Proof-Stitch: Proof Combination for Divide-and-Conquer SAT Solvers

Abhishek Nair, Saranyu Chattopadhyay, Haoze Wu, Alex Ozdemir, and Clark Barrett
Stanford University, Stanford, USA.
{aanair, saranyuc, haozewu, aozdemir, barrettc}@stanford.edu

***Abstract*—With the increasing availability of parallel computing power, there is a growing focus on parallelizing algorithms for important automated reasoning problems such as Boolean satisfiability (SAT). Divide-and-Conquer (*D&C*) is a popular parallel SAT solving paradigm that partitions SAT instances into independent sub-problems which are then solved in parallel. For unsatisfiable instances, state-of-the-art D&C solvers generate DRAT refutations for each sub-problem. However, they do not generate a single refutation for the original instance. To close this gap, we present *Proof-Stitch*, a procedure for combining refutations of different sub-problems into a single refutation for the original instance. We prove the correctness of the procedure and propose optimizations to reduce the size and checking time of the combined refutations by invoking existing trimming tools in the proof-combination process. We also provide an extensible implementation of the proposed technique. Experiments on instances from last year's SAT competition show that the optimized refutations are checkable up to seven times faster than unoptimized refutations.**

***Index Terms*—Parallel SAT, Divide and Conquer, Refutation Checking**

I. Introduction

Boolean satisfiability (SAT) solvers have improved dramatically in recent years. They are now regularly used in a wide variety of application areas including hardware verification [1], computational biology [2] and decision planning [3].

With the emergence of cloud-computing and improvements in multi-processing hardware, the availability of parallel computing power has also increased dramatically. This has naturally led to an increased focus on parallelizing important algorithms, and SAT is no exception. There are two traditional approaches to parallel SAT solving - the Divide-and-Conquer (*D&C*) approach [4]–[6] and the portfolio approach [7]. In the D&C approach, the original SAT instance is partitioned into independent sub-problems to be solved in parallel, while in the portfolio approach multiple SAT solvers are independently run on the original instance. Although the portfolio approach in combination with clause sharing performs well for small portfolio sizes, the D&C approach scales better in environments with large parallel computing power such as the cloud. Several implementations of D&C solvers exist [4]–[6], [8]. Every implementation uses: a *divider* to split up the original instance into sub-problems, and a *base SAT solver* to solve the independent sub-problems. For example, `ggSAT` [8] uses CadiCaL [9] as its base solver.

If a SAT problem is unsatisfiable, a proof of unsatisfiability (or *refutation*) can be produced and independently checked to validate the result. Since 2013, the annual SAT competition has required SAT solvers to generate refutations. The most commonly supported refutation format today is the DRAT format [10]. Existing D&C SAT solvers produce refutations for each sub-problem independently. However, even if the refutation for each sub-problem passes the proof-checker, this is not a formal guarantee that the original instance also admits a refutation, as there could have been an error in the partitioning strategy. For example, a buggy solver may incompletely partition the SAT instance $(\neg\ell_1) \wedge (\ell_2 \vee \ell_3) \wedge (\neg\ell_2 \vee \ell_3)$ into sub-problems with cubes ℓ_1 and $\neg\ell_2$. Both of these sub-problems are unsatisfiable, even though the instance is satisfiable. Transient errors in the underlying distributed system may also cause sub-problem refutations to be truncated or missing. To address these challenges, we introduce *Proof-Stitch*, which implements a strategy for combining DRAT refutations for sub-problems into a single refutation for the original instance, a process we call *refutation stitching*. Our contributions are:

- We describe an algorithm for combining DRAT refutations of partitions of problems into a single refutation for the original problem and provide an open-source implementation on GitHub [11].
- We describe an optimization technique leveraging existing trimming tools (e.g., *drat-trim* [12]) to improve the quality of the combined refutations.
- We evaluate our implementation on benchmarks from last year's SAT competition [13]. Our results show that trimmed refutations are checkable up to seven times faster than untrimmed refutations.

The rest of this paper is organized as follows. Section II discusses background and related work. Section III presents the *Proof-Stitch* algorithm and theoretically justifies our method of combining refutations. We also describe an optimization technique that reduces the checking time and the size of the combined refutations. Section IV details our tool implementation. Results are presented in Section V, and Section VI concludes.

This work was partially funded by a gift from Amazon Web Services' Automated Reasoning group.

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_14

II. Background and Related Work

A. Propositional refutations

We assume familiarity with the basic concepts of CDCL SAT algorithms (see, e.g., [14]). We also assume that a base SAT solver can produce a *DRAT* refutation, which we define below (following [15]).

Throughout the paper we model clauses as *sets* of literals and formulas as *multisets* of clauses. By $\cdot \cup \cdot$, we denote the standard union operation on sets, and the multiplicity-summing union on multisets.

Let $F = \{C_1, \ldots, C_n\}$ be a formula. F *unit propagates on* ℓ to $F' = \{C \setminus \{\neg\ell\} : C \in F, \ell \notin C\} \cup \{\ell\}$ (written $F \to_\ell F'$) if there exists a clause $\{\ell, \ell_1, \ldots, \ell_k\} \in F$ such that $\{\neg\ell_i\} \in F$ for $i \in [1, k]$. If $F \to_\ell F'$ for some ℓ, then $F \to F'$. We say that $F \to \bot$ if F contains an empty clause. Let the relation $\to^*$ denote the reflexive, transitive closure of $\to$. We say that $F \mapsto F'$ when $F \to^* F'$ and there is no $F'' \neq F'$ such that $F' \to F''$. One can show that the $\mapsto$ relation is a function. We say that $C = \{\ell_1, \ldots, \ell_k\}$ has *asymmetric tautology* (AT) with respect to F if $F \cup \{\neg\ell_1\} \cup \cdots \cup \{\neg\ell_k\} \mapsto \bot$. We say that C has *resolution asymmetric tautology* (RAT) with respect to literal $\ell_1 \in C$ and F if for all $C' \in F$ containing $\neg\ell_1$, $C \cup (C' \setminus \{\neg\ell_1\})$ has AT.

Let o_i denote an operation. Consider a sequence of operation-clause pairs $\pi = ((o_1, C_1), \ldots, (o_m, C_m))$, where each o_i indicates either the addition ($\oplus$) or deletion ($\ominus$) of a clause from a formula.

Let ϕ denote a CNF formula. Define ϕ_i recursively: $\phi_0 = \phi$, and ϕ_{i+1} is $\phi_i \cup \{C_{i+1}\}$ when o_{i+1} is $\oplus$, or $\phi_i \setminus \{C_{i+1}\}$ otherwise. The sequence π is a *DRAT refutation* of ϕ if when $o_{i+1} = \oplus$ then C_{i+1} has RAT with respect to ϕ_i, and if the last element in π is $(\oplus, \emptyset)$.

B. Divide-and-Conquer SAT solving

One parallel SAT solving paradigm is *Divide-and-Conquer*: a SAT instance is divided into simpler SAT instances (sub-problems), which are then solved in parallel. Typically, the sub-problems represent partitions of the search space, such that the disjunction of all the sub-problems is equisatisfiable with the original problem. The sub-problems are derived from the original instance by assigning Boolean values to literals. The set of literals that are assigned (decided) for a particular sub-problem is called the *cube* of the sub-problem and the number of literals in the cube is the *depth* of the sub-problem. There are many D&C-based solvers [4]–[6], including: `Psato` [16], `Painless` [17], and `AMPHAROS` [18]. One prominent D&C approach, Cube-and-Conquer [19], uses a lookahead solver to divide instances and a CDCL solver to solve sub-problems. This approach has been successful for large mathematical problems [20] and is implemented by tools such as `Paracooba` [21] and `gg-sat` [8].

D&C SAT solvers generate separate DRAT refutations for each sub-problem. There has been little work on combining these refutations into a single refutation for the original instance. One work [22] considers proof composition, but its parallel composition rule does not apply to DRAT refutations. Another work [23] gives an alternate proof calculus for parallel solvers.

III. Methodology

In this section, we present an algorithm to combine sub-problem refutations into a refutation for the original Boolean instance. Then we show the algorithm's correctness. Finally, we present a technique to optimize the combined refutations.

A. Algorithm

The first step in the *Proof-Stitch* algorithm is to construct a decision tree representing the steps taken by the D&C solver. The root of the tree represents the original instance, and the leaves represent the sub-problems. Figure 1 shows the decision tree for an example instance.

Algorithm 1: Stitching algorithm

In : Instance: ϕ,
Decision literal: x,
Refutations of:
$\phi \cup \{\{x\}\}$: $\pi = ((o_1, C_1), \ldots, (o_n, C_n))$,
$\phi \cup \{\{\neg x\}\}$: $\pi' = ((o'_1, C'_1), \ldots, (o'_m, C'_m))$,
Out : Refutation of ϕ
procedure *stitching* (ϕ, x, π, π')
return

$$\Big((o_1, C_1 \cup \{\neg x\}),\ \ldots,\ (o_n, C_n \cup \{\neg x\}),\ (o'_1, C'_1 \cup \{x\}),\ \ldots,\ (o'_m, C'_m \cup \{x\}),\ (\oplus, \emptyset) \Big)$$

Next, *Proof-Stitch* performs a sequence of *stitching* operations to produce a single refutation for the original SAT instance. A stitching operation (Algorithm 1) reads in a SAT instance ϕ, a decision variable x and two refutations π and π' corresponding to the sub-problems $\phi \cup \{\{x\}\}$ and $\phi \cup \{\{\neg x\}\}$ respectively. It produces a single refutation corresponding to the instance ϕ. The refutation for instance ϕ contains the clauses from refutation π appended with the literal $\neg x$ and the clauses from refutation π' appended with the literal x. More generally, the clauses from a refutation are appended with the negation of the decision literal used to generate the sub-problem. Figure 2 illustrates the stitching operation.

As an example of the proof combination process, consider Figure 3. First the refutations π_{00} and π_{01} are combined. Then π_{10} and π_{11} are combined, and finally, π_0 and π_1 are combined to produce the refutation π corresponding to the original instance. In *Proof-Stitch*, the stitching operations are ordered according to the following rule: A stitching operation to combine a pair of refutations π and π' can only occur after all refutations with greater depth have been combined. Informally, this means that refutations are combined in decreasing order of their depth, as shown in Figure 3. Stitching operations at the same depth are independent and can occur in parallel.

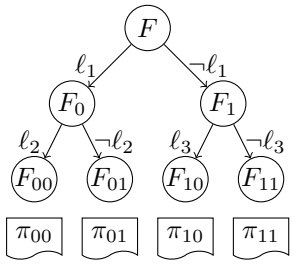

Fig. 1: Decision tree of an example unsatisfiable SAT instance.

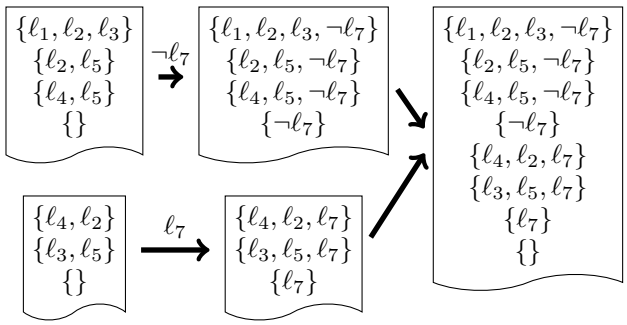

Fig. 2: Stitching operation on example refutations

B. Justification for the stitching operation

We now show that Algorithm 1 is correct: given suitable inputs, it produces a DRAT refutation for ϕ.

Definition 1. *A DRAT refutation π is **preserving** if for all C, $(\ominus, C)$ occurs at most as many times in π as $(\oplus, C)$.*

Lemma 1. *Let ϕ be a CNF formula, x be a variable, and π and π' be preserving DRAT refutations of $\phi \cup \{\{x\}\}$ and $\phi \cup \{\{\neg x\}\}$ respectively. Then, stitching(ϕ, x, π, π') outputs a preserving DRAT refutation of ϕ.*

Proof. Let π^* be the output of *stitching*. Let $\pi = ((o_1, C_1), \ldots, (o_n, C_n))$ and $\pi' = ((o'_1, C'_1), \ldots, (o'_{n'}, C'_{n'}))$. Let $\psi = \phi \cup \{\{x\}\}$ and $\psi' = \phi \cup \{\{\neg x\}\}$. Define ψ_i recursively, by $\psi_0 = \psi$ and $\psi_{i+1} = \psi_i \cup \{C_{i+1}\}$ when o_{i+1} is an addition, and $\psi_{i+1} = \psi_i \setminus \{C_{i+1}\}$ otherwise. Define ψ'_i (respectively ϕ_i) analogously, based on formula ψ' (resp. ϕ) and refutation π' (resp. π^*).

By construction, π^*'s final step is $(\oplus, \emptyset)$. Moreover, since π and π' are preserving and formulas are clause *multisets*, π^* is preserving. Thus, our main task is to show that each addition $(\oplus, C^*_{i+1})$ in π^* has RAT with respect to ϕ_i. C^*_{i+1} is either derived from a clause in π, derived from a clause in π', or is the final empty clause. We begin with the first case: $C^*_{i+1} = C_{j+1} \cup \{\neg x\}$.

First, we show that if C_{j+1} has AT with respect to ψ_j, then C^*_{i+1} has AT with respect to ϕ_i. Note that $\psi_j \cup \{\{\neg\ell_1\}, \ldots, \{\neg\ell_k\}\} = F' \cup \{\{x\}\} \cup \{\{\neg\ell_1\}, \ldots, \{\neg\ell_k\}\} \rightarrow_x F'' \cup \{\{x\}\} \cup \{\{\neg\ell_1\}, \ldots, \{\neg\ell_k\}\} \mapsto \bot$. Now, consider $F''' = \phi_i \cup \{\{x\}, \{\neg\ell_1\}, \ldots, \{\neg\ell_k\}\}$. If $F''' \mapsto \bot$, then C^*_{i+1} has the desired property. Observe that $F''' \rightarrow_x F'' \cup \{\{x\}\} \cup \{\{\ell_1\}, \ldots, \{\ell_k\}\}$; thus, since the latter propagates to bottom, F''' does too.

Second, we show that if C_{j+1} has RAT with respect to literal ℓ and formula ψ_j, then $C^*_{i+1} = \{\neg x\} \cup C_{j+1}$ has RAT with respect to literal ℓ and formula ϕ_i. Let C^* be a clause in ϕ_i that contains $\neg\ell$. If $C^*_{i+1} \cup (C^* \setminus \{\neg\ell\})$ has AT with respect to ϕ_i, we are done. Since C^* is a clause in ϕ_i, there is some C in ψ_j such that $C \cup \{\neg x\} = C^*$ or $C = C^*$. Thus, $C^*_{i+1} \cup (C^* \setminus \{\neg\ell\}) = \{\neg x\} \cup C_{j+1} \cup (C \setminus \{\neg\ell\})$. Let $\neg x, \ell_1, \ldots, \ell_k$ be the literals of this clause. As before, since $\psi_j \cup \{\{\neg\ell_1\}, \ldots, \{\neg\ell_k\}\}$ unit propagates to bottom, $\phi_i \cup \{\{x\}, \{\neg\ell_1\}, \ldots, \{\neg\ell_k\}\}$ does too.

In the case that $C^*_{i+1} = C'_{j+1} \cup \{x\}$ (i.e., C^*_{i+1} is derived from π'), the argument is similar. The key insight is that an initial propagation on $\neg x$ in any AT check removes all the clauses added by π. Since π deletes no clauses from the original formula, this leaves an intermediate propagation result that shows C'_{j+1} is RAT.

The final step in π^* is $(\oplus, \emptyset)$. It has AT because ϕ_{n+m} contains both $\{x\}$ and $\{\neg x\}$. Since π^*'s added clauses all have the AT or RAT properties, and the final step adds an empty clause, π^* is a valid DRAT refutation of ϕ.

□

In *Proof-Stitch*, the final refutation is built through stitching operations on DRAT refutations of the sub-problems. Since each stitching operation produces a preserving DRAT refutation, recursive application of Lemma 1 proves that the final refutation is a valid DRAT refutation of the original instance.

C. Optimization

Empirically, we have observed that refutations created through stitching operations contain a large number of clauses that are not needed during validation ("redundant" clauses). Identifying and removing these clauses reduces the time required to check the refutation and the storage space required to save the refutation. One approach to remove such redundant clauses is by identifying the "unsatisfiable core" as described in [24]. This approach optimizes the refutation by only retaining clauses that are essential for validation by a proof-checker. Our implementation optimizes refutations by using *drat-trim* to extract the unsatisfiable core after every stitching operation.

However, aggressively invoking the optimization technique (e.g., after every stitching operation) could incur significant runtime overhead in the refutation generation process. This calls for a heuristic to decide when to apply the optimization technique. Empirically we observe that refutations with larger clauses (more literals) require longer to check. We hypothesize that this occurs because larger clauses are less likely to contribute to unit-propagation while simultaneously consuming more memory in the cache of the refutation checker. Therefore, optimizing refutations with large clauses should yield the greatest benefit. To implement this, we introduce a threshold parameter CL_{avg}. After each stitching step, the refutation is optimized only if the average clause length in the refutation is greater than CL_{avg}.

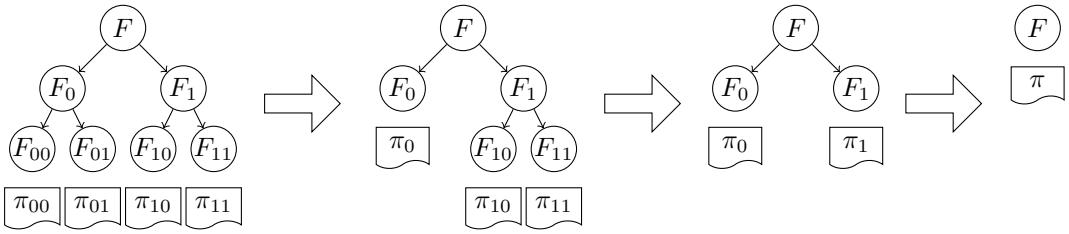

Fig. 3: Refutation stitching process for the SAT instance shown in Figure 1. The decision literals are omitted.

IV. Implementation

In this section, we describe our implementation of the *Proof-Stitch* algorithm. *Proof-Stitch* is implemented in Python and uses *drat-trim* [12] to optimize refutations. Our tool comprises of just under 300 lines of Python code and is available on GitHub [11].

The tool inputs are the original SAT instance in CNF form, the refutations and cubes for each sub-problem, and the threshold value CL_{avg}. Our implementation requires that the cube of each sub-problem be encoded in the name of the corresponding refutation file. For example, the refutation file corresponding to refutation π_{00} in Figure 1 is named $\ell_1\_\ell_2.proof$. The output is a single file containing a refutation of the original instance. As noted in section III, stitching operations at the same depth of the decision tree are independent and their combined refutations can be optimized in parallel. Our tool supports this. Setting the parameter $CL_{avg} = 0$ enables optimization after every stitching operation and $CL_{avg} = -1$ turns off optimization (only stitching is performed). We denote refutations combined with $CL_{avg} = 0$ as "fully optimized" and refutations combined with $CL_{avg} = -1$ as "unoptimized".

V. Experiments

To evaluate *Proof-Stitch*, we run it on six benchmarks from the parallel track of last year's SAT competition [13]. The chosen benchmarks can be solved by `Paracooba` [21] within 1 minute of run-time. We also attempted running the tool on harder instances from the parallel track. While unoptimized proofs can be produced quickly (within a few minutes) on those instances, proof-checking and optimization are both computationally prohibitive due to the limitation of the underlying proof-checker (e.g., *drat-trim* fails to validate the combined refutations on harder instances even with a 24 hour time limit). For large refutations, the proof-checker faces memory and run-time bottlenecks on almost all the intermediate optimization steps. Therefore, we do not consider harder instances in our evaluation, but note that the proposed techniques in principle apply to larger instances once the scalability of the underlying proof-checker improves.

In our experiments, we compare the checking time and size of unoptimized refutations against fully optimized refutations to show the benefit of optimization. We also report the tool run-time to demonstrate that *Proof-Stitch* does not introduce unacceptable overheads. Finally, we analyze the average checking time and tool run-time for $CL_{avg} = 10$, a value empirically determined to perform well. We perform our evaluation on an Intel Xeon E5-2640 v3 machine with 128 GBytes of DRAM and 16 cores.

TABLE 1: Refutation checking time (T_c) (s), tool run-time (T_g) (s), and size of refutation file (S_g) (MB) for six benchmarks from last year's SAT competition [13]

Benchmarks	Un-optimized			Fully Optimized		
	T_c(s)	T_g(s)	S_g(MB)	T_c(s)	T_g(s)	S_g(MB)
p01_lb_05	987	271	1700	141	686	184
ktf_TF-4.tf_2_0.02_18	212	78	385	76	600	77
satch2ways12u	1370	275	1600	272	836	655
pb_300_10_lb_06	163	107	536	36	459	27
mp1-Nb6T06	241	106	586	44	201	222
E02F17	417	223	1500	112	467	294

Table 1 shows the time required for *drat-trim* to check the final refutations for the benchmarks (T_c), tool execution time to combine refutations (T_g), and the size of the combined refutations (S_g). The time required to check refutations reduces by between $(2.7 - 7)\times$ for all the benchmarks when full optimization is performed. Full optimization also results in smaller refutation file sizes, but increases the tool run-time.

Figure 4 compares the average run-time to combine refutations (denoted "merging" time) and the average run-time to check refutations for unoptimized, $CL_{avg} = 10$, and fully optimized refutations. Interestingly, running our tool with $CL_{avg} = 10$ *decreases* the total validation time (merging + checking) compared to the unoptimized case. This points to the benefit of optimizing refutations in parallel—the overhead associated with optimizing refutations can be amortized by the savings in refutation checking time. Another important observation is that setting $CL_{avg} = 10$ reduces the time required to combine refutations compared to the unoptimized case. We believe the reason is as follows: optimizing refutations decreases their size. When $CL_{avg} = 10$, we optimize all intermediate refutations with average clause length greater than 10. Since the intermediate refutations are now smaller, the next stitching operation on this refutation takes lesser time. The time spent in optimizing refutations is mitigated by the savings in stitching time.

VI. Conclusion

We have presented *Proof-Stitch*, a technique that complements Divide-and-Conquer SAT solvers by combining sub-problem refutations into a single refutation for the original

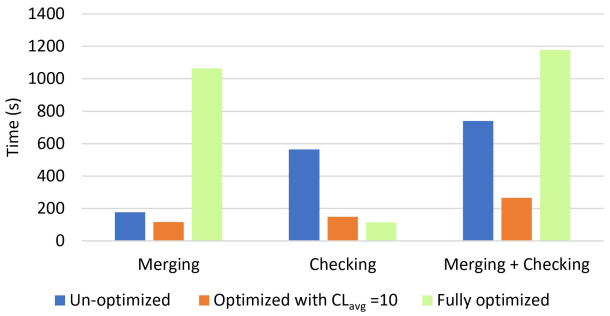

Fig. 4: Average merging time and refutation checking time when the refutations are not optimized, optimized with $CL_{avg} = 10$ and fully optimized

instance. *Proof-Stitch* also uses existing proof-trimming tools to optimize the combined refutation.

Future Work: Proof-Stitch's run-time overhead can be reduced by performing more stitching operations in parallel. Currently, only stitching operations at the same tree depth are parallelized, while in principle, any two independent stitching operations could be parallelized. Another potential future direction would be to incorporate parallelism in the refutation checker itself, likely requiring extension of the DRAT format to incorporate structural information of the search tree. Finally, it would be interesting to evaluate alternative measures for guiding the optimization process, such as Literal Block Distance [25], and to look into additional ways to reduce refutation sizes.

Acknowledgement: This work began as a course project for Caroline Trippel's CS357S (Fall 2021) at Stanford University.

REFERENCES

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

[2] A. Graça, J. Marques-Silva, I. Lynce, and A. L. Oliveira, "Efficient haplotype inference with pseudo-boolean optimization," in *Proceedings of the 2nd International Conference on Algebraic Biology*, ser. AB'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 125–139.

[3] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI)*, 1992, pp. 359–363.

[4] W. Blochinger, C. Sinz, and W. Küchlin, "Parallel propositional satisfiability checking with distributed dynamic learning," *Parallel Computing*, vol. 29, no. 7, pp. 969–994, 2003.

[5] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, "Partitioning sat instances for distributed solving," in *Logic for Programming, Artificial Intelligence, and Reasoning*, C. G. Fermüller and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 372–386.

[6] A. E. Hyvärinen, T. Junttila, and I. Niemelä, "A distribution method for solving sat in grids," in *International conference on theory and applications of satisfiability testing*. Springer, 2006, pp. 430–435.

[7] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008.

[8] A. Ozdemir, H. Wu, and C. Barrett, "Sat solving in the serverless cloud," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 241–245.

[9] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[10] M. J. Heule and A. Biere, "Proofs for satisfiability problems," *All about Proofs, Proofs for all*, vol. 55, no. 1, pp. 1–22, 2015.

[11] "Proof-stitch," https://github.com/abhisheknair1729/Proof-Stitch/commit/d93a0c33b6114044413eb22962c677b06308b00e, 2022.

[12] N. Wetzler, M. J. Heule, and W. A. Hunt, "Drat-trim: Efficient checking and trimming using expressive clausal proofs," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2014, pp. 422–429.

[13] "Sat competition 2021," https://satcompetition.github.io/2021/, 2021.

[14] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability: Second Edition*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021. [Online]. Available: https://books.google.com/books?id=dUAvEAAAQBAJ

[15] M. Heule, M. Järvisalo, and A. Biere, "Clause elimination procedures for cnf formulas," in *LPAR*, 2010.

[16] H. Zhang, M. P. Bonacina, and J. Hsiang, "Psato: a distributed propositional prover and its application to quasigroup problems," *Journal of Symbolic Computation*, vol. 21, no. 4, pp. 543–560, 1996.

[17] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Modular and efficient divide-and-conquer sat solver on top of the painless framework," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 135–151.

[18] S. Nejati, Z. Newsham, J. Scott, J. H. Liang, C. Gebotys, P. Poupart, and V. Ganesh, "A propagation rate based splitting heuristic for divide-and-conquer solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2017, pp. 251–260.

[19] M. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 7261. Springer, 2011, pp. 50–65.

[20] M. J. H. Heule, O. Kullmann, and V. W. Marek, "Solving and verifying the boolean pythagorean triples problem via cube-and-conquer," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9710. Springer, 2016, pp. 228–245.

[21] M. Heisinger, M. Fleury, and A. Biere, "Distributed cube and conquer with paracooba," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2020, pp. 114–122.

[22] M. J. Heule and A. Biere, "Compositional propositional proofs," in *Logic for Programming, Artificial Intelligence, and Reasoning*, 2015.

[23] T. Philipp, "Unsatisfiability proofs for parallel sat solver portfolios with clause sharing and inprocessing." in *GCAI*, 2016, pp. 24–38.

[24] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for cnf formulas," in *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003, pp. 886–891.

[25] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.