranteed to be selected as soon as possible. Fig. 3b shows the modified MCSat decide method: note that on line 2 hinted terms are selected before the terms suggested by EVSIDS, shown on line 4. As hints are intended to react on specific search states, they are cleared on backtracking. Theory plugins can hint a variable to the core solver whenever they detect, by theory-specific reasoning, in the current search state that the variable is a good candidate for the next decision. This is usually the case when a plugin has a "good" choice for a variable's value, i.e. the number of feasible assignments for a variable is limited or assigning a specific value is expected to have a significant impact on the search.

Example 3. Assume a trail M and two yet unassigned real variables x and y. A new term C is added to M that reduces the set of feasible values for x to a single value via theory reasoning, e.g. in the case of reals, using root isolation. However, $y \in (-\infty, \infty)$ still holds. Then assigning x before y avoids potential incorrect guesses on y when there is another term $D(x, y)$.

In practice, hinting is used whenever the set of feasible assignments for a variable is small or a singleton. Note that even if the set is a singleton, propagating the variable with the singleton value is not generally feasible because it involves generating a term that only contains already assigned variables (cf. [23]). The

```
recache():
  num_recaches = get_num_recaches()
  clear(target_cache)
  target_depth = 0
  if !(num_recaches % 2):
    value_cache = best_cache
    clear(best_cache)
    best_depth = 0
```

(a) `recache` method

```
update_cache():
  if (trail_size > target_depth):
    target_cache = trail
    target_depth = trail_size
  if (trail_size > best_depth):
    best_cache = trail
    best_depth = trail_size
```

(b) `update_cache` method

Fig. 4: MCSat `recache` and `update_cache` methods

term is substituted for the propagated variable when building a lemma out of conflict analysis. In the theories of reals and integers such a term cannot always be found and hinting the next decision is a promising alternative in such cases.

A feasible set interval is an interval that (tightly) over-approximates the feasible set. When its size is small, the associated variable is a good candidate for decision because the possible values are limited. In Section 3.3 we evaluate a modification to the reals/integers plugin that hints a real variable whenever its feasible set is unit or its *feasible set interval* size less or equal to 1.

3.2 Value Selection Heuristics

Whenever a variable is chosen for a decision, the theory plugin responsible for its type needs to find a value. While this is a theory-specific choice, there are theory-independent caching heuristics that improve performance in all evaluated theories.

Value Caching [Baseline]. The idea of value cache is to retain the value of a variable when its assignment is undone. When a decision is to be made for a variable, the previously cached value is used, if it is still feasible. The value caching is a generalization of *phase saving* [31] in SAT solving. This approach is based on the idea that solvers tend to revisit similar parts of the search space repeatedly. In MCSat we use the term *value* instead of *phase* to reflect the bigger space of potential assignments. The MCSat search loop's `decide` method, as shown in Fig. 1b, first attempts the previously assigned value (lines 5-6) by calling the appropriate theory plugin. If the cached value is not valid (lines 7-8), the `decide` method uses a new feasible value provided by the theory plugin.

Does the value caching heuristic provide performance benefits? Fig. 2 shows plots of Yices2 with and without value caching. They show that Yices2 with value caching solves more benchmarks and is a lot faster than the version without value caching. Clearly, the value caching heuristic seems crucial for performance, in particular for the integer benchmarks.

Target, Best Caching, and Recaching [New Heuristic]. While phase saving has been standard for many years in SAT solving, more recent work [10,6] has indicated that different strategies for phase selection are beneficial. Inspired by the success of these caching strategies and the importance of value caching in the MCSat procedure, we have extended the MCSat search procedure to include two additional caches: *target* and *best*, along with a recaching mechanism.

Similar to the concept in SAT, the target cache stores cached values of variable assignments. However, unlike the value cache where values are updated after each decision, which is frequently, the target cache focuses on maintaining a *promising partial assignment* that does not lead to a conflict after propagation. The target cache is updated when the MCSat core solver discovers a 'more promising assignment'. A (partial) assignment is considered 'more promising' than earlier saved assignments if it assigns more variables/terms than the earlier assignments. The target cache update occurs before a restart (when in a non-conflicting state) and after back-jumping during conflict analysis and lemma learning. The updated MCSat search loop is illustrated in Fig. 3a: line 6 and line 18 execute the target cache update before a restart and after back-jumping, respectively.

When making a decision, the value stored in the target cache takes precedence over the value cache when selecting a value for a variable assignment. If the target cache does not have a variable assignment, then the value from the value cache is chosen. The revised `decide` method is depicted in Fig. 3b: lines 7-8 attempt the value from the target cache first, and lines 9-10 try the value from the value cache if needed.

As the target cache is favored for selecting variable assignments and is updated based on an objective function defined by the number of assigned terms, there is a risk of the target cache trapping the search in a local maximum state. To address this issue, we periodically *recache* both the target cache and the value cache, as also done is modern SAT solvers. Recaching is invoked during the main search loop when the recaching limit is reached, as indicated in lines 9-10 in Fig. 3a. Recaching clears the target cache but makes use of another cache, the *best* cache, to retain the most promising assignments. Similar to the target cache, the best cache is updated when a more promising assignment is found: Fig. 4b shows how this is done by tracking the number of assigned terms

Table 2: Evaluation of different Boolean scaling

	Total (37512)	QF_NRA (12154)		QF_NIA (25358)	
Scaling factor	solved	solved	sat/unsat	solved	sat/unsat
1	28777	11197	5522/5675	17580	11964/5616
5	28946	11240	5534/5706	17706	12098/5608
10	29039	11280	5541/**5739**	17759	12099/5660
20	**29101**	11290	**5556**/5734	**17811**	**12143**/**5668**
30	29062	**11293**	5554/**5739**	17769	12119/5650

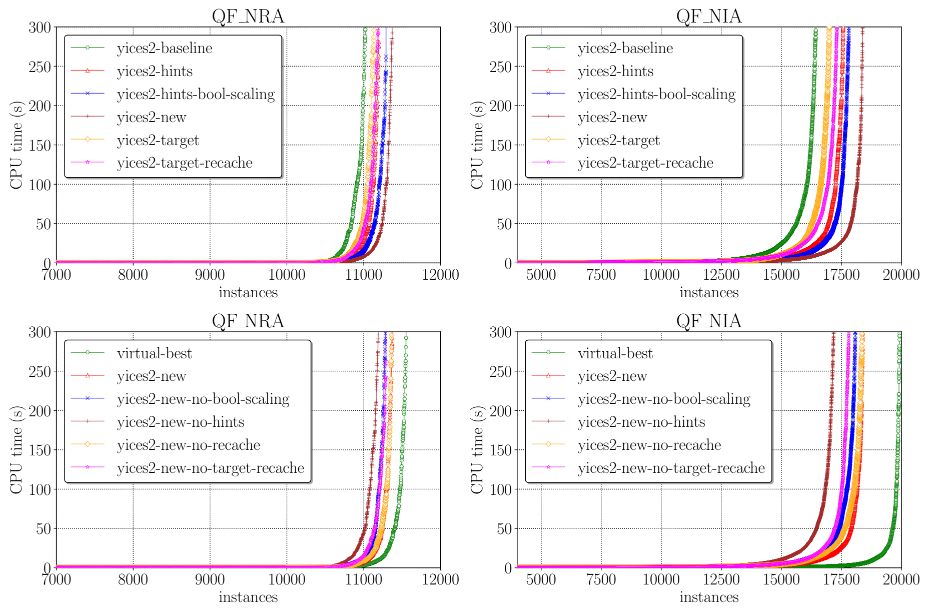

Fig. 5: Evaluation of new heuristics

as `target_depth` and `best_depth` for the target and best caches, respectively. The target cache and the best cache differ at the point of recaching (Fig. 4a): the target cache is cleared at every recaching and the best cache is copied to the value cache at every other recaching.

3.3 Evaluation of New Heuristics

In Table 2, a comparison of Yices2 using hints with different Boolean factors (1, 5, 10, 20, 30) is shown. The table indicates that a Boolean scaling factor of 20 provides the best performance overall (when looking at the QF_NRA and QF_NIA benchmarks in total) among the tested factors.

In Fig. 5, the top row compares the novel heuristics introduced in this work for variable (Boolean scaling, variable hinting) and value selection (target caching and recaching) against the baseline Yices2 (including EVSIDS and value caching). The baseline Yices2 is represented by `yices2-baseline`, with additional features added on top: `yices2-hints` for variable hinting, `yices2-hints-bool-scaling` for hinting with the Boolean scaling factor 20, `yices2-target` for the target cache, `yices2-target-recache` for the target cache with recaching, and `yices2-new` representing the configuration with all techniques.

In the bottom row of Fig. 5, various plots compare `yices2-new` with one heuristic turned off (`yices2-new-no-heuristic` refers to the `yices2-new` configuration without the `heuristic`). Additionally, the virtual best solver for these configurations is plotted as virtutal-best. The plots reveal that all techniques

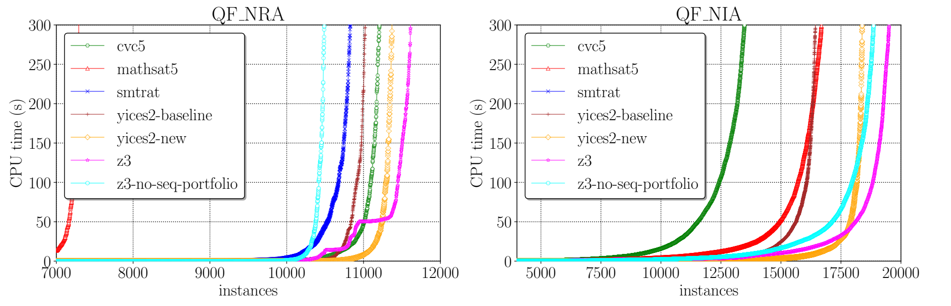

Fig. 6: Solver performance comparison

have a positive impact on performance in QF_NIA and QF_NRA. Interestingly, they contribute to solving different benchmarks, as demonstrated by the virtual best solver, especially in the case of QF_NIA.

We have also evaluated, under the same experimental conditions, the performance improvements on the combined theories QF_UFNRA and QF_UFNIA, which add uninterpreted functions. In the former, we can solve all benchmarks; in the latter we improved by almost 15%, solving 677 out of 806 with `yices2-new` compared to `yices2-baseline`.

4 Comparison Against Other SMT Solvers

We have compared our new solver `yices2-new` against the Yices2 [15] baseline `yices2-baseline`, `cvc5` [2] (version 1.2.0), `mathsat5` [13] (version 5.6.11), `smtrat` [14] (version 24.06), and `z3` [27,9] (version 4.13.3). We noticed that `z3` implements a sequential portfolio approach for nonlinear arithmetic [9] which can clearly be observed in the step-shaped cactus in Fig. 6. Since Yices2 does not (yet) exploit sequential portfolio techniques, we have included in our comparisons a version of `z3` with portfolio solving disabled (`z3-no-seq-portfolio`),

Table 3: Comparison with other SMT solvers. Solved benchmarks within 5 minutes and 8 GB RAM. Total number of benchmarks in top row.

	QF_NRA (12 154)			QF_NIA (25 358)		
Solver	solved	sat/unsat	time(s)	solved	sat/unsat	time(s)
Yices2-baseline	11 022	5 394/5 628	28 829	16 436	11070/5366	141 928
Yices2-new	**11 370**	**5 626**/5 744	23 462	18 387	**12 774**/5 613	98 871
cvc5	11 207	5 428/**5 779**	36 306	13 484	8 922/4 562	312 345
MathSAT5	7 292	2 742/4 550	23 677	16 689	11 451/5 238	314 851
SMT-RAT	10 828	5 345/5 483	47 833	-	-/-	-
Z3-no-seq-portfolio	10 489	5 453/5 036	18 844	**18 861**	12 364/**6 497**	305 923
Z3	**11 612**	**5 761**/**5 851**	67 633	**19 507**	**12 991**/**6 516**	264 204

making it algorithmically much closer, and comparable, to Yices2 MCSat. (The `z3` commands used for i) QF_NRA: z3 tactic.default_tactic="(then simplify propagate-values solve-eqs elim-uncnstr simplify qfnra-nlsat)", and ii) QF_NIA: z3 tactic.default_tactic="(then simplify propagate-values solve-eqs elim-uncnstr simplify smt)".) The results are presented in Table 3 and Fig. 6.

When comparing `yices2-baseline` to `yices2-new`, it is evident that the latter is hugely improved, especially on the QF_NIA benchmarks. `yices2-new` can now solve a much larger number of benchmarks in less time than `yices2-baseline`. In comparison to other non-portfolio SMT solvers, `yices2-new` excels in performance on the QF_NRA benchmarks. While slightly behind `z3` (both versions) on the QF_NIA benchmarks, `yices2-new` outperforms the other solvers. We believe that by exploring other heuristics in Yices2 MCSat, as well as sequential portfolio techniques, the gap between `z3` and `yices-new` can be further reduced.

5 Conclusion

In this work, we have demonstrated the importance of decision heuristics in the MCSat search procedure. Our empirical results have shown that dynamic variable ordering is crucial for having a performant MCSat solver. Additionally, we have shown that value caching is also vital for the performance of an MCSat solver based on our experiments.

We introduced a theory-guided hinting mechanism that enhances the variable selection and improves value selection techniques through the combination of value and target caches with recaching. Our evaluation indicates that theory-guided hinting provides the most significant performance boost, followed by target caches with recaching, and then Boolean scaling. With these new heuristics integrated, Yices2 is now more efficient and solves more benchmarks than other state-of-the-art solvers.

Our work is inspired by the recent advancements in propositional SAT solving. The positive results of our work suggest the potential for applying well-established SAT heuristics in the MCSat context.

In the future, we would like to conduct a more comprehensive empirical evaluation by including additional theory benchmarks. Moreover, we would explore other SAT heuristics like the VMTF decision heuristic, chronological backtracking, dynamic restart and clause database cleaning strategies based on the literal block distance (LBD) concept.

Acknowledgments. The authors thank Mathias Fleury for interesting insights into SAT solving techniques as well as Laura Kovács and Daniela Kaufmann for valuable feedback. We acknowledge funding from ERC Consolidator Grant ARTIST 101002685, the TU Wien SecInt Doctoral College, the NSF award CCRI-2016597, and from SRI Internal Research And Development funds. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US Government or NSF.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. J. Log. Algebraic Methods Program. **119**, 100633 (2021)
2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: TACAS (1). Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022)
3. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
4. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021). https://doi.org/10.3233/FAIA201017, https://doi.org/10.3233/FAIA201017
5. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: CaDiCaL 2.0. In: International Conference on Computer Aided Verification. pp. 133–152. Springer (2024)
6. Biere, A., Fleury, M.: Chasing target phases. In: Workshop on the Pragmatics of SAT (2020)
7. Biere, A., Fleury, M., Froleyks, N., Heule, M.J.: The SAT museum. In: POS@ SAT. pp. 72–87 (2023)
8. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18. pp. 405–422. Springer (2015)
9. Bjørner, N.S., Nachmanson, L.: Arithmetic solving in Z3. In: CAV (1). Lecture Notes in Computer Science, vol. 14681, pp. 26–41. Springer (2024)
10. Cai, S., Zhang, X., Fleury, M., Biere, A.: Better decision heuristics in cdcl through local search and target phases. Journal of Artificial Intelligence Research **74**, 1515–1563 (2022)
11. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Experimenting on solving nonlinear integer arithmetic with incremental linearization. In: SAT. Lecture Notes in Computer Science, vol. 10929, pp. 383–398. Springer (2018)
12. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. ACM Trans. Comput. Log. **19**(3), 19:1–19:52 (2018)
13. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013)
14. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: SAT. Lecture Notes in Computer Science, vol. 9340, pp. 360–368. Springer (2015)
15. Dutertre, B.: Yices 2.2. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014)
16. Dutertre, B.: An empirical evaluation of SAT solvers on bit-vector problems. In: SMT. CEUR Workshop Proceedings, vol. 2854, pp. 15–25. CEUR-WS.org (2020)

17. Eén, N., Sörensson, N.: An extensible SAT-solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
18. Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., Biere, A.: Satisfiability modulo user propagators. J. Artif. Intell. Res. **81**, 989–1017 (2024). https://doi.org/10.1613/JAIR.1.16163, https://doi.org/10.1613/jair.1.16163
19. Graham-Lengrand, S., Jovanovic, D., Dutertre, B.: Solving bitvectors with MC-SAT: explanations from bits and pieces. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR), Part I. LNCS, vol. 12166, pp. 103–121. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_7, https://doi.org/10.1007/978-3-030-51074-9_7
20. Hader, T., Kaufmann, D., Irfan, A., Graham-Lengrand, S., Kovács, L.: MCSat-based finite field reasoning in the yices2 SMT solver (short paper). In: IJCAR (1). Lecture Notes in Computer Science, vol. 14739, pp. 386–395. Springer (2024)
21. Irfan, A., Graham-Lengrand, S.: Arrays reasoning in MCSat. In: SMT@CAV. CEUR Workshop Proceedings, vol. 3725, pp. 24–35. CEUR-WS.org (2024)
22. Jovanovic, D.: Solving nonlinear integer arithmetic with MCSAT. In: VMCAI. Lecture Notes in Computer Science, vol. 10145, pp. 330–346. Springer (2017)
23. Jovanovic, D., Barrett, C., de Moura, L.: The design and implementation of the model constructing satisfiability calculus. In: Intl. Conf on Formal Methods in Computer-Aided Design (FMCAD). pp. 173–180. IEEE (2013). https://doi.org/10.1109/FMCAD.2013.7027033
24. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19. pp. 123–140. Springer (2016)
25. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535 (2001)
26. de Moura, L., Jovanovic, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Intl. Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 7737, pp. 1–12. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_1, https://doi.org/10.1007/978-3-642-35873-9_1
27. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
28. de Moura, L.M., Bjørner, N.S.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011)
29. Nalbach, J., Kremer, G., Ábrahám, E.: On variable orderings in MCSAT for non-linear real arithmetic. In: SC-square@ SIAM AG (2019)
30. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Baader, F., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 3452, pp. 36–50. Springer (2004). https://doi.org/10.1007/978-3-540-32275-7_3, https://doi.org/10.1007/978-3-540-32275-7_3
31. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings 10. pp. 294–299. Springer (2007)
32. Preiner, M., Schurr, H.J., Barrett, C., Fontaine, P., Niemetz, A., Tinelli, C.: SMT-LIB release 2024 (non-incremental benchmarks) (Apr 2024). https://doi.org/10.5281/zenodo.11061097, https://doi.org/10.5281/zenodo.11061097

33. Reynolds, A., Tinelli, C., Jovanovic, D., Barrett, C.W.: Designing theory solvers with extensions. In: FroCoS. Lecture Notes in Computer Science, vol. 10483, pp. 22–40. Springer (2017)
34. Ryan, L.: Efficient algorithms for clause-learning SAT solvers (2004)