Arrays Reasoning in MCSat

Ahmed Irfan[1,*], Stéphane Graham-Lengrand[1]

[1] *SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025 USA*

Abstract

In this paper, we present the support for the (extensional) theory of arrays in the MCSat scheme for SMT solving. We describe its implementation in the (MCSat component of) the Yices2 SMT-solver, allowing Yices2 to solve, for the first time, benchmarks that combine arrays with nonlinear arithmetic. Our experimental results show that this implementation outperforms other state-of-the-art SMT solvers when solving such benchmarks (QF_ANIA + QF_AUFNIA), and also demonstrates decent performance on other SMT logics that involve arrays.

1. Introduction

A key mathematical concept in the modeling of both software and hardware systems is the concept of (persistent) arrays. For example, arrays are used to model (mutable) array data structure or the memory heap (e.g., Frama-C tool for C) for program analysis [1, 2, 3, 4, 5], as well as memories [6, 7, 8] for hardware design verification. Efficient methods for reasoning about arrays are essential for analyzing and understanding software and hardware.

Dedicated methods for array reasoning are typically found in Satisfiability Modulo Theories (SMT) solvers, which provide the automated reasoning capabilities of numerous formal methods toolchains. These solvers check the satisfiability of a first-order formula in some background theory $\mathcal{T}$ (or combination of theories). If the formula is true in some model of the theory(ies), it is said to be satisfiable (SAT). If not, it is said to be unsatisfiable (UNSAT). While most SMT solvers traditionally follow the CDCL($\mathcal{T}$) [9] scheme for solving (i.e., deciding) such satisfiability problems, the Model-Constructing Satisfiability (MCSat) [10] scheme offers a deeper integration of Boolean and theory reasoning that has proved particularly successful for nonlinear arithmetic. MCSat also offers a new explanation functionality generalizing unsat cores to theory reasoning, which provides new algorithms for interpolation [11] and quantifier-supporting reasoning [12].

Yices2 is a state-of-the-art SMT-solver that implements both the CDCL($\mathcal{T}$) and MCSat schemes, in two distinct (sub-)solvers. Nonlinear arithmetic is only supported in the MCSat-based solver, which does offer the aforementioned explanation and interpolation functionalities for all of the theories it supports. Until recently, these theories did not include the theory of arrays, which was only supported in the CDCL($\mathcal{T}$)-based solver. More generally and to the best of our knowledge, no SMT-solver has been supporting arrays in the MCSat approach. Hence, even if a solver uses an MCSat-like procedure to reason about nonlinear arithmetic (e.g., cvc5 [13], Z3 [14]), the nonlinear arithmetic reasoner interacts with the rest of the solver via a traditional theory combination scheme (typically a variant of the Nelson-Oppen scheme [15]) when trying to solve benchmarks involving both arrays and nonlinear constraints (e.g., benchmarks for the SMT logics QF_ANIA and QF_AUFNIA). Yices2 itself did not support this theory combination, as its CDCL($\mathcal{T}$) and MCSat components do not interact. In this work, we address this limitation by extending the MCSat component of Yices2 with array reasoning.

The contributions of this paper are: 1) we describe how to integrate array reasoning in an MCSat-based SMT solver using the concept of *weak equivalence graph* [16]; 2) we describe the implementation of that integration in Yices2 – the source is publicly available on GitHub[1]; 3) we experimentally evaluate

SMT 2024: 22nd International Workshop on Satisfiability Modulo Theories, July 22–23, 2024, Montreal, Canada

[*] Corresponding author.

ahmed.irfan@sri.com (A. Irfan); stephane.graham-lengrand@sri.com (S. Graham-Lengrand)
https://ahmed-irfan.github.io/ (A. Irfan); https://www.csl.sri.com/users/sgl/ (S. Graham-Lengrand)
0000-0001-7791-9021 (A. Irfan); 0000-0002-2112-7284 (S. Graham-Lengrand)

[1] https://github.com/SRI-CSL/yices2

our implementation on the array benchmarks from SMT-LIB and compare it against other SMT solvers.

Notation. We assume the standard many-sorted first-order logical setting with the usual notions of signature, term, formula, and interpretation. A *theory* is a pair $\mathcal{T} = (\Sigma, \mathbf{I})$, where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations that are the *models* of $\mathcal{T}$. A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* $\mathcal{T}$ if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$.

Theory of Arrays. Let $\mathcal{T}_A$ be the standard theory of arrays [17] with extensionality. We assume sorts for arrays, indices, and elements, and function symbols *read* and *write*. Here and rest of the paper, we use a and b to refer to arrays, i and j to refer to array indices, and e to refer to array elements. The theory contains the class of all interpretations satisfying the following axioms:

$$\forall a, i, j, e.\, i = j \implies read(write(a, j, e), i) = e \tag{idx}$$

$$\forall a, i, j, e.\, i \neq j \implies read(write(a, j, e), i) = read(a, i) \tag{read-over-write}$$

$$\forall a, b.\, (\forall i.\, read(a, i) = read(b, i)) \implies a = b \tag{ext}$$

Related Work. The theory of arrays has been a significant focus in the development of SMT solvers since their early days. Two main approaches have been utilized to decide the theory of arrays: *rewriting-based* and *instantiation-based* techniques.

Rewriting-based techniques address the problem using rewrite rules with a specific ordering to ensure completeness. Some notable work in this category can be found in [18, 19].

On the other hand, instantiation-based techniques involve adding instantiations of array axioms, typically done lazily under certain conditions to minimize the number of instantiations. Most CDCL($\mathcal{T}$)-based SMT solvers, such as proposed in [20, 21, 22, 23], utilize instantiation-based approaches.

One specific instantiation-based approach is the *weakly equivalent arrays* method, as implemented in the CDCL($\mathcal{T}$)-based SMT solver SMTInterpol [16, 24], as well as in the CDCL($\mathcal{T}$) solver in Yices2.[2] Our work leverages weakly equivalent arrays reasoning, integrating it into an MCSat-based SMT solver.

2. Weakly Equivalent Arrays

Our approach uses the weakly equivalent arrays decision procedure proposed by [16], which we briefly describe here. The procedure exploits the use of chains of write function applications by introducing the notion of weak equivalence: two arrays connected via a chain of write function application can differ only at finitely many indices. This essentially generalizes the read-over-write axiom to reason about a chain of write applications. The lemmas produced from that reasoning have the advantage of using small number of new terms. In fact, we can compute the set of terms that would appear in those lemmas by scanning the input formula. Note that the finiteness of that set of terms used for generating all lemmas is a key requirement for termination, making MCSat a decision procedure.

Let ϕ be the input formula whose satisfiability is under question and $\mathscr{A}$ be the set of array terms in ϕ.

Definition 1. *The set of* tracked terms V *is defined as:*

$$\begin{aligned} V = \mathscr{A} \cup \{&read(a, i), read(write(a, i, e), i), i, e \mid write(a, i, e) \in \mathscr{A}\} \cup \\ &\{read(a, i), i \mid read(a, i) \in \phi\} \end{aligned} \tag{1}$$

Let $\sim_V \subseteq V \times V$ be an equivalence relation on the set V.

[2] Yices2 implements a similar approach to [16].

Definition 2. *A* weak equivalence graph G^W *is an undirected graph where the vertices are the array terms $\mathscr{A}$ and the edges are either unlabelled or labelled with the indices used in the write function application, defined as follows: 1) there is an unlabelled edge $a \leftrightarrow b$ if $a \sim_V b$, and 2) there is a labelled edge $a \overset{i}{\leftrightarrow} b$ if either a is of the form $write(b, i, \cdot)$ or b is of the form $write(a, i, \cdot)$. Given a path P in G^W, let $Indices(P)$ be the set of indices appearing on the labelled edges in P.*

Definition 3. *Two array terms a and b are called* weakly equivalent *if there exists a path P between nodes a and b in G^W.*

Definition 4. *Two arrays a and b are called* weakly equivalent modulo *i, denoted by $a \approx_i b$, if and only if they are connected by a path that does not contain an edge $\overset{j}{\leftrightarrow}$ where $j \sim_V i$.*

Lemma 1. Read-over-weakeq*: Given an equivalence relation $\sim_V$, $read(a, i)$ and $read(b, j)$ from V, if $i \sim_V j$ and $a \approx_i b$ then $read(a, i) \sim_V read(b, j)$ holds.*

Definition 5. *Given an equivalence relation $\sim_V$, array terms a and b are* weakly congruent modulo *i, denoted by $a \cong_i b$, if and only if they have the same value at index i.*

$$a \cong_i b \ := a \approx_i b \vee (\exists a' b' jk. a \approx_i a' \wedge i \sim_V j \wedge read(a', j) \sim_V read(b', k) \wedge k \sim_V i \wedge b' \approx_i b)$$

Lemma 2. Weakeq-ext*: Given an equivalence relation $\sim_V$, if array terms a and b connected via a path P in G^W and for all indices $i \in Indices(P)$ we have $a \cong_i b$, then $a \sim_V b$ holds.*

To produce the full explanation, we need to add equality constraints of the path involved in read-over-weakeq and weakeq-ext.

Definition 6. *Let $Cond(\cdot)$ (resp. $Cond_i(\cdot)$) be the function that takes as input a path P in the weak equivalence graph and computes a condition under which a weak equivalent or weak congruence holds (resp. weak equivalence modulo i holds), defined by induction on P as follows:*

$$\begin{aligned}
&Cond(\emptyset) \ := true && Cond_i(\emptyset) \ := true \\
&Cond((a \leftrightarrow b) \cdot P) \ := (a = b) \wedge Cond(P) && Cond_i((a \leftrightarrow b) \cdot P) \ := (a = b) \wedge Cond_i(P) \\
&Cond((a \overset{j}{\leftrightarrow} b) \cdot P) \ := Cond(P) && Cond_i((a \overset{j}{\leftrightarrow} b) \cdot P) \ := (i \neq j) \wedge Cond_i(P)
\end{aligned}$$

$$Cond(a \approx_i b) \ := Cond_i(P), \text{ where } P \text{ is a path between } a \text{ and } b, \text{ and } \forall j \in Indices(P). i \nsim_V j$$

$$Cond(a \cong_i b) \ := \begin{cases} Cond(a \approx_i b) & \text{if } a \approx_i b \\ Cond(a \approx_i a') \wedge i = j & \text{if } a \approx_i a' \wedge i \sim_V j \\ \wedge read(a', j) = read(b', k) & \wedge read(a', j) \sim_V read(b', k) \\ \wedge k = i \wedge Cond(b' \approx_i b) & \wedge k \sim_V i \wedge b' \approx_i b \end{cases}$$

Algorithm 1: check-read-over-write-conflict(G^W, V, $\sim_V$)

```
for $a, b, i, j \in V$ such that $read(a, i), read(b, j) \in V$ do
    if $a \approx_i b \wedge i \sim_V j \wedge read(a, i) \nsim_V read(b, j)$ then
        return $i = j \wedge Cond(a \approx_i b) \wedge read(a, i) \neq read(b, j)$;
end
return NULL;
```

The algorithm 3 presents a decision procedure based on weakly equivalent arrays reasoning for the extensional theory of arrays. The procedure is sound and complete [16] for the theory.

Algorithm 2: check-ext-conflict(G^W, V, $\sim_V$)

```
for $a, b \in V$ such that $a \nsim_V b$ do
    if there is a path $P$ between $a$ and $b$ such that $\forall i \in Indices(P). a \cong_i b$ then
        return $Cond(P) \wedge \bigwedge_{i \in Indices(P)} Cond(a \cong_i b) \wedge a \neq b$;
end
return NULL;
```

Algorithm 3: arrays-check(G^W, V, $\sim_V$)

```
$conflict$ := check-idx-conflict($G^W$, $V$, $\sim_V$); ;            /* check for idx-lemma conflicts */
if $conflict$ = NULL then $conflict$ := check-read-over-write-conflict($G^W$, $V$, $\sim_V$);
if $conflict$ = NULL then $conflict$ := check-ext-conflict($G^W$, $V$, $\sim_V$);
if $conflict$ = NULL then return (SAT, NULL);
return (UNSAT, $conflict$);
```

3. MCSat Overview

MCSat applies CDCL-like mechanisms to perform theory reasoning. (Figure 1 illustrates the high level flow of the MCSat framework.) The MCSat architecture consists of a *core solver*, an *assingment trail*, and *reasoning plugins*. The core solver explicitly and incrementally constructs models with first-order variable assignments—maintained in the assignment trail—while maintaining the invariant that none of the constraints evaluate to false. It is also responsible for dispatching notifications (e.g. new term notification) and handling requests from the reasoning plugins. The core solver decides upon assignments (values provided by reasoning plugins) when there is choice, it can propagate them when there is not, and it backtracks upon conflict. One of its key roles is to perform conflict analysis when a reasoning plugin detects a conflicting state. The lemmas learned via conflict analysis are based on theory-specific explanations, provided by reasoning plugins, of conflicts and propagations.

When formulas are asserted in MCSat, the core solver notifies all plugins of the asserted formulas. The reasoning plugins analyze the formulas and report all *relevant terms* back to the core. Relevant terms include theory variables and Boolean terms (excluding negations). When computing the value of a compound Boolean term or its negation, relevant terms are the term itself and its closest sub-terms needed for value computation.

The trail is a key data structure in MCSat, holding *relevant term* assignments for easy retrieval of the satisfying assignment upon termination of the MCSat search. The trail functions as a partial model constructed by MCSat during the search process, allowing for term evaluation based on the model values of their relevant subterms. 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 in M. Evaluation-consistency is maintained in the trail, ensuring that no term evaluates to different values within it.

The role of reasoning plugins is to provide assignments for decisions, perform propagations, detect conflicts, and produce explanations. So, to implement a new theory in the MCSat framework, we need a reasoning plugin for that theory that must support in decision-making, propagation (including conflict detection), and explanation generation for propagated terms. Moreover, to ensure termination, there is the *finite-basis* requirement on plugins, i.e. any literals introduced by plugins come from a finite set of literals.

3.1. MCSat Equality and Uninterpreted Functions Plugin

The Equality and Uninterpreted Function (EUF) MCSat-Plugin provides equality reasoning over the uninterpreted sorts and uninterpreted function. The equality reasoning is done by tracking terms of uninterpreted sorts and uninterpreted functions and their trail values in an E-graph [25, 26, 27]. The

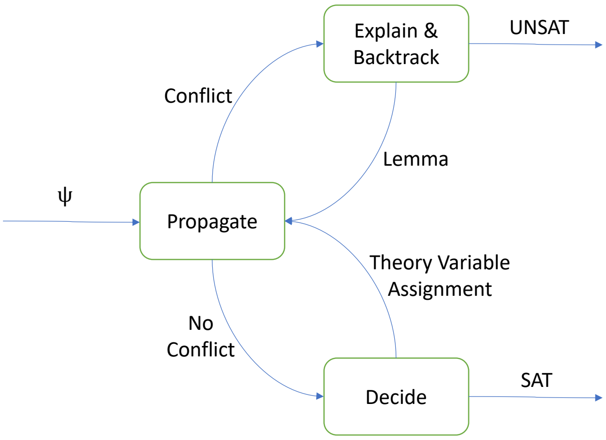


Figure 1: The MCSat framework consists of the following steps: 1) Propagate the trail. 2) If a conflict is found during propagation, check if there is any decision to backtrack over. If not, return UNSAT. Otherwise, explain the conflict using a lemma, backtrack the trail, and repeat step 1. 3) If no conflict is found during propagation, decide on a variable that is not on the trail. If there is nothing left to decide, return SAT. Otherwise, add the decided variable to the trail and repeat step 1.

EUF plugin updates the E-graph when new assignments are made in the trail to the tracked terms. This extended E-graph ensures: 1) If an equivalence class contains an evaluable term c, then the representative of that class is evaluable; 2) Two evaluable terms c_1 and c_2 in the same equivalence class must evaluate to the same value, otherwise this is a source of *conflict*. The conflict is reported to the core MCSat solver as a set of equalities and disequalities that contributed to the conflict.

Propagation and Explanation. The EUF plugin propagates model values of uninterpreted sort terms when two class nodes get merged and one of them is not evaluable. In our implementation we also track equalities between Boolean terms in the E-graph. This allows the EUF plugin to propagate Boolean terms based on equivalence classes. The explanations of these propagations are tracked (or lazily produced), and are provided to the core solver when the propagated terms appear in a conflict when doing conflict analysis and lemma learning. We satisfy the finite-basis requirement because the EUF plugin only "introduces" equalities (disequalities) between terms that already exist in the trail.

Decision Assignment. The EUF plugin is also responsible for providing assignment values for variables and function applications of uninterpreted sorts. A value to uninterpreted sort term is essentially an integer identifier that the MCSat partial model keeps track of. The EUF plugin maintains a set of infeasible values for each term to be decided (terms that EUF is responsible to provide assignment for).

4. Extending MCSat-EUF Plugin with Arrays Reasoning

To be able to do model-based arrays reasoning in MCSat, we extend the EUF plugin with: i) tracking of relevant array terms, ii) array propagation and explanation, iii) array conflict detection, and iv) array decision assignment.

Relevant Array Terms. The extended EUF plugin tracks array terms, in addition to uninterpreted function terms, as relevant terms when the core solver notifies it about new terms. Array terms include array variables, array write terms, and array read terms.[3] The set of relevant terms returned to the core solver may also include terms not present in the term notified by the core solver. These additional terms are array read terms and indices, as defined by the set $V(1)$ in Section 2.

Example 1. *Let* $read(a, i)$, $write(b, j, e)$ *be the new terms notified by the core solver to the extended EUF*

[3] Array reads in Yices2 are uninterpreted function applications, so there was no need to extend the EUF plugin for those terms.

plugin. The relevant array terms set, in this case, is

$$\{write(b, j, e), a, b\} \cup \{read(write(b, j, e), j), read(b, j), j, e\} \cup \{read(a, i), i\}.$$

The main purpose of adding these additional terms is to prompt the MCSat core solver to make decisions on these terms, resulting in assignments to these terms in the trail. This allows us to detect array theory conflicts, for such a trail with assignments, using weakly equivalent arrays reasoning. It is important to note that we are able to satisfy the finite-basis requirement of the extended plugin as weakly equivalent arrays reasoning only uses the tracked terms in the generated arrays lemmas.

Propagation and Explanation. For propagation, we rely on the existing EUF propagation mechanism that propagates model values of terms that are unassigned in the MCSat trail. This covers the read terms and array terms that are evaluable in the E-graph but are not assigned in the MCSat trail. Therefore, the explanation procedure of the existing EUF plugin can be used to explain array term propagations.

Example 2. *Let $M := \{read(a, i) \mapsto \alpha_1, a \mapsto \alpha_2\}$ be an assignment in the solver trail. Suppose the extended EUF plugin deduces, using the E-graph, that the array term b is equal to a and $read(b, j)$ is equal to $read(a, i)$ – note that b and $read(b, j)$ do not have assignments in the trail. The plugin returns the assignment values $\{b \mapsto \alpha_2, read(b, j) \mapsto \alpha_1\}$ as propagations to the core solver, which adds these assignments to the solver trail.*

Array Conflict detection via Weak-Equivalence Graph. To check that the equivalence relation (assignment in the trail and E-graph) satisfies the array theory axioms, we can build the weak equivalence graph for every term in V that has an assignment in the trail. First, we check the (idx) lemma. If violated, we report its negation as a conflict. Otherwise, we check for generalized read-over-write axiom violation using Algorithm 1. We report the conflict detected by the algorithm. Otherwise, we check for generalized extensionality axiom violation using Algorithm 2. Similarly, we report the conflict detected by the algorithm. If no conflicts are detected, then the current MCSat trail satisfies the arrays axioms.

Array Decision Assignment. We use a simple mechanism to make decisions for array variables and array writes. We choose different values for different array terms when making decisions. If two array terms get merged they get equal values via propagation.

4.1. Implementation Details

We have implemented our approach in the MCSat solver of Yices2 [28]. The MCSat implementation in Yices2 already supports real/integer arithmetic [29], bitvectors [30, 27], and uninterpreted functions [27]. Here we provide some important details of our implementation.

Eager Lemma Instantiation. When the core solver notifies the EUF plugin about new write terms, we eagerly instantiate the (idx) axiom for each write and assert the resulting lemma to the solver by adding it to the clause database.

Relevant Array Terms Set. As described in [16], we also optimize our implementation by using a smaller relevant array terms set V. If the element theory is *stably infinite*, we do not need to add $read(a, i)$ for every $write(a, i, e)$. This optimization has the potential to reduce the number of read terms in V, resulting in less work for both Algorithm 1 and Algorithm 2.

Table 1
Different Yices2-MCSat Options Results

	QF_AX (551)		
Solver	solved	sat/unsat	time
Yices2-MCSat	**551**	272/279	177.96
Yices2-MCSat-no-opt-a	**551**	272/279	1244.93
Yices2-MCSat-no-opt-b	550	272/278	276.51
Yices2-MCSat-no-opt-c-1	550	272/278	703.94
Yices2-MCSat-no-opt-c-2	**551**	272/279	228.47

Weak Equivalence Graph Data Structure and Conflict Detection. We utilize the data structure proposed in [16] to store the weak equivalence graph. Every vertex in the graph corresponds to an array term, and an edge represents a write operation between the two vertices – the direction can inverted during the construction of the graph. This data structure offers an efficient way to detect if two arrays are weakly equivalent (as well as modulo i equivalent). For further details, we recommend referring to Section 7 in [16]. The reasoning for weakly equivalent arrays, as presented earlier, assumes that we have a model that satisfies the EUF axioms. It is important to note that the MCSat model is built incrementally as the search progresses, unlike in CDCL($\mathcal{T}$) where the model is built after the reasoning is complete. Therefore, we do not wait till we have a full model that is consistent with the EUF axioms. Instead, we construct the weak equivalence graph as soon as we have a model value for each term in the set V (see (1)). This allows to detect array conflicts earlier than the approach where we wait until the EUF model is complete.

5. Experimental Evaluation

Solvers and Benchmarks. We refer to the implementation of our proposed approach in Yices2 as Yices2-MCSat– we have used the git commit #17369e6. To evaluate its performance, we have done experiments on various quantifier-free logic benchmarks containing arrays from the SMT-LIB [31] release 2023 [32], including QF_AX, QF_ABV, QF_AUFBV, QF_ALIA, QF_AUFLIA, QF_ANIA, and QF_AUFNIA. We have evaluated the different optimizations of our implementation on the QF_AX benchmarks. Moreover, we have compared the optimized version against cvc5 [13] (version 2024-03-25-a40d28f), MathSAT5 [33] (version 5.6.10), and Z3 [14] (version 4.13.0). Yices2 (git commit #17369e6) and Bitwuzla [34] (version 0.4.0) have been also included in the experiments for the supported logics.

Experimental Setup. The experiments were conducted on a 96-core AMD-CPU server running Ubuntu 20.04.6 LTS. We used a time limit of 3 minutes and a memory limit of 8 GB for each benchmark solved by the solvers.

The results are presented in tables 1 to 5. Each table provides information on the logic category, total number of benchmarks in the top row. The solved column shows the number of solved instances, the sat/unsat column shows the number of solved satisfiable/unsatisfiable instances, the time column shows the total solving time for each solver. The aggregated results are also presented in Figure 2, showing cactus plots of different logics.

Evaluation of Optimizations. Table 1 displays the results of evaluating various options of our implementation on the arrays (QF_AX) benchmarks.

The default option, Yices2-MCSat, incorporates all optimizations outlined in Section 4.1. These optimizations include: a) utilizing a smaller relevant array term set, b) eagerly adding the (idx) lemma when the core solver detects a write to the EUF plugin, and c) checking for array conflicts when all relevant array terms have been assigned a value in the trail. Yices-MCSat-no-opt-a and Yices-MCSat-no-opt-b refer to versions without optimization a and optimization b, respectively. Yices-MCSat-no-c-1 and

Table 2
Arrays Benchmarks Results

	QF_AX (551)		
Solver	solved	sat/unsat	time
cvc5	545	272/273	296.29
MathSAT5	**551**	272/279	20.49
Yices2	**551**	272/279	4.01
Yices2-MCSat	**551**	272/279	177.96
Z3	**551**	272/279	28.12

Table 3
Arrays + Nonlinear Arithmetic Benchmarks Results

	QF_ANIA (155)			QF_AUFNIA (17)		
Solver	solved	sat/unsat	time	solved	sat/unsat	time
cvc5	98	89/9	455.56	6	3/3	22.95
MathSAT5	123	116/7	340.79	**17**	5/12	4.75
Yices2-MCSat	**125**	111/14	3406.64	**17**	5/12	7.52
Z3	85	69/16	981.06	**17**	5/12	29.05

Table 4
Arrays + Linear Arithmetic Benchmarks Results

	QF_ALIA (176)			QF_AUFLIA (1303)		
Solver	solved	sat/unsat	time	solved	sat/unsat	time
cvc5	91	22/69	635.24	1286	542/744	609.52
MathSAT5	**160**	88/72	89.08	1300	543/757	177.57
Yices2	**160**	88/72	123.38	**1303**	543/760	29.69
Yices2-MCSat	137	67/70	2046.33	1261	547/714	781.05
Z3	144	73/71	1555.19	**1303**	543/760	20.62

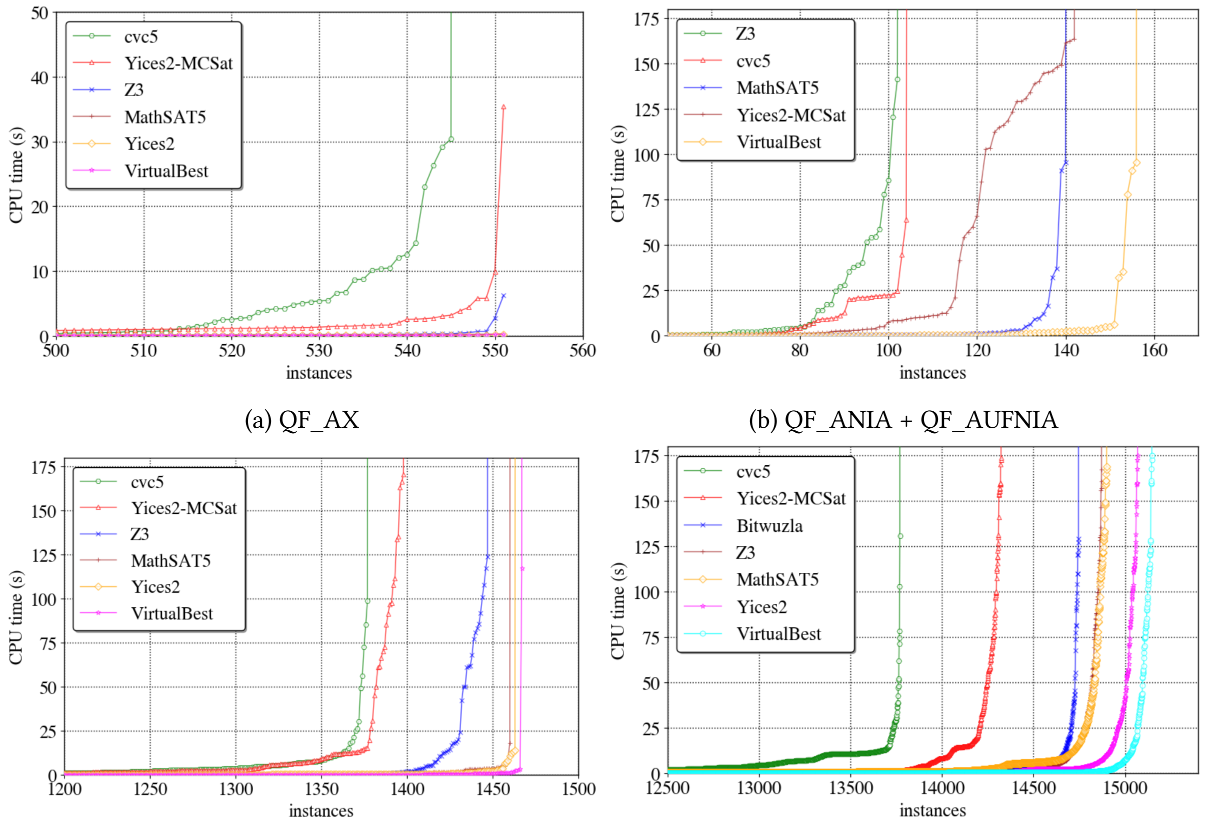
Table 5
Arrays + Bit-vector Benchmarks Results

	QF_ABV (15147)			QF_AUFBV (67)		
Solver	solved	sat/unsat	time	solved	sat/unsat	time
Bitwuzla	14695	10339/4356	4751.64	**52**	14/38	304.20
cvc5	13731	9274/4456	10926.10	41	11/30	355.15
MathSAT5	14902	10321/4580	14205.00	41	11/30	31.11
Yices2	**15018**	10414/4604	10206.00	51	14/37	345.36
Yices2-MCSat	14285	10242/4042	12932.50	40	14/26	839.69
Z3	14827	10259/4568	9092.97	45	11/34	733.71

Yices-MCSat-no-c-2 are versions without optimization c, with the former checking for array conflicts (*early check*) whenever the E-graph does not identify a conflict, and the latter checking for array conflicts (*late check*) when each term in the E-graph has an assignment in the trail.

The results clearly indicate the significance of all optimizations, with optimization a standing out as the most critical. Additionally, we were able to replicate the impact of optimization a, as proposed in [16].

Comparison Againts Other SMT solvers. In Table 2, results for the arrays (QF_AX) benchmarks are shown. Yices2-MCSat solves all the benchmarks and performs competitively with other solvers, even solving more benchmarks than cvc5.



(a) QF_AX

(b) QF_ANIA + QF_AUFNIA

(c) QF_ALIA + QF_AUFLIA

(d) QF_ABV + QF_AUFBV

Figure 2: Cactus Plots of solvers performance on differnt benchmarks: a) Arrays-only, b) Arrays + Nonlinear Arithmetic, c) Arrays + Linear Arithmetic, d) Arrays + Bitvector.

Table 3 displays results for the arrays with nonlinear arithmetic (QF_ANIA and QF_AUFNIA) benchmarks. In this category, Yices2-MCSat demonstrates its strength by solving the highest number of benchmarks compared to other solvers, with a couple of benchmarks lead over MathSAT5.[4] We can notice the complementarity of the various approaches in Figure 2b.

For the arrays with linear arithmetic (QF_ALIA and QF_AUFLIA) benchmarks in Table 4, Yices2-MCSat competes closely with cvc5 and Z3, although it falls behind MathSAT5 and Yices2 in terms of the number of benchmarks solved. The longer solving time for Yices2-MCSat in this category may be due to its use of CAD-based [35] nonlinear reasoning engine even for linear problems.

Table 5 presents the results for the arrays with bitvectors (QF_ABV and QF_AUFBV) benchmarks. Yices2-MCAST's performance is better than cvc5 but not on par with other solvers. Yices2-MCSat uses word-level model-based reasoning for bitvector constraints, which is less efficient than the bit-blasting approach used by the other solvers on SMT-LIB bitvector benchmarks containing bitwise operations. This difference in approach may explain Yices2-MCSat's performance on these benchmarks.

6. Conclusion and Future Work

We have presented a new MCSat-based solver for the extensional theory of arrays. The array decision procedure at its core incorporates EUF and weakly equivalent arrays reasoning. By using the weakly equivalent arrays reasoning, we meet the finite-basis requirement of the MCSat framework. Our approach has been implemented in the Yices2 SMT solver, enabling it to tackle array problems involving nonlinear arithmetic. The performance of this new solver is competitive with the current state of the art and excels in handling array problems with nonlinear arithmetic constraints. Our future plans include

[4] We are not reporting Yices2 (CDCL($\mathcal{T}$)) results here because it does not support nonlinear arithmetic.

expanding the implementation to include *constant arrays* and the *diff* function [19, 36], which will allow us to experiment with generating quantifier-free array interpolants using the model-based interpolation procedure [11] in Yices2-MCSat. Ultimately, we aim to use this to model-check array-based transition systems [37, 38] in the Sally [39, 40] model-checker.

Acknowledgments. We would like to thank Jochen Hoenicke for fruitful discussions about the weakly equivalent arrays that helped us in implementing the reasoning procedure in Yices2. This material is based upon work supported by NSF with award CCF-1816936. 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.

References

[1] A. Armando, M. Benerecetti, J. Mantovani, Abstraction refinement of linear programs with arrays, in: Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, Portugal, March 24-April 1, 2007. Proceedings 13, Springer, 2007, pp. 373–388.

[2] K. R. M. Leino, Dafny: An automatic program verifier for functional correctness, in: International conference on logic for programming artificial intelligence and reasoning, Springer, 2010, pp. 348–370.

[3] A. Cimatti, A. Griggio, S. Tonetta, The VMT-LIB language and tools, arXiv preprint arXiv:2109.12821 (2021).

[4] A. Gurfinkel, T. Kahsai, A. Komuravelli, J. A. Navas, The SeaHorn verification framework, in: International Conference on Computer Aided Verification, Springer, 2015, pp. 343–361.

[5] A. Griggio, M. Jonás, Kratos2: An SMT-based model checker for imperative programs, in: CAV (3), volume 13966 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 423–436.

[6] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, R. Sebastiani, Verilog2SMV: A tool for word-level verification, in: DATE, IEEE, 2016, pp. 1156–1159.

[7] A. Niemetz, M. Preiner, C. Wolf, A. Biere, Btor2 , BtorMC and Boolector 3.0, in: CAV (1), volume 10981 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 587–595.

[8] K. Y. Rozier, R. Dureja, A. Irfan, C. Johannsen, K. Nukala, N. Shankar, C. Tinelli, M. Y. Vardi, MoXI: An intermediate language for symbolic model checking, in: Proceedings of the 30th International Symposium on Model Checking Software (SPIN). LNCS, Springer (April 2024), ????

[9] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T), J. ACM 53 (2006) 937–977.

[10] L. M. de Moura, D. Jovanovic, A model-constructing satisfiability calculus, in: VMCAI, volume 7737 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 1–12.

[11] D. Jovanovic, B. Dutertre, Interpolation and model checking for nonlinear arithmetic, in: CAV (2), volume 12760 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 266–288.

[12] M. P. Bonacina, S. Graham-Lengrand, C. Vauthier, QSMA: A new algorithm for quantified satisfiability modulo theory and assignment, in: B. Pientka, C. Tinelli (Eds.), Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings, volume 14132 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 78–95. URL: https://doi.org/10.1007/978-3-031-38499-8_5. doi:10.1007/978-3-031-38499-8_5.

[13] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, cvc5: A versatile and industrial-strength SMT solver, in: TACAS (1), volume 13243 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 415–442.

[14] L. M. de Moura, N. S. Bjørner, Z3: an efficient SMT solver, in: TACAS, volume 4963 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 337–340.

[15] G. Nelson, D. C. Oppen, Simplification by cooperating decision procedures, ACM Trans. Prog. Lang. Syst. 1 (1979) 245–257.

[16] J. Christ, J. Hoenicke, Weakly equivalent arrays, in: FroCos, volume 9322 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 119–134.

[17] J. Mccarthy, Towards a mathematical science of computation, in: In IFIP Congress, North-Holland, 1962, pp. 21–28.

[18] A. Armando, M. P. Bonacina, S. Ranise, S. Schulz, New results on rewrite-based satisfiability procedures, ACM Trans. Comput. Log. 10 (2009) 4:1–4:51.

[19] R. Bruttomesso, S. Ghilardi, S. Ranise, Quantifier-free interpolation of a theory of arrays, Log. Methods Comput. Sci. 8 (2012).

[20] A. Stump, C. W. Barrett, D. L. Dill, J. R. Levitt, A decision procedure for an extensional theory of arrays, in: LICS, IEEE Computer Society, 2001, pp. 29–37.

[21] A. Goel, S. Krstić, A. Fuchs, Deciding array formulas with frugal axiom instantiation, in: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, 2008, pp. 12–17.

[22] R. Brummayer, A. Biere, Lemmas on demand for the extensional theory of arrays, J. Satisf. Boolean Model. Comput. 6 (2009) 165–201.

[23] L. M. de Moura, N. S. Bjørner, Generalized, efficient array decision procedures, in: FMCAD, IEEE, 2009, pp. 45–52.

[24] J. Christ, J. Hoenicke, A. Nutz, Smtinterpol: An interpolating SMT solver, in: SPIN, volume 7385 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 248–254.

[25] D. Detlefs, G. Nelson, J. B. Saxe, Simplify: a theorem prover for program checking, J. ACM 52 (2005) 365–473.

[26] R. Nieuwenhuis, A. Oliveras, Proof-producing congruence closure, in: RTA, volume 3467 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 453–468.

[27] S. Graham-Lengrand, D. Jovanovic, B. Dutertre, Solving bitvectors with MCSAT: explanations from bits and pieces, in: IJCAR (1), volume 12166 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 103–121.

[28] B. Dutertre, Yices 2.2, in: CAV, volume 8559 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 737–744.

[29] D. Jovanovic, Solving nonlinear integer arithmetic with MCSAT, in: VMCAI, volume 10145 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 330–346.

[30] S. Graham-Lengrand, D. Jovanovic, An MCSAT treatment of bit-vectors, in: SMT, volume 1889 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, pp. 89–100.

[31] C. Barrett, P. Fontaine, C. Tinelli, The Satisfiability Modulo Theories Library (SMT-LIB), `www.SMT-LIB.org`, 2016.

[32] M. Preiner, H.-J. Schurr, C. Barrett, P. Fontaine, A. Niemetz, C. Tinelli, SMT-LIB release 2023 (non-incremental benchmarks), 2024. URL: https://doi.org/10.5281/zenodo.10607722. doi:`10.5281/zenodo.10607722`.

[33] A. Cimatti, A. Griggio, B. J. Schaafsma, R. Sebastiani, The MathSAT5 SMT solver, in: TACAS, volume 7795 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 93–107.

[34] A. Niemetz, M. Preiner, Bitwuzla, in: CAV (2), volume 13965 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 3–17.

[35] G. E. Collins, Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis, SIGSAM Bull. 10 (1976) 10–12.

[36] T. Schindler, SMT solving, interpolation, and quantifiers, Ph.D. thesis, Dissertation, Universität Freiburg, 2022, 2022.

[37] S. Ghilardi, S. Ranise, MCMT: A Model Checker Modulo Theories, in: IJCAR, volume 6173 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 22–29.

[38] M. Mann, A. Irfan, A. Griggio, O. Padon, C. W. Barrett, Counterexample-guided prophecy for model checking modulo the theory of arrays, Log. Methods Comput. Sci. 18 (2022).

[39] B. Dutertre, D. Jovanovic, J. A. Navas, Verification of fault-tolerant protocols with Sally, in: NFM, volume 10811 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 113–120.

[40] D. Jovanovic, B. Dutertre, Property-directed k-induction, in: FMCAD, IEEE, 2016, pp. 85–92.