Partitioning Strategies for Distributed SMT Solving

Amalee Wilson*, Andres Noetzli†, Andrew Reynolds‡, Byron Cook§, Cesare Tinelli‡, and Clark Barrett*
*Stanford University, Stanford, USA {amalee, barrett}@cs.stanford.edu
†Cubist, Inc., San Diego, USA n@cubist.dev
‡The University of Iowa, Iowa City, USA {andrew-reynolds, cesare-tinelli}@uiowa.edu
§Amazon Web Services, Seattle, USA byron@amazon.com

***Abstract*—For many users of Satisfiability Modulo Theories (SMT) solvers, the solver's performance is the main bottleneck in their application. One promising approach for improving performance is to leverage the increasing availability of parallel and cloud computing. However, despite many efforts, the best parallel approach to date consists of running a portfolio of solvers, meaning that performance is still limited by the best possible sequential performance. In this paper, we revisit divide-and-conquer approaches to parallel SMT, in which a challenging problem is partitioned into several subproblems. We introduce several new partitioning strategies and evaluate their performance, both alone as well as within portfolios, on a large set of difficult SMT benchmarks. We show that hybrid portfolios that include our new strategies can significantly outperform traditional portfolios for parallel SMT.**

I. Introduction

State-of-the-art Satisfiability Modulo Theories (SMT) solvers such as Bitwuzla [1], CVC5 [2], MathSAT [3], OpenSMT [4], Yices2 [5], and Z3 [6] are widely used as reasoning engines in the context of verification [7], model checking [8], security [9], synthesis [10], test case generation [11], scheduling [12], and optimization [13].

For many users of SMT solvers, the solver's performance is a bottleneck for their application, and so improving solver performance continues to be a top priority for solver developers. Today, most of the aforementioned solvers remain single-threaded, and performance improvements have primarily been achieved through new solving techniques and heuristics. With the increasing availability of CPUs with large numbers of cores, high-performance computing (HPC), and cloud computing, a natural question is whether these resources together with parallel algorithms for SMT could be used to significantly boost solver performance. Current research in this area can be divided into two main directions: *portfolio solving* and *divide-and-conquer solving*.

Portfolio solving is an approach in which multiple solvers (or different configurations of a solver), attempt to solve the same (or perturbed but equivalent) SMT problem in parallel [14]. It is well-known that SMT solvers are highly sensitive to small perturbations, which can dramatically improve or degrade their performance. While efforts to reduce this sensitivity are an interesting research direction, it is difficult because of the inherent instability of the heuristics used and the uneven nature of the search space. Portfolio solving aims to leverage this sensitivity by sampling from the possible configurations with the hope of finding one that performs well. In fact, this strategy *can* produce significant speed-ups and is currently considered the best way to leverage distributed computing resources to improve performance.

Of course, naive portfolio solving is always limited by the best possible sequential performance, meaning that beyond some point, additional parallel resources do not help. Also, portfolios are empirically ineffective for some classes of benchmarks. For these reasons, it is appealing to pursue the alternative "divide-and-conquer" strategy. In this approach, a problem is partitioned into independent subproblems in such a way that solving the subproblems provides a solution to the original problem. The hope is that because the subproblems have smaller search spaces, solving them in parallel will be faster than solving the original problem. Divide-and-conquer has the potential to dramatically outperform the best sequential performance, but only if an effective partitioning algorithm can be discovered. Such an algorithm has been elusive.

One promising direction is to adapt the *cube-and-conquer* [15] approach that has been successfully applied to the more basic problem of Boolean satisfiability (SAT). In this approach, a partitioning heuristic is used to select a set of n Boolean variables. Typically, a *lookahead* heuristic [16] is used, which chooses variables that, when assigned, most significantly prune the search space. These variables are used to partition the problem into 2^n independent subproblems, which are then solved in parallel. Unfortunately, attempts to adapt this approach for SMT have had limited success. In fact, for some cases, it has been observed that more partitioning is associated with larger, not smaller, runtimes [17].

In this paper, we introduce several new partitioning strate-

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0020347. Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. This work was also supported by an Amazon Research Award and the Stanford Center for Automated Reasoning.

https://doi.org/

gies which build on—but also go beyond—the basic cube-and-conquer approach. In particular, we look at different ways of combining *sources* for collecting atoms (the SMT version of variables) with ways of using those atoms to create different *partition types*. We evaluate an implementation of these strategies in CVC5 on a diverse set of benchmarks from the SMT-COMP cloud track and from previous work on parallel SMT. We show that a portfolio of partitioning strategies outperforms individual strategies, and we introduce the notion of a *graduated* portfolio which performs particularly well. We also show that *hybrid* portfolios combining partitioning and traditional portfolio strategies perform even better. Finally, we show that using a *multijob* scheduling algorithm for the partitioning portfolio accelerates performance even more. We also demonstrate that these approaches scale, i.e., we continue to get additional speedup with more parallelism.

In summary, our contributions are the following:

- the introduction of several novel partitioning strategies for parallel divide-and-conquer SMT solving;
- the introduction of graduated and hybrid partitioning portfolio strategies; and
- an implementation and evaluation of these strategies on a large set of benchmarks, including the first empirical results demonstrating a parallel solving technique that significantly outperforms a traditional non-partitioning portfolio and continues to do so as the number of partitions is increased.

II. Preliminaries

We assume the standard many-sorted first-order logic setting with the usual notions of signature, term, 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. For convenience, we assume a fixed background theory $\mathcal{T}$ with signature Σ including the Boolean sort BOOL. We further assume that all terms are Σ-terms, that entailment ($\models$) is entailment modulo $\mathcal{T}$, equivalence is equivalence modulo $\mathcal{T}$, and that interpretations are $\mathcal{T}$-interpretations. An *atom* is a term of sort BOOL that does not contain any proper sub-terms of sort BOOL. A *literal* is either an atom or the negation of an atom. A *cube* is a conjunction of literals. A formula φ is a term of sort BOOL and is *satisfiable* (resp., *unsatisfiable*) if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A formula whose negation is unsatisfiable is *valid*.

In this paper, we discuss partitioning algorithms that make use of the CDCL(T) framework employed by modern SMT solvers. We give a brief overview of the CDCL(T) framework and introduce the relevant terminology. Then, we describe how partitioning can be used for parallel SMT solving.

CDCL(T)-based SMT solvers solve problems via the cooperation of a SAT solver and one or more theory solvers [18]. The role of the SAT solver in this framework is to build a truth assignment M that satisfies the Boolean abstraction of the problem. Typically, M is built incrementally, and each time the SAT solver assigns a value to an atom, it calls the theory solvers to check whether M is consistent with $\mathcal{T}$. Theory solvers return conflict clauses and optionally new lemmas to the SAT solver. A conflict clause is a valid disjunction of literals that is falsified by M, and a lemma is any other heuristically-chosen valid formula. When lemmas and conflict clauses are received by the SAT solver, they are added to the original problem. The process of finding satisfying assignments is repeated until one of two outcomes is achieved: either M is a complete SAT assignment and no conflicts are detected by the theory solvers, meaning the problem is satisfiable; or, an unrecoverable conflict is derived, and the problem is therefore unsatisfiable.

Parallel SMT Solving with Partitioning The satisfiability of a formula ϕ can be determined in parallel by dividing it into n independent *subproblems* ϕ_1, ..., ϕ_n. Provided the disjunction $\phi_1 \vee \cdots \vee \phi_n$ is equisatisfiable with ϕ, if any of the subproblems are satisfiable, then the original problem is satisfiable, and if all of the subproblems are unsatisfiable, then the original problem is unsatisfiable. In this simple scenario, no synchronization is necessary during solving, because the subproblems are independent.

A *partitioning strategy* constructs subproblems ϕ_i of the form $\phi \wedge p_i$. We call each p_i a *partitioning formula* and refer to each subproblem as a *partition*. Though not required for correctness, it is generally desirable for the partitions to be disjoint (i.e., for each $i \neq j$, the formula $\phi \wedge p_i \wedge p_j$ is unsatisfiable) to avoid performing duplicate work. In the cube-and-conquer partitioning strategy, a set of N atoms is selected, and each of the 2^N possible cubes using these atoms is used as a partitioning formula, resulting in 2^N partitions. Scattering [19] is an alternative strategy which differs from cube-and-conquer in that it creates partitioning formulas that are not cubes. Instead, scattering produces a series of N partitioning formulas as follows. The first partitioning formula is some cube C_1. The second is $\neg(C_1) \wedge C_2$ for some new cube C_2. The next is $\neg C_1 \wedge \neg C_2 \wedge C_3$ for a new cube C_3, and so on. The N^{th} partitioning formula is simply $\neg C_1 \wedge \cdots \wedge \neg C_{N-1}$. Note that by construction, the partitioning formulas are disjoint. However, there is considerable freedom in how the cubes are chosen.

III. Related Work

As mentioned, much of the existing research literature on parallel and distributed SMT solving focuses on *portfolios*. A portfolio consists of multiple solver instances running in parallel, each of which attempts to solve the same problem. The instance that finishes first produces the result and ends the portfolio run. Each instance differs from the others in some way: a different solver or configuration is used, or the problem has been perturbed in some equisatisfiable way. Some portfolio frameworks enhance this basic strategy by sharing information (e.g., learned lemmas or clauses) among the solver instances running in parallel. Z3 was one of the first solvers to support portfolio solving with sharing [14]. SMTS [20] is a parallel framework that also supports portfolio solving with sharing [21]. In fact, SMTS implements

the parallelization tree formalism [22] [23], which involves recursively combining both partitioning and portfolio solving. Our work mainly explores partitioning. We focus on finding specific effective partitioning strategies, which could then be integrated into a framework such as SMTS. Interestingly, the two approaches (portfolio solving and partitioning) *can* be effectively combined as observed in [20] and as we discuss in Section V. Note that we do not (yet) consider information sharing, as this would add another layer of complexity, and there is enough to understand without it.

There is also previous work on partitioning strategies. *Cube-and-conquer* [15] and other types of splitting [24] have been successfully applied to SAT problems, but there is no consensus on how these approaches should be lifted to the SMT context. OpenSMT2 supports two different lookahead strategies for creating cubes in a cube-and-conquer-like partitioning strategy [17]. One is based on the global number of free atoms, and the other is based on the number of unassigned atoms in the clauses. Previous evaluations using these strategies were mixed, with the strategies performing well on some benchmarks but not on others. OpenSMT2 also supports a *scattering* strategy that was originally developed for solving SAT problems [19]. In their implementation, each cube is obtained by taking atoms from the decisions made up along a particular search branch during a run of OpenSMT2. The number of literals in each cube varies according to a heuristic. Details can be found in [19], [21], [23]. When the scattering partitioning strategy was compared to portfolio solving in [21], they found that portfolio performed better on quantifier-free linear real arithmetic (QF_LRA) benchmarks, especially on unsat problems. We compare all of the OpenSMT2 strategies with our own strategies in our evaluation.

PBoolector [25], a parallel SMT solver built on top of the Boolector SMT solver [26], uses a cube-and-conquer style strategy for QF_BV SMT formulas, with the goal of evaluating whether lookahead methods work well in combination with term-rewriting rules and bit-blasting techniques. On quantifier-free bitvector (QF_BV) benchmarks, PBoolector saw familiar results: each configuration of their solver performed well on a subset of the benchmarks while performing poorly on others. Because of our focus on partitioning strategies rather than implementations and the similarity of its partitioning strategy (lookahead-based) to that of OpenSMT2, we do not directly compare with PBoolector.

Previous work on partitioning has also been limited in terms of which SMT-LIB logics were supported: benchmarks over quantifier-free uninterpreted functions (QF_UF) were used in [23], QF_BV benchmarks in [25], and both QF_UF and QF_LRA benchmarks in [17], [20], [21]. We are the first to implement a general-purpose partitioning strategy that works for all SMT-LIB logics. We comment on which types of problems are well-suited for our partitioning algorithms in Section V.

IV. Partitioning

In this section, we introduce a set of partitioning strategies parameterized in four dimensions: atom source, selection heuristic, partition type, and partition timing. Pseudocode for partitioning based on these parameters is given in Algorithm 1. More details on these parameters and their relationship to Algorithm 1 are given in the following subsections, but briefly, the atom source and selection heuristic specify *what* the partitions are made of, the partition type specifies *how* the partitions are made, and the partition timing specifies *when* the partitions are made.

In all cases, partitions are made by invoking an instrumented version of an SMT solver that calls Algorithm 1 periodically during the solving process. We call this solver the *partitioning solver*. By instrumenting an existing solver, our approach takes advantage of well-tested infrastructure for parsing, preprocessing, and reasoning about SMT problems. The first step in Algorithm 1 is to check whether, according to the *partition timing* heuristic (see Section IV-C), it is the right time to make a partition. If not, nothing is done. Otherwise, a set of atoms is collected from the specified source (see Section IV-A). If an insufficient number of atoms is collected, again, nothing is done. Otherwise, $makePartitions$ is called. Note that the behavior of $makePartitions$ depends on the *partition type* (see Section IV-B). If $ptype$ is CUBE, then $makePartitions$ will emit all partitioning formulas and return true, and Algorithm 1 will not be called again. If $ptype$ is SCATTER, then as long as the number of partitinos generated so far is less than $N-2$, $makePartitions$ creates a single partitioning formula and returns false. In this case, the partitioning solver also blocks the part of the search space corresponding to the generated partitioning formula by adding the negation of the cube part of the partitioning formula as a lemma (called a *blocking lemma*). The partitioning solver then continues to work on solving the problem until its next call to Algorithm 1. If $ptype$ is SCATTER and the number of partitions generated is $N-2$, then $makePartitions$ creates the last 2 partitions (as described in Sec. IV-B, below) and returns true.

It is possible that the partitioning solver actually solves the problem. If the solver determines that the problem is satisfiable, or if it finds that the problem is unsatisfiable before any partitions have been made, then the problem has been solved, and there is no need to continue or to solve any partitioned formulas. However, if the partitioning solver returns unsatisfiable after having emitted some partitions, then these partitions still need to be solved. This is because of the blocking lemmas that were added which prune the parts of the search space (in the partitioning solver) covered by the emitted partitions.

A. Atom Source and Selection Heuristics

There are two parameters for atom selection: $atomSource$ and $atomSelHeur$. The $atomSource$ parameter describes where the atoms should come from. We explore three different sources of atoms for our partitioning strategies: the SAT heap, which is a priority queue of Boolean variables in the SAT

Algorithm 1 Partitioning strategy pseudocode.

Input: $N \geq 2$, $cubeSize = \log_2(N)$
Input: $atomSource \in$ {HEAP, DECISION, CL}
Input: $atomSelHeur \in$ {RAND, SPEC}
Input: $ptype \in$ {CUBE, SCATTER}
Input: $t_1, t_2 \geq 0, tHeur \in$ {CHECK, TIME}
Output: returns true iff done partitioning

```
1:  timeToPart ← isTimeToPartition(t1, t2, tHeur)
2:  if timeToPart then
3:      atoms ← collectAtoms(atomSource, atomSelHeur)
4:      if atoms.size() ≥ cubeSize then
5:          atoms.resize(cubeSize)
6:          if makePartitions(ptype, atoms, N) then
7:              return true
8:          end if
9:      end if
10: end if
11: return false
```

solver; the decision trail of the SAT solver, which contains the decisions made by the SAT solver along its current search branch; and conflict-or-lemma (CL) atoms, which are atoms contained in the lemmas and conflict clauses sent from the theory solver to the SAT solver. These sources of atoms correspond to HEAP, DECISION, and CL in Algorithm 1, respectively. The $atomSelHeur$ parameter describes how atoms should be selected from the available atoms in the source. Every source supports selecting atoms at random (RAND in Algorithm 1). The other option is to use a heuristic specific to the source (SPEC). We describe these below. To guarantee that partitions can be made quickly, each of the heuristics is lightweight and does not rely on sophisticated or computationally expensive score calculations.

As mentioned above, when $atomSource$ is HEAP, the source of atoms is an internal data structure in the SAT solver. Typically, SAT solvers make decisions based on the *activity* score of each variable. The activity of a variable is determined by how often it appears in conflicts [27], and the variable with the highest score is used when the SAT solver makes decisions. When using HEAP as its source, the SPEC heuristic simply chooses the $cubeSize$ variables with the highest activity scores. The rationale is that highly active variables may be a good choice for helping shape partitions. Note that this heuristic requires no additional computation by the partitioning algorithm because the SAT solver already orders the variables by their activity.

Variables that are good for SAT decisions may not always be ideal for SMT partitions. For example, some high-activity variables may be closely related to other high-activity variables, because they represent theory atoms that contain similar terms. Ideally, however, variables used in partitions should be as *independent* as possible, so that each partition has roughly the same difficulty. The DECISION option attempts to address this weakness. It uses the decision trail as a source of atoms. The decision trail contains all the variables that have been decided on along a particular branch of the search tree during the solving process. The rationale is that variables in the decision trail are, roughly speaking, more likely to be independent (for example, if two variables entail each other, then deciding on one will always propagate the other, so they cannot both be in the decision trail at the same time). When selecting atoms from the decision trail, the SPEC heuristic chooses the earliest decisions in the trail. As before, this heuristic requires no additional computation by the partitioning algorithm, because the decisions are stored in the trail from least to most recent.

Finally, the CL option uses conflict clauses and lemmas coming from theory solvers as the atom source. Intuitively, atoms that appear in conflict clauses and lemmas are those that are "contributing" in some way to the solution process. While the appearance of an atom in a conflict clause is reflected in its activity score, an appearance in a lemma has no effect on the activity score. This is because when lemmas are generated, they are simply added as additional clauses. The role of a lemma is to help guide the search in a theory-specific way. Thus, we expect that atoms appearing in lemmas may be important. When selecting atoms from conflict clauses and lemmas, the SPEC heuristic selects those atoms that occur most frequently. These atoms are tracked as they are sent from the theory solver to the SAT solver, and a counter is maintained for each atom.

There are two additional issues to consider when selecting atoms: the number of atoms to use for each partition and filtering out unusable atoms. In Algorithm 1, we fix the number of atoms per partition, $cubeSize$, to be $log_2(N)$ where N is the number of requested partitions. However, as we discuss below, for the SCATTER partition type, this is not necessary and $cubeSize$ could be set as an additional parameter. Regarding filtering, because the construction of partitions is done by appending a partitioning formula to the original formula, anything appearing in the partitioning formula must make sense in this context. In particular, if an atom contains some symbol generated internally by the solver (i.e., not appearing in the original problem), we filter that atom out. This filtering is done during the $collectAtoms$ routine.

B. Partition Type

We consider two ways of creating partitioning formulas from the selected atoms. The first, the CUBE partition type, follows the *cube-and-conquer* approach. The second, the SCATTER partition type, uses a scattering strategy.

1) Cubes: The cube strategy requires $log_2(N)$ atoms to be selected during the atom selection phase. Once the required number of atoms has been collected, they are immediately used to create N mutually exclusive cubes. The partitioning solver then terminates. These cubes, C_1 through C_{2^n}, correspond to each possible conjunction of atoms that can be created from the selected atoms. For example, if the two atoms selected are x_1 and x_2, then the following partitions would be emitted: $C_1 = x_1 \wedge x_2$, $C_2 = \neg x_1 \wedge x_2$, $C_3 = x_1 \wedge \neg x_2$, and $C_4 = \neg x_1 \wedge \neg x_2$.

2) Scattering: Scattering is a dynamic strategy for creating partitioning formulas. When the partition type is SCATTER, Algorithm 1 produces only a single partition with each call to $makePartitions$. The partitioning formula constructed at each call takes the current cube and conjoins it with the negation of previous cubes, as described in Section II. After each generated partition, the negation of the partitioning formula is added as a lemma to the partitioning solver, to ensure that it explores a different part of the search space during the rest of its run.

Note that $cubeSize$ does not have to be equal to $\log_2(N)$ when using scattering. Indeed, it can even vary from partition to partition. In [23], a particular strategy is suggested that does vary the $cubeSize$ across partitions. In this paper, we simply use $\log_2(N)$, for the $cubeSize$.

Note that it takes at least $N-1$ calls to Algorithm 1 to compute N partitions with the SCATTER partition type. The final partition is emitted immediately after the $N-1^{st}$ partition, because the final partition is simply the negation of all previously used cubes.

C. Partition Timing

Partition timing specifies *when* partitions should be made. There is a trade-off between collecting sufficient information to create good partitions and avoiding spending unnecessary time on partitioning that could have been used for solving. Algorithm 1 supports two different kinds of partition timing. The first, when $tHeur$ is CHECK, simply counts the number of times that Algorithm 1 has been called (in our implementation, this is done during the *check* phase as we explain in Section V below). The second possibility is TIME, which simply measures the amount of time that has passed. In Algorithm 1, two inputs, t_1 and t_2, are used for timing. The t_1 parameter determines how long to wait (either number of checks or time in seconds) before the partitioning solver creates any partitions. The idea of this parameter is to allow the partitioning solver to make some progress and get into an interesting state before starting partitioning. The t_2 parameter determines how long (again in either checks or seconds) to wait *between* each pair of partitions. Note that for the CUBE partition type, t_2 is irrelevant, because all partitions are created at once.

There is another trade-off between predictability of partitioning time and predictability of partitioning formulas. When counting checks, the partitioning formulas are deterministic (as long as the execution of the SMT solver is deterministic). When using time, however, there can be variation in the current state of the solver at time t from one run the to next. Thus, it may seem like check counting is preferable. The problem is that the number of checks varies greatly from one problem to the next. One SMT formula may trigger thousands of checks in the solver in the first minute of solving, while other SMT formulas have only a handful. Thus, for predictable partitioning time (though less predictable partitions), it can be better to use time instead of check counting to help ensure that the time to create partitions is relatively stable across many problems. We discuss this further in Section V-A, below.

V. Evaluation of Partitioning Strategies

We instrumented CVC5 to be a partitioning solver by (i) implementing Algorithm 1 in CVC5; and (ii) having CVC5 call Algorithm 1 after each decision made by its internal SAT solver. Below, we report on several sets of experiments with this instrumented version of CVC5. We ran all experiments reported here on a cluster with 26 nodes running Ubuntu 20.04 LTS, each with 128 GB of RAM, and two Intel Xeon CPU E5-2620 v4 CPUs with 8 cores per CPU. Although both CVC5 and OpenSMT2 are used as partitioning solvers, all partitions, scrambles, and original formulas are solved using CVC5.

In our evaluation, we compare several configurations of our CVC5-based partitioning solver and an OpenSMT2-based partitioning solver on a set of benchmarks drawn from the cloud track of the 2022 edition of SMT-COMP [28], the SMT-LIB QF_LRA benchmarks, and the SMT-LIB QF_UF benchmarks [29]. The SMT-COMP cloud benchmarks were selected by the SMT-COMP organizers to be challenge problems for parallel solving and are thus a good target for this work. The QF_LRA and QF_UF benchmarks have been the subject of previous studies on parallel solving [17], [23], [21].

We exclude benchmarks based on several criteria. First, we exclude any benchmark with quantifiers. The challenges for quantified benchmarks are typically the result of too many possible quantifier instantiations. In contrast, partitioning targets challenges stemming from large Boolean search spaces. Second, we exclude benchmarks that are solved in less than 600 seconds by the sequential version of CVC5. This is simply to focus on problems that are challenging for sequential solvers. Third, if no partitions can be made by the partitioning solver, the benchmark is excluded. This can happen if the SAT solver makes no or almost no decisions and Algorithm 1 is not called enough times to create partitions. We consider such benchmarks poor candidates for solving via partitioning. Finally, benchmarks that are solved during partitioning by any partitioning solver are excluded, again because we consider them easy, and furthermore, our aim is to compare partitioning strategies, not partitioning solvers. One way that a problem can be solved during partitioning if it is trivial and is solved before any calls to $makePartitions$ in Algorithm 1. The other way a problem can be solved during partitioning can only happen if the SCATTER partitioning strategy is used *and* the problem is satisfiable. In this case, it is possible that after some number of calls to $makePartitions$, each of which blocks some part of the search space, the partitioning solver stumbles upon a satisfying solution before the next call to $makePartitions$. After these filters are applied, we are left with 214 challenging benchmarks in 5 SMT-LIB logics: QF_LRA (139), QF_IDL (48), QF_LIA (16), QF_UF (7), and QF_RDL (4).

To measure the performance of a particular partitioning strategy on a given benchmark, we first measure the time to partition it. Then, we run each partition on the cluster (with a maximum of 16 jobs per node, i.e., one per core, and 8 GB

TABLE I
EFFECT OF WAIT TIMES ON CUBING OF 125 QF_LRA BENCHMARKS

Time	Solved	PAR-2
1s	40	223636
3s	42	219187
15s	42	219313

of memory per job) using the CVC5 SMT-COMP 2022 script (which runs a theory-dependent sequential instance of CVC5) and record the time. Then, we record the total solving time for the benchmark as the sum of the partitioning time and either the maximum time required to solve any of its partitions, if the benchmark is unsatisfiable, or the minimum runtime of the satisfiable partitions, if the benchmark is satisfiable. Note that this simulates a parallel run in which all of the partitions are run simultaneously on separate cores.

To evaluate a partitioning strategy on the set of benchmarks as a whole, we use the PAR-2 score [30], which is also used for scoring the annual SAT competition. The PAR-2 score is the sum of the runtimes for all solved instances plus *twice* the timeout value multiplied by the number of unsolved instances. The lower the PAR-2 score is, the better. We use the PAR-2 score because it provides a single metric that incorporates both runtime and number of benchmarks solved. In this work, we are primarily interested in the scalability of the different partitioning strategies, so most experiments measure the PAR-2 score for different numbers of partitions. We use a 20 minute timeout when solving partitions, which includes both the time to partition and the time to solve.

In the following subsections we explore a broad design space and, in the spirit of transparency, we chose to share the outcomes of both successful and less successful partitioning strategies. We chose to share as much as possible in an effort to both document our process and enrich future research in this area. When first developing our partitioning strategies, we tested them on 23 problems, a little more than 10% of the total problems in our set of challenge benchmarks. It was not until after extensive testing on this smaller subset that we ran the full set of 214 benchmarks with our various strategies to produce the data for the graphs that follow.

A. Partitioning Strategies

We first explore the different partitioning strategies in the parameter space of Algorithm 1, starting with the partition timing parameter.

First, we measure the impact of t_1 when using TIME for $tHeur$. In general, we find that if the value of t_1 is too small, then the partitioning solver does not have enough time to provide good atoms, but beyond a threshold there is little additional benefit. To demonstrate this, Table I shows a set of results using different values of t_1. For these runs, we set $atomSource$ to HEAP, $ptype$ to CUBE, and $atomSelHeur$ to SPEC, and run on a subset of benchmarks consisting of 125 QF_LRA benchmarks. We see that with $t_1 = 3$, we are able to solve more problems and obtain a better PAR-2 score than with $t_1 = 1$. At the same time, increasing t_1 to 15 does not result in any additional problems being solved and has a small negative effect on the PAR-2 score. Additional experiments (not shown here) using other parameter settings and benchmarks show similar results. Based on these observations, we use $t_1 = 3$ when using TIME.

Next, we compare $tHeur$ = CHECK and $tHeur$ = TIME. For these experiments, we use $ptype$ = SCATTER (since otherwise t_2 plays no role) and $atomSelHeur$ = SPEC. For $tHeur$ = TIME, we use $t_1 = 3$ and $t_2 = 0.1$, so the first call to Algorithm 1 is after three seconds, and each subsequent call is after an additional tenth of a second. For the CHECK runs, we use $t_1 = 1$ and $t_2 = 1$, so the first call is after the first check, and each subsequent call is after the next check. Figure 1 shows the results[1] of our experiments for different atom sources and partition sizes. This figure shows that when $atomSource$ is DECISION or HEAP, TIME clearly outperforms CHECK. With CL, the results are not conclusive, with both strategies performing similarly.

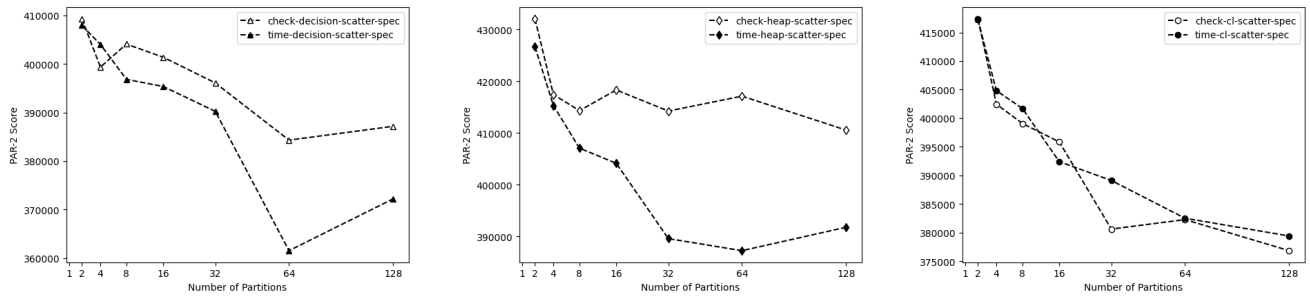
One likely reason for the poor performance of CHECK in the first two cases is that with $t_1 = 1$ we start partitioning on the first call to Algorithm 1. As we mentioned above, it is usually better to wait some time before partitioning. However, there is a tremendous amount of variation in the number of calls to Algorithm 1 across different benchmarks. This makes it difficult to find values other than 1 for t_1 and t_2 that work consistently across all benchmarks. On the other hand, using TIME with $t_1 = 3$ and $t_2 = .1$ tends to perform well across different benchmarks and parameters. For these reasons, we use these parameters in the remaining experiments.

It is worth noting that the disadvantage of nondeterminism when using TIME remains. In future work, we hope to find a way to get the consistency of TIME while also having deterministic results.

We now turn our attention to the other parameters of Algorithm 1. Figure 2 shows the six possible strategies using $atomSelHeur$ = SPEC compared with a random strategy (TIME-HEAP-CUBE-SPEC) as a baseline (other random strategies are similar). The HEAP strategies consistently perform worse than the random strategy, which suggests that relying on the SAT solver's priority queue heuristic is not particularly effective for making partitions.

The DECISION strategies, on the other hand, both perform quite well, beating the random strategy most of the time. This suggests that selecting the earliest decisions from the decision trail is a useful heuristic when selecting atoms for partitioning. The CL strategies are mixed. The CUBE variant performs favorably compared to random, while the SCATTER variant generally does not. Based on these results, we choose three configurations as our top partitioning strategies, and refer to them in the following as decision-cube (TIME-DECISION-CUBE-SPEC), decision-scatter (TIME-DECISION-SCATTER-SPEC), and cl-cube (TIME-CL-CUBE-SPEC).

[1]These and subsequent experiments are on the full set of 214 benchmarks.



(a) Decision Scattering (b) Heap Scattering (c) CL Scattering

Fig. 1. Comparison of using TIME vs CHECK for $tHeur$ when scattering.

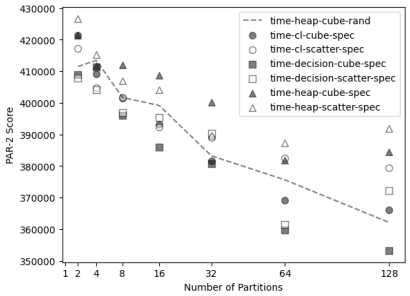


Fig. 2. Various partitioning strategies vs a random strategy.

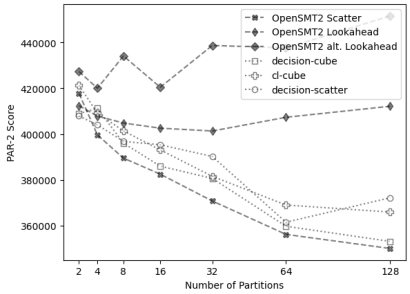


Fig. 3. CVC5 partitioning strategies vs OpenSMT2 partitioning strategies

Clearly, there are many more possible strategies and variants that could be explored. We expect this to be a fruitful direction for future work.

B. Comparison to OpenSMT2 Partitioning Strategies

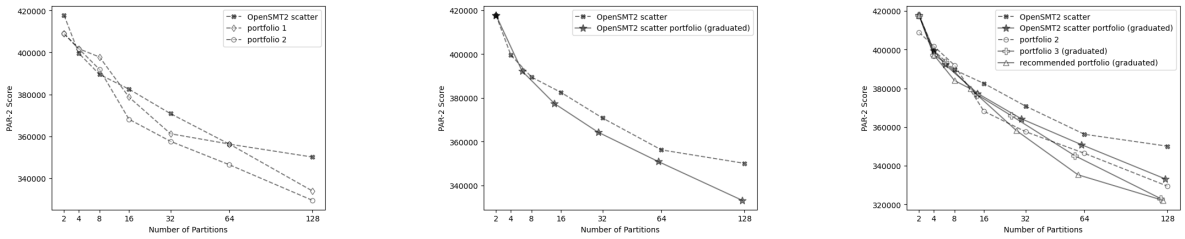
The most extensive studies on partitioning strategies in previous work are from the OpenSMT2 team. We next compare our best partitioning strategies, decision-cube, cl-cube, and decision-scatter, to the three partitioning strategies available in OpenSMT2. Recall that none of the selected benchmarks are solved during partitioning, so this is a comparison only of the partitioning strategies, not the solvers. Figure 3 shows the results of this comparison. The two lookahead partitioning strategies in OpenSMT2 perform worse than our three best strategies. This is partly because they are slow and often time out during partitioning. On the other hand, the OpenSMT2 scattering strategy performs very well. In particular, their scattering strategy outperforms our individual partitioning strategies, though decision-cube is quite close. Interestingly, the OpenSMT2 scattering strategy also uses decisions as its source of atoms, making it very similar to our decision-scatter strategy. However, they have a few additional parameters that have been fine-tuned. For example, they vary the number of literals per partitioning formula, something that our strategy does not do. This suggests that our decision-scatter strategy could likely be improved in a similar way. In the spirit of "if you can't beat them, join them," we replace our decision-scatter strategy with the OpenSMT2 scattering strategy in our list of top performing strategies going forward and refer to it as osmt-scatter.

VI. Portfolios of Partitioning Strategies

One consistent observation in this and previous work is that there is a lot of variation in how well strategies work across benchmarks. Every strategy fails on some benchmarks and works well on others. Given the success of portfolio solving in general, a natural question is whether a portfolio of partitioning strategies can outperform individual partitioning strategies. It is not immediately obvious whether this should be true, since, to be fair, we require that partitioning portfolios divide up their partitioning budget. For example, we would compare a partitioning portfolio using 2 strategies, each creating 16 partitions to an individual strategy that can use 32 partitions. We explore four different types of portfolios, each with a common goal of maximizing the diversity of solving strategies while minimizing the number of solver instances.

A. Types of Portfolios of Partitioning Strategies

The first type of portfolio is a *partitioning portfolio* where multiple strategies are used to create the same number of partitions as a single strategy. The goal of using this type of portfolio is to allow for different partitioning strategies to compensate for the weaknesses of the other strategies included in the partitioning portfolio. Figure 4a shows the results of using two different partitioning portfolios and compares them to the best individual strategy, osmt-scatter. Portfolio 1 creates $N/2$ partitions with each of decision-cube and cl-cube,



(a) Partitioning Portfolios (b) Graduated Portfolio (c) Graduated Portfolio Comparisons

Fig. 4. Comparison of various partitioning portfolios.

and the results demonstrate that a portfolio of two (worse) strategies can do better than the best individual strategy. Portfolio 2 creates $N/2$ partitions with decision-cube, $N/4$ partitions with cl-cube, and $N/4$ partitions with osmt-scatter and demonstrates that adding more variety in the portfolio can result in even better performance. These results suggest that the orthogonality of the individual strategies is high and that adding more strategies at the expense of the number of partitions can be beneficial.

The next type of portfolio is a *graduated portfolio* which takes the above idea of maximizing diversity a bit further. Since diverse strategies are helpful, we can also diversify within a single strategy by instantiating that strategy several times, each time using a different number of partitions. To construct a graduated portfolio with a single partitioning strategy, the strategy is used to create 2 partitions, then it is used to create 4 partitions, and so on until the total desired number of partitions is nearly met (powers of 2 may not perfectly add up to the total number of desired partitions) but never exceeded. To visualize the value of graduated partitioning, Figure 4b compares the stand-alone osmt-scatter strategy with a graduated partitioning portfolio version of the same strategy. To see the difference, note that for $N = 16$, the stand-alone strategy runs osmt-scatter once to make 16 partitions, whereas the graduated portfolio runs osmt-scatter three times, making 2, 4, and 8 partitions, respectively. Notice that although the graduated portfolio uses two fewer partitions for every plotted value of N, it clearly outperforms the stand-alone strategy, especially as the number of partitions is increased. Strategies with the fewest partitions require the least resources, so in some sense, they are providing the greatest diversity at the lowest cost. Thus, we use graduated portfolios to achieve the most diversity at the lowest cost.

Notice that the above two strategies can be combined into a third type of portfolio called a *graduated partitioning portfolio*. A graduated partitioning portfolio based on m individual partitioning strategies is constructed as follows. First, take each of the m individual strategies and parameterize them by N for values of N that are powers of 2, e.g., osmt-scatter-2 is the strategy that uses osmt-scatter to make 2 partitions, and cl-cube-64 is the strategy that uses cl-cube to make 64 partitions. Now, rank all of the strategies, with smaller values of N being ranked higher. To break ties when $m > 1$, use a ranking on individual strategies (for our top performing strategies, we rank osmt-scatter-N higher than decision-cube-N higher than cl-cube-N). Finally, to obtain the graduated portfolio strategy for N partitions, simply collect strategies from this list, in order, until it is no longer possible to add strategies without exceeding N. For example, for $N = 32$ and $m = 3$, we would choose all of the 2-partition strategies, all of the 4-partition strategies, and the osmt-scatter-8 strategy, for a total of 26 partitions. We do not attempt to use the remaining 6 partitions in our "partition budget."

We experimented with all possible combinations of $m = 1, 2, 3$ and our top performing strategies. Figure 4c shows selected results (and also the best strategies from Figures 3, 4a and 4b for comparison). Portfolio 3 is the graduated partitioning portfolio with $m = 3$ using all three top-performing strategies: osmt-scatter, decision-cube, and cl-cube. Portfolio 3 outperforms portfolio 2 for large numbers of cores. However, the strongest portfolio uses $m = 2$ and only combines osmt-scatter and decision-cube (in this case, it appears that the diversity of cl-cube does not compensate for the lack of additional versions of the other two strategies). We refer to this as the "recommended portfolio." Our recommended portfolio strategy has consistently better performance than all other strategies we have considered, even though it uses fewer partitions than the other strategies.

The fourth and final type of portfolio we refer to as a *hybrid portfolio*. This type of portfolio combines a traditional portfolio and a portfolio of partitioning strategies. For this approach, given N cores, we simply run a recommended portfolio of size $N/2$ and a traditional portfolio with the remaining cores. Once again, the goal is to diversify the solving approaches while keeping the number of solver instances as low as possible. Results for this approach are discussed in the following section.

B. Comparison to a Traditional Portfolio

Finally, we conclude by trying to address a very practical question. Given our results, how should one go about using N-way parallelism to solve a challenging SMT problem? In particular, how do the best partitioning approaches compare with traditional portfolio approaches. To represent the latter, we use a *scrambling* portfolio. Given a problem and a value of N, we construct a scrambling portfolio of size N by running N versions of the problem: the original problem plus $N - 1$

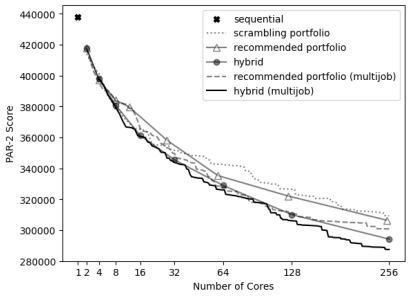


Fig. 5. Comparison of different portfolios. Hybrid and recommended portfolios are graduated.

TABLE II
NUMBER OF PROBLEMS SOLVED WITH 256 CORES

Strategy	Solved (out of 214)
sequential	54
scrambling portfolio	107
recommended portfolio	109
hybrid	111
recommended portfolio (multijob)	112
hybrid (multijob)	114

copies obtained using the SMT-COMP scrambler [31] with different random seeds. The runtime for the scrambling portfolio on the problem is then the minimum of these runtimes (we include the time required to scramble in the individual runtimes, but this is typically negligible).

Figure 5 compares, for different values of N, a scrambling portfolio with our recommended portfolio, based on a simulation where each partition or scramble is run on a separate core in parallel (it also includes the performance of a single sequential run for comparison). We see that the scrambling portfolio does indeed outperform the recommended partitioning portfolio for $N < 32$, but the recommended portfolio wins as the number of cores gets larger.

The result of using the hybrid portfolio is shown as "hybrid" in Figure 5. Recall that for this approach, half the cores are used to run a recommended portfolio while the other half run a scrambling portfolio. Remarkably, the hybrid strategy clearly outperforms the scrambling portfolio, even for small numbers of cores. We note that, to our knowledge, this is the first time any parallel strategy has been shown to be consistently superior to a traditional portfolio.

Finally, there is one more optimization that we can apply: we can use the *multijob* strategy referenced in the original cube-and-conquer paper [15]. The key idea is that because partitions can be very uneven (in runtime), using one core per partition means that many cores (the ones that get assigned easy partitions) will be idle most of the time. The multijob strategy simply consists of using *many more partitions* than cores, and then scheduling the partitions as cores become available. This automatically load-balances the partitions and results in being able to include the results from more partitions without much additional (wall-clock) runtime. This optimization always improves the performance of partitioning approaches, but it has no effect on traditional portfolios since a traditional portfolio cannot benefit from load-balancing. Consider a problem that is unsolved by the (non-multijob) hybrid strategy. For this problem to be unsolved, all scrambling jobs must time out and at least one of the partitions for each partitioning strategy must also time out. We observe that, in practice, many of the partitions finish quickly, meaning that the resources allocated to those partitions are idle while the other jobs run to timeout. The multijob strategy simply uses these additional available resources to run more partitioning strategies.

The recommended portfolio (multijob) line shows the result of simulating a recommended portfolio that includes *all* versions of osmt-scatter and decision-cube (from 2 up to 128) for different numbers of cores. We schedule the smaller partitions first, and no core is allowed to do more than 20 minutes of work. The hybrid (multijob) shows the results of using $N/2$ cores for a scrambling portfolio and running the multijob scheduling algorithm on the other $N/2$ cores. The best strategy is hybrid (multijob) and with 256 cores, it improves the PAR-2 score by 34% (compared to a single sequential solver). Table II shows the total number of problems solved by each strategy when 256 cores are used. Note that each additional problem solved represents a significant step forward as these are very difficult problems unsolved by the previous techniques. These results show that in order to achieve robustness and consistent performance improvement, advanced portfolio techniques that incorporate multiple partitioning strategies should be preferentially used over more fragile individual partitioning strategies.

VII. CONCLUSION

We have shown that a portfolio of partitioning strategies, hybrid portfolios in particular, can outperform traditional portfolio solving for a range of SMT problems. These new strategies are a step toward better utilization of HPC and cloud systems for solving SMT problems. Additionally, we show that these new portfolio techniques perform much better than individual partitioning strategies and even manage to outperform a traditional portfolio.

There are many promising directions for future work that we are looking forward to investigating. For individual partitioning strategies, we plan to systematically explore the $cubeSize$ parameter for SCATTER, improve our decision-scatter algorithm to work similar to OpenSMT2's version, explore additional atom selection heuristics, and find a way to achieve the consistency of TIME for $tHeur$ without sacrificing determinism. We also plan to implement information sharing, explore recursive partitioning, push the multijob optimization further, and expand the benchmarks for evaluation, including other quantifier-free logics and quantified benchmarks.

REFERENCES

[1] A. Niemetz and M. Preiner, "Bitwuzla at the SMT-COMP 2020," *CoRR*, vol. abs/2006.01621, 2020. [Online]. Available: https://arxiv.org/abs/2006.01621

[2] 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, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24

[3] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *Proceedings of TACAS*, ser. LNCS, N. Piterman and S. Smolka, Eds., vol. 7795. Springer, 2013.

[4] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, "OpenSMT2: An SMT solver for multi-core and cloud computing," 2016.

[5] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, Jul 2014, p. 737–744.

[6] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 337–340.

[7] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using SMT-based context-bounded model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 331–340. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/1985793.1985839

[8] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-based model checking for recursive programs," *Formal Methods in System Design*, vol. 48, pp. 175–205, 2014.

[9] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 2, feb 2011. [Online]. Available: https://doi-org.stanford.idm.oclc.org/10.1145/1890028.1890031

[10] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, "Counterexample-guided quantifier instantiation for synthesis in smt," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 198–216.

[11] J. Peleska, E. Vorobev, and F. Lapschies, "Automated test case generation with SMT-solving and abstract interpretation," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 298–312.

[12] W. Steiner, "An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 375–384.

[13] R. Sebastiani and P. Trentin, "OptiMathSAT: A tool for optimization modulo theories," *Journal of Automated Reasoning*, pp. 1–38, 2015.

[14] C. M. Wintersteiger, Y. Hamadi, and L. de Moura, "A concurrent portfolio approach to SMT solving," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 715–720.

[15] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Hardware and Software: Verification and Testing*, K. Eder, J. Lourenço, and O. Shehory, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 50–65.

[16] M. J. H. Heule and H. van Maaren, "Look-ahead based sat solvers," in *Handbook of Satisfiability*, 2021.

[17] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, "Lookahead in partitioning SMT," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 271–279.

[18] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to dpll(t)," *J. ACM*, vol. 53, no. 6, p. 937–977, nov 2006.

[19] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, "A distribution method for solving sat in grids," in *Theory and Applications of Satisfiability Testing - SAT 2006*, A. Biere and C. P. Gomes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–435.

[20] M. Marescotti, A. Hyvärinen, and N. Sharygina, "SMTS: Distributed, visualized constraint solving," in *Logic Programming and Automated Reasoning*, 2018.

[21] M. Marescotti, A. E. J. Hyvärinen, and N. Sharygina, "Clause sharing and partitioning for cloud-based SMT solving," in *Automated Technology for Verification and Analysis*, C. Artho, A. Legay, and D. Peled, Eds. Cham: Springer International Publishing, 2016, pp. 428–443.

[22] A. Hyvärinen and C. M. Wintersteiger, "Parallel satisfiability modulo theories," in *Handbook of Parallel Constraint Reasoning*, 2018.

[23] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, "Search-space partitioning for parallelizing smt solvers," in *Theory and Applications of Satisfiability Testing – SAT 2015*, M. Heule and S. Weaver, Eds. Cham: Springer International Publishing, 2015, pp. 369–386.

[24] G. Andersson, P. Bjesse, B. Cook, and Z. Hanna, "Design automation with mixtures of proof strategies for propositional logic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 8, pp. 1042–1048, 2003.

[25] C. Reisenberger, "Pboolector: a parallel smt solver for qf_bv by combining bit-blasting with look-ahead," Ph.D. dissertation, Master's thesis, Johannes Kepler Univesität Linz, Linz, Austria, 2014.

[26] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014. [Online]. Available: https://doi.org/10.3233/sat190101

[27] A. Biere and A. Fröhlich, "Evaluating cdcl variable scoring schemes," in *Theory and Applications of Satisfiability Testing – SAT 2015*, M. Heule and S. Weaver, Eds. Cham: Springer International Publishing, 2015, pp. 405–422.

[28] H. Barbosa, F. Bobot, and J. Hoenicke, "SMT-COMP 2022."

[29] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," http://smtlib.cs.uiowa.edu, 2016.

[30] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, "SAT competition 2020," *Artificial Intelligence*, vol. 301, p. 103572, 2021.

[31] T. Weber, A. Niemetz, J. Hoenicke, A. Hyvärinen, H. Barbosa, and M. Schlaipfer, "SMT-COMP benchmark scrambler," https://github.com/SMT-COMP/scrambler, 2023.