Karlsruher Institut für Technologie

Master thesis

Fuzzy Clause Weights Through Trail Sampling

Achim Kriso

Date: 22. März 2023

Reviewer: Prof. Dr. Peter Sanders
Second reviewer: Prof. Dr. Thomas Bläsius
Supervisors: Dr. Markus Iser
Dominik Schreiber

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

The work on efficiently solving the Boolean Satisfiability Problem (SAT) has progressed in great strides within the last decades. With the rise of multi-core processors the parallel Portfolio architecture has been dominant for designing a successful parallel SAT solver. Such parallel solvers run multiple single threaded SAT solvers simultaneously and exchange learned clauses between them to improve their performance. In this thesis, we define *literal stability* as a statistical aggregation of the assignment values of a literal by sampling the trail in regular intervals. We define heuristics using literal stability which assign fuzzy weights to each foreign clause as a measure of usefulness and only import the best. Finally, we empirically evaluate these heuristics inside the clause exchange procedure. Unfortunately, our approach was unable to show any improvement over the default.

Die Arbeit an der effizienten Lösung des booleschen Erfüllbarkeitsproblems (SAT) hat in den letzten Jahrzehnten große Fortschritte gemacht. Mit dem Aufkommen von Mehrkernprozessoren hat sich die parallele Portfolio Architektur als dominant für das Design einenes erfolgreichen parallelen SAT-Lösers herausgestellt. Solche parallelen Löser führen mehrere sequentielle SAT-Löser gleichzeitig aus und tauschen die gelernte Klauseln zwischen ihnen aus, um ihre Leistung zu verbessern. In dieser Arbeit definieren wir die *Literalstabilität* als eine statistische Aggregation der Zuweisungswerte von Literalen durch regelmäßiges Abtasten. Wir definieren Heuristiken welche Literalstabilität verwenden um Gewichte als Maß für deren Nützlichkeit zu jeder fremden Klausel zuweisen. Die besten Klauseln werden jeweils importiert. Schließlich bewerten wir diese Heuristiken empirisch im Rahmen des Klauselaustauschverfahrens. Leider konnten wir mit unserem Ansatz keine Verbesserung nachweisen.

Acknowledgments

Many thanks to my supervisor Dr. Markus Iser for his guidance during this thesis. I also want to thank Dominik Schreiber for any assistance when working with Mallob and patience with any issues I encountered. Finally, I want to express my gratitude to the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for providing computing time on the Supercomputer SuperMUC NG at Leibniz Supercomputing Centre, as well as Prof. Peter Sanders and his institute for the provided computing resources.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Contents

1 Introduction

The boolean Satisfiability Problem (SAT) is a famous algorithmic problem in computer science. It asks whether there exists a variable assignment for some boolean formula in conjunctive normal form, so that the formula evaluates to true. Both theoretical research and practical applications have explored this problem in depth within the last half century. On the theoretical side it is at the core of the famous P vs NP question as it was the first problem proven to be NP-complete [8]. Every subsequent proof for the NP-hardness of a problem was performed using a (transitive) polynomial transformation from the SAT problem. As computer hardware performance increased and more advanced SAT solving algorithms were developed it became feasible to use SAT solvers in practical application. For example, modern constraint solvers use SAT solvers internally as part of their solving procedures [11]. Microchip manufacturers can encode their circuits into propositional formulas and use SAT solvers to formally verify the correctness before starting the expensive production process [23]. Other uses are found in bioinformatics [24], automated planning [21] and cryptography [26]. On a fundamental level, a SAT solver performs an exhaustive search on all possible solutions. With the use of clause learning, restart policies and search heuristics enhanced by highly optimized data structures, it has become feasible to solve problems containing millions of variables and clauses. As the performance of a single processor core plateaued, the next natural step was to explore SAT solving in a multi-core setting. A common approach is to run multiple sequential SAT solvers in parallel and coordinate their solving procedures. Each sequential solver is running a unique configuration to diversify the exploration of the search space. Parallel SAT solvers will often use multiple distinct SAT solver implementations. In order to further benefit from multi-core systems, the clauses learned by each sequential solver are shared, so every solver can benefit by the work of its siblings. However, importing clauses too eagerly carries a significant performance cost [4]. Therefore, it is very important to be selective when importing foreign clauses.

In this thesis we develop a new approach for clause sharing using the notion of *literal stability*. Literal stability summarizes the recent assignment behavior of each variable. This information is gathered during solver execution by sampling the current trail. We develop an efficient implementation to perform trail sampling. A solver can use literal stability information to predict the usefulness of a clause to determine if it should be imported. We explore and analyze seven different heuristics which integrate stability to optimize clause selection.

Structure of Thesis

We start by providing an explanation of the fundamentals needed to understand the rest of this thesis in Chapter 2. We cover basic definitions and provide background on the current state-of-the-art SAT solving technologies. In Chapter 3 we summarize related work that explores different strategies for effective clause sharing. We continue describing our own approach in Chapter 4. To analyze the effectiveness of our strategy and evaluate the results we perform a variety of benchmarks in Chapter 5. Finally, in Chapter 6 we give our final thoughts and give suggestions for future work.

2 Fundamentals

This chapter summarizes important concepts used in this thesis, as well as other relevant background information.

2.1 Definitions

Let V be the set of boolean variables[1]. Each variable $v \in V$ can only be assigned one of three values of the set $\mathbb{S} = \{T, F, U\}$. Where T stands for true, F for false and U represents unassigned. The negation of all elements in $\mathbb{S}$ is defined as $\neg T = F$, $\neg F = T$ and $\neg U = U$. Literals $L = V \cup \{\neg v | v \in V\}$ are the set of all variables and their logical negations. Their complement operator $\bar{\cdot} : L \to L$ is defined as

$$\bar{l} = \begin{cases} \neg l & \text{if } l \in V \\ v & \text{if } l = \neg v, v \in V \end{cases}$$

We define a clause c to be a set of literals $\{l_0, l_1 \dots l_k\}$ which are implicitly connected by disjunction. A SAT formula f is the conjunction of a set of clauses $\{c_0, c_1 \dots c_n\}$.

A variable assignment is a function $\sigma : V \to \mathbb{S}$. If all variables in an assignment are either T or F we call it a *complete assignment*. We define the semantics of a SAT formula using an interpretation function I which determines the truth value of a formula given a variable assignment. I is defined as:

$$I(l, \sigma) = \begin{cases} \sigma(v) & \text{if } l \in V \\ \neg\sigma(v) & \text{if } \bar{l} \in V \end{cases}$$

$$I(c, \sigma) = \begin{cases} T & \text{if } T \in W_c := \{I(l, \sigma) \mid l \in c\} \\ U & \text{if } T \notin W_c,\ U \in W_c \\ F & \text{else} \end{cases}$$

$$I(f, \sigma) = \begin{cases} F & \text{if } F \in W_f := \{I(c, \sigma) \mid c \in f\} \\ U & \text{if } F \notin W_f,\ U \in W_f \\ T & \text{else} \end{cases}$$

[1] The variables are not strictly boolean as they can be assigned three values. The third value U is an artifact of the solving algorithm with partial solutions. The core intuition for boolean variables remains the same.

Given an assignment σ, a literal l is *satisfied* iff $I(l, \sigma) = T$. Likewise, a clause c is satisfied iff $I(c, \sigma) = T$ and formula f is satisfied, iff $I(f, \sigma) = T$. Otherwise, they are *unsatisfied*. We call a formula f *satisfiable* iff there exists a σ such that $I(f, \sigma) = T$. Otherwise, if there does not exist such an assignment, the formula is *unsatisfiable*. The SAT problem asks if a given SAT formula is satisfiable.

Example 2.1.1. *Satisfiable*

Let $f = \{\{x_1, x_2\}, \{\overline{x_1}, \overline{x_2}\}\}$. In order for f to be satisfied x_1 and x_2 can not be both false, as this would make the first clause unsatisfied. Likewise, if they are both true then the second clause is unsatisfied. If they are assigned different truth values, both literals will each satisfy one of the clauses each, thereby, satisfying the formula.

Example 2.1.2. *Unsatisfiable*

Let $f = \{\{x_1, x_2\}, \{\overline{x_1}, x_2\}, \{\overline{x_2}\}\}$. This formula is unsatisfiable. To prove unsatisfiability, one can try each combination of variable assignments. However, in this particular example we can infer that x_2 has to be false, in order to satisfy the third clause. Using this information we can effectively remove x_2 from the first and second clause because x_2 can not aid in satisfying these two clauses. Literals like $\overline{x_2}$ which are the last remaining unassigned literals within an unsatisfied clause are called unit literals. *This propagation of information is a core part of a modern SAT solver and further explained in Subsection 2.2.1. After propagating x_2, we are left with $\{\{x_1\}, \{\overline{x_1}\}\}$. Either assignment for x_1 will invalidate one remaining clause which proves the unsatisfiability of f.*

We adopt the notion of *triggering* as is defined in [32]. If the unit propagation of a sequential SAT solver finishes without causing a conflict, a triggering clause c would have allowed the unit propagation engine to make further progress by either causing a conflict or allow for a new literal to be propagated. Based on this motivation, a triggering clause is defined as follows: Given an assignment σ and a clause c of size s. We say that c *triggers* σ iff:

(i) For each literal $l \in c$, $I(l, \sigma) \neq T$.

(ii) For at least $s - 1$ literals $l \in c$, $I(l, \sigma) = F$.

The first case excludes clauses that contain a satisfied literal, as such clauses are ignored during unit propagation. The second case specifies that at most one literal can be unassigned. Together both rules define a clause to trigger an assignment, iff the clause would cause unit propagation to result in a conflict or turn into a unit clause for further propagation.

2.2 SAT Solving Algorithms

Because this thesis explores details of parallel SAT solving using clause learning, we provide a short overview and explanation of the relevant SAT solving algorithms building up to state-of-the-art CDCL solvers.

2.2.1 DPLL

The DPLL algorithm [10], named after its inventors: Davis, Putnam, Logemann and Loveland is a complete algorithm which searches the space of all variable assignments trying to find a satisfying assignment.

On a high abstraction level, the DPLL algorithm eagerly assigns unit literals in a process called *unit propagation* and then recursively splits the search space by trying both truth values for a currently unassigned variable, called the *decision variable*. Unit propagation detects clauses that contain a single unassigned literal l where all other literals within the cause are unsatisfied. Such a clause has to assign l to be true, in order to satisfy this clause.

Every decision the current *decision level* increases by one starting with zero at the start of the solving procedure. A *conflict* occurs when all literals in a clause are false. In this case the current partial assignment cannot be part of a satisfiable assignment as is proven by the conflicting clause. If it ever encounters a satisfying assignment, it has shown the input formula to be satisfiable. Otherwise, if the search procedure finishes and never encountered a satisfying assignment, the input formula has to be unsatisfiable. The original paper [10] additionally uses *pure literals*, another mechanism similar to *unit literals*, to reduce the search space. However, subsequent iterations on the DPLL algorithm like CDCL ignore pure literals. Therefore, we have chosen to omit pure literals in the explanation of the DPLL algorithm.

2.2.2 Resolution

The Resolution algorithm is based on the resolution rule in logic which states that given two hypothesis clauses $\{x_0, \ldots, x_n, z\}$ and $\{y_1, \ldots, y_m, \ldots \overline{z}\}$ then $\{x_0, \ldots, x_n, y_1, \ldots, y_m\}$, the *resolvent*, has to be true as well. To informally justify the correctness of the resolution rule, consider both cases of z. If z is true, then there has to exist a y_i which is satisfied, otherwise the second hypothesis would be not satisfied. Since y_i is contained in the resolvent it is also satisfied. On the other hand, if z is false, there has to exist an x_i which is satisfied. By the same reasoning as before x_i is part of the resolvent and thereby satisfies it.

The resolution algorithm repeatedly tries to find two clauses in its input formula where the resolution rule applies. The resolvent is added to the formula. This operation preserves the satisfiability of the original formula. Because the empty clause is always unsatisfiable, if the resolution rule ever generates the empty clause, the entire formula has to be unsatisfiable. Otherwise, the algorithm will reach a fixed-point since there exists only a finite number of clauses on a finite number of variables. If the fixed-point is reached, then the formula is satisfiable.

2.2.3 CDCL

The CDCL algorithm [7], short for conflict driven clause learning, combines the DPLL algorithm with resolution. Resolution is used to learn a new clause at every conflict, which motivates the name CDCL. These new clauses allow unit propagation to assign more literals, thereby pruning entire subtrees of the search space.

The clause learning procedure works as follows: Every newly assigned literal, either by unit propagation or decision, is placed in order on the *trail*. The trail is a stack of literals which defines the literals that are assigned true under the current partial assignment. There is never a need to assign a literal to be false on the trail, as its complement can be assigned true instead.

If a literal on the trail was assigned due to unit propagation, it remembers which clause forced this literal to be assigned. The so-called *reason clause* of the assigned literal. Once a conflict occurs during the search, all literals in the conflicting clause are false by definition. There exists at least one literal in the conflicting clause which is false because its complement was assigned true due to unit propagation. This complement has an associated reason clause. Since the conflicting clause contains the original literal and the reason clause contains its literal complement, the resolution rule can be applied to create the resolvent of those two clauses. Continuing with the resolvent, we can iteratively apply resolution as long as there is still a literal assigned false due to unit propagation. The clause created in this process is added to the set of clauses maintained by the solver.

The addition of clause learning has dramatically improved the performance of SAT solvers. Whereas DPLL solvers would repeatedly solve the same subproblem, CDCL solvers can summarize its solution in the form of learned clauses. By propagating literals in the learned clauses the solver can skip the subproblem. Additionally, a newly learned clause defines which decision variable needs to be flipped in order for the clause to become satisfied. Backtracking to this decision level is called *non-chronological backtracking*.

2.3 Parallel SAT Solving

The frequency of single cores has been stagnant for at least a decade and it is unlikely it will increase exponentially ever again without a major breakthrough in microchip design. Modern CPU architectures, in response, have shifted to increasing the number of parallel cores on a CPU instead. With the rise of CPUs containing up to several hundred cores, SAT solving algorithms need to adjust if they want to utilize this extra computational power.

State-of-the-art parallel SAT solver use multiple instances of a single threaded (sequential) solver to explore and solve different subproblems in parallel. The main differentiator between different parallel solver is how their single threaded solver are configured, what information they get to share and how they share this information.

First, this section will explore the main techniques for parallel SAT solving. Finally,

we take a closer look at two specific parallel SAT solving engines *HordeSAT* [4] and *Mallob* [33].

2.3.1 Search Space Splitting

Search Space Splitting is the classical approach to parallel SAT solving and uses a divide-and-conquer approach strategy. It divides the search space of a propositional formula into multiple disjoint parts and distributes them among multiple workers. If a worker finishes with a satisfiable solution, we know the entire input formula must be satisfiable. To prove unsatisfiability we need to wait for all workers to finish with an unsatisfied result.

On the surface, this partitioning of the search space into n parts, one might expect a speedup of n. However, the difficulty of the different subproblems varies drastically and it is hard to predict how long a single subproblem will take to be solved. To ameliorate this issue dynamic work stealing is used to supply workers with additional work if they are idle. The most common technique utilizes a *guiding path* which was first introduced in the parallel DPLL solver PSATO [35]. A master-slave architecture is used, where the slaves are running a sequential SAT solver on a given part of the search space, while the master organizes the search space splitting and load balancing between slaves.

The master process maintains a binary search tree where the nodes represent a subproblem of the original formula. Each node splits the subproblem it represents by selecting an unassigned variable v. Both assignments of v define two new smaller subproblems as its children where v is assigned true or false respectively. Therefore, each leaf is defined by a partial assignment which can be derived from the path beginning at the root node to the leaf. This path is called the guiding path and is passed to the slave which starts solving this particular subproblem. If a slave is idle, but no remaining subproblems are left unassigned the master process can split an unsolved subproblem to generate new work for the idle slave and reducing the subproblem for the slave whose subproblem was split.

Besides using guiding paths, alternatives to split up the search space use *scattering functions* [19] and *xor scattering* [31]. A more modern variant of search space splitting is called *cube-and-conquer* which uses a *lookahead* solver [7] to generate partial assignments which are solved by CDCL solvers. Lookahead solvers have been shown to be effective at solving certain problem classes but do not generalize well to large industrial instances. The cube-and-conquer approach aims to combine the strengths of both lookahead and CDCL solvers. Using a specialized lookahead heuristic a cube-and-conquer solver was able to disprove the Pythagorean triples problem [18].

2.3.2 Parallel Portfolio

The parallel portfolio approach to parallel SAT solving also uses multiple sequential SAT solvers running in parallel. Each solver is configured with different parameters. The performance of a SAT solver on a given problem can change drastically based on the specific

configuration on parameters. Therefore, if multiple solver configurations represent a set of orthogonal strategies they can search different parts of the search space in parallel. As an example consider that fast restarts generally benefit the solving of unsatisfiable formulas but penalize satisfiable instances [28]. A very simple portfolio solver with two cores therefore might run two sequential solvers, one with a fast restart mechanism and the other with few restarts in order two optimize both of these cases. All the configuration options and how they are selected is called the *diversification* strategy. ManySAT [15] selects, as part of its diversification strategy, between different restart policies, decision heuristics and clause learning schemes. It is possible to simulate the search space splitting approach in a portfolio solver by specifying its guiding path as a phase. The selection heuristics of the SAT solver will then select the assignments in the guiding path and lead the solver to this specific subproblem. Thus, portfolio can be viewed as a generalization of search space splitting. Even without any strategy to ensure disjointedness of search spaces, the overlap between solvers in a pure portfolio setting appears to be small [4].

2.3.3 Clause Sharing

Sequential SAT solver learn clauses to prune parts of their search tree. In a parallel setting, the different sequential solvers can share their learned clauses with each other so that they can benefit from each others work. Clause sharing between different solvers, therefore, allows a parallel SAT solver to solve a formula faster than any of its sequential solvers could solve alone. However, to avoid exponential blow-up and overwhelming the unit propagation procedures it is important to limit the amount of shared clauses.

A common heuristic is the size of a learned clause in order to decide if the clause is learned. Mallob [33], for instance, shares the smallest clauses up to a certain amount that contain 8 or fewer literals by default. Alternative heuristics use the *literal blocks distance* (LBD) [34] instead. LBD have proven effective at predicting the usefulness of a clause within a single threaded solver. The benefits of LBD in parallel SAT solving are less clear. This is further complicated by the fact that the concrete LBD value depends on the solver state and can not always be determined for other solvers.

When a solver imports a clause learned by a different solver, it needs to incorporate this clause into its own clause database. Depending on the complexity of its internal data structures this can be non-trivial. Solvers using the common two-watched-literal data structure [27] for efficient unit propagation and backtracking need to take care not to invalidate their invariants when adding a foreign clause to their database. Alternatively, they can only import clauses when the current decision level is zero. For example, immediately after a restart. Most parallel solvers with clause sharing, batch the learned clauses in a shared buffer and only import after a certain time period has passed to avoid excessive overhead from communication between different cores.

2.3.4 HordeSAT

HordeSAT[4] is a massively parallel portfolio SAT solver, scalable up to thousands of cores. Its architecture treats each solver as a black box that satisfies a minimal interface. This interface is split into three parts: (i) starting and interrupting, (ii) diversification and (iii) clause sharing. State-of-the-art solvers are well suited to implement this interface, which makes it easy to add new solvers. The exact details on what settings and behavior diversification affects is left up to the individual solvers.

Multiple MPI [13] processes each manage several solver instances. Following the initialization of each solver they are diversified. Each MPI process maintains a buffer which collects the learned clauses by its solvers. Periodically, this buffer is exchanged with all other MPI processes using the `allgather` MPI operation and the received clauses are passed to each solver. In order to reduce shared duplicate clauses, HordeSAT utilizes several Bloom filters. In addition to a global Bloom filter, every MPI process also maintains a Bloom filter for each solver. The solver specific Bloom filter remember which clauses have been learned by their solver. The global Bloom filter maintains the set of clauses shared. Every newly learned clause is first added to the solvers Bloom filter. If it is neither too large for it to be shared nor a duplicate, it is added to the global Bloom filter and clause buffer.

The decentralized nature of HordeSAT without a central node managing the search or communication, simplifies the implementation and allows for easy scalability over up to thousands of nodes.

2.3.5 Mallob

Mallob [33] builds upon HordeSAT principles. Besides having a performant parallel SAT solving engine, Mallob in general is a work scheduling system with sophisticated load balancing mechanisms. As part of this thesis we will focus on Mallob as a parallel SAT solver and describe its difference compared to HordeSAT. Mallob structures its processing elements as a binary tree, called *jobtree*. Each processing element can execute one or more SAT solvers depending on the configuration and hardware capabilities. Whereas HordeSAT used an `allgather` operation to share learned clauses between all participating solvers, Mallob utilizes its jobtree for communication. Each node in the jobtree maintains a buffer of clauses learned by its solvers.

In regular intervals, the learned clause buffers are aggregated along the jobtree. Each non-leaf node performs a three- or two-way merge on the (aggregated) buffers of its children based on the size of the clauses. This strategy does not leave any gaps in the clause aggregation buffer that usually occur during the concatenation of buffers in HordeSATs implementation. Additionally, all literals within a clause are sorted, thereby allowing the filtering of duplicate clauses immediately by comparing the current with the previous clause during the merge. This strategy significantly reduces the memory sent between different nodes since it does not send any unused memory. The aggregation buffer size for a node

does not need to grow linearly to the number of participating solvers. Furthermore, within a processing element a dynamic data structure is used to collect learned clauses before they are aggregated and shared with other solvers. This data structure keeps smaller clauses and, if necessary, discards larger clauses when the available memory runs low. The same data structure is used for imported clauses so that the best clauses are kept if a solver takes a long time before importing again. Finally, Mallob modifies the clause filtering strategy used by HordeSAT. It omits the global clause filter per processing element as it already performs deduplication as part of the clause aggregation process. However, they complement the solver specific filter by adding an exact set in order to filter unit clauses.

Mallob has proven its effectiveness by winning every the cloud track of every SAT competition it participated [12, 16, 17].

3 Related Work

In this chapter we will explore alternative approaches to clause sharing in a parallel SAT solver.

3.1 GPUShareSAT

Like HordeSAT and Mallob, *GPUShareSAT* [32] is also a parallel portfolio SAT solver. Its main feature is utilization of a GPU to improve clause sharing as its name implies. It is difficult to exploit the massive parallel computational capacities of GPUs effectively for solving the Satisfiability problem. Current CPU based implementations for SAT solvers can not be easily translated to run on a GPU. Not only is the available memory and cache for an individual core much lower, for maximal efficiency the different cores of a GPU need to execute the same execution path. This is problematic for DPLL/CDCL style solvers with bespoke data structures which usually perform highly non-uniform computations. Previously, GPUs were used to aid in unit propagation [9], pre-processing [29] and survey propagation [25] which, individually, are more suited to the architecture of a GPU. GPUShareSAT uses the GPU to detects clauses that might be useful for a solver S based on an analysis of previous trails of S that were sent to the GPU.

The architecture of GPUShareSAT consists of several solver threads which send their trail and learned clauses to a single shared master thread. The trail is the variable assignment of a solver at its current state. The master thread communicates with the GPU in order to quickly determine which clauses might be useful for a solver based on its sent trails. To determine which clauses are useful the GPU finds the clauses that trigger an assignment. GPUShareSAT operates under the common assumption that clauses which would have been useful in the recent past are likely to be useful in the near future. Therefore, a clause triggering an assignment implies that this clause would have been useful to the solver who sent the assignment. Once it receives this clause, we expect an above average probability that this clause will cause a conflict or unit propagation as the solver progresses.

The amount of data generated by the individual solver instances is sufficiently large to overwhelm the GPU. Consequently, only the trails after propagations which did not result in a conflict are shared. For all other trails more clauses would not have helped the solver in making more progress since the solver would have triggered just the same. Additionally, the GPU makes sophisticated use of bit-instructions and pooling of assignments to speedup the detection of triggering clauses on the GPU.

3.2 Progress Saving Based Quality Measure

Progress Saving [30], also often referred to as Phase Saving, describes the technique where the last true/false assignment of each variable is remembered as P. The value of P does not change after unassigning a variable. Originally [30], its intended use was within the decision procedure to select the assignment for the new decision variable v as $P(v)$. Progress saving speeds up solvers using non-chronological backtracking and restarts. Such solvers often encounter the same subproblem repeatedly and therefore will often find its solution within P if the same subproblem was solved previously. While progress saving has been widely adopted by popular SAT solver within their decision procedure, its proven useful for other areas as well.

The authors of [2] use P to determine how useful a learned clause is likely to be. They define a *progress saving based quality measure* as $psm_P(c) = |P \cap c|$. $psm_P(c)$ counts the number of literals in c that are true in the memorized polarities of P. A low $psm_P(c)$ value indicates that c is likely to be useful, considering that triggering clauses have a psm of one or zero. psm is used in [2] as part of a dynamic clause management policy where clauses with a high psm are frozen and temporarily removed from the active clause database used in unit propagation. If the psm of a frozen clause decreases below a certain threshold it is added again. Finally, the idea is extended for use in a parallel SAT solver [1], where imported clauses are evaluated based on their psm and frozen for future use if it is too high. Note however, that the exporting solver still use a global measure like size or LBD to decide which clauses to export in the first place to keep the total communication volume at a manageable rate.

3.3 Lazy Clause Exchange

Lazy Clause Exchange is a clause sharing technique proposed in [3]. Contrary to our and GPUShareSATs approach it does not rely on the trail of a sequential solver to determine usefulness. Instead, they make the following observation: For an unsatisfiable formula, only around half of all learned clauses are used in the final proof for unsatisfiability. Naturally, this value varies between different clause database reduction strategies but appears to remain around 50%. In this sense only around half of all learned clauses are useful. Moreover, only around a third of all learned clauses are seen multiple times during all conflict analyses and even less during conflict analyses that do not follow immediately. The paper suggests that these clauses are interesting enough to be shared with other solvers. This observation motivates the name „Lazy Clause Exchange“ as clauses are only shared lazily once they have been seen multiple times. In addition, imported clauses are not used within the search immediately. Instead, imported clauses are placed in a separate clause database where only conflicts are detected which reduces overhead. Only once an imported clause is falsified does it get promoted to where it is watched for unit literals.

3.4 Control-based Clause Sharing

Variable State Independent Decaying Sum (VSIDS) is a popular branching heuristic for modern SAT solver that measures the activity of every variable. During every conflict resolution, the activity of every encountered variable is increased. In regular intervals, the activity of all variables is decreased by a factor. VSIDS has proven itself to be an effective heuristic to decide the next decision variable. Naturally, its viability for branching has inspired other uses of this heuristic.

In [14] the authors use VSIDS to evaluate the relevancy of clauses imported from other solvers. However, instead of only filtering for relevant clauses, they choose to increase the throughput of clauses from solvers that, on average, have sent relevant clauses. Likewise, the throughput is reduced for solvers that have not been sending relevant clauses.

4 Stability-based Clause Weights

In this chapter we describe our approach to clause sharing in parallel SAT solving. We begin by introducing the concept of literal stability and how this information can be efficiently gathered by a SAT solver using trail sampling. The second part of this chapter focuses on how literal stability is used by a range of clause weighing heuristics to determine the usefulness of a clause for a sequential solver participating in the parallel solving process.

4.1 Literal Stability

During the solving procedure for a SAT problem, variables are assigned one of three values: true, false and unassigned. Depending on the assignment distribution of a variable, we can draw conclusion about its behavior in the future.

The notion of literal stability was first introduced in [20], where it was used to select better blocking literals. The authors were able to show improved performance on cryptographic problem instances. While very similar, our definition of stability differs in that we do not count the number of times a variable has been true, false and unassigned since starting the solving procedure. Instead, we determine the statistical probability that a variable is assigned a certain value in recent history and thereby predict that in the near future it will have a very similar probability distribution. Contrary to [20], we choose the exponential moving average (EMA) when accumulating the assignment values for a variable, in order to gain an understanding how the variable behaved in recent assignment history.

To naively measure the stability values of each literal, the assignment of each variable is sampled in regular intervals. This requires the notion of time within a SAT solver. A naive choice for time measuring uses the real time and therefore samples every few milliseconds. This definition of time is difficult to justify as it is unnecessarily non-deterministic and therefore can hurt reproducibility. Instead, it is better to choose a regular event, called *epoch*, within the CDCL solver loop. We choose the conflict as the unit of time. We will further justify this choice in Section 4.2.

Given this notion of an epoch, we define T_i to be the set of all literals on the assignment trail during the i-th epoch. Since a variable can only ever be assigned the value true or false, either the positive or negative literal can appear in T_i and never both. Using T_i we define three stability values mirroring the three assignment states. $S_i^t(v)$ describes the probability

that the variable $v \in V$ is true in recent sampling history. In analogy, $S_i^f(v)$ and $S_i^u(v)$ behave the same and correspond to the probability that v is false or unassigned.

S_i^t, S_i^f and S_i^u are defined as follows:

$$S_i^t(v) = \alpha\ \sigma(v \in T_i) \qquad + (1-\alpha)S_{i-1}^t(v) \tag{4.1.1}$$

$$S_i^f(v) = \alpha\ \sigma(\neg v \in T_i) \qquad + (1-\alpha)S_{i-1}^f(v) \tag{4.1.2}$$

$$S_i^u(v) = \alpha\ \sigma(v \notin T_i \wedge \neg v \notin T_i) + (1-\alpha)S_{i-1}^u(v) \tag{4.1.3}$$

This definition incorporates the usual definition of the exponential moving average. α is the EMA smoothing coefficient and σ an indicator function defined over a predicate. If the provided predicate is true, σ is 1, otherwise it is 0. The base case $S_0^{\{t,f,u\}}$ and biasing of the EMA is ignored for now as an implementation detail. As would be expected: $S^t + S^f + S^u = 1$ and $0 \leq S_i^t, S_i^f, S_i^u \leq 1$. This implies that it suffices to only collect two of the three stability values. $S_i^{\{t,f,u\}}(v)$ is the function of stability values for each variable which we extend over all literals.

$$S_i^t(l) = \begin{cases} S_i^t(l) & l \in V \\ S_i^f(\bar{l}) & \bar{l} \in V \end{cases}$$

$S_i^f(l)$ is defined analogously, while $S_i^u(l) = S_i^u(|l|)$. Where applicable, we will omit the epoch i and literal l if it is not needed or implicit in the current context.

We use the exponential moving average to aggregate the stability information for each variable because we are interested in the local behavior of a variable. As the solver explores different subproblems (communities) we expect the stability of variables to change. The assignment distribution of different subproblems does not remain relevant and should be forgotten to avoid unnecessary noise in the stability information. Moreover, the exponential moving average can approximate the cumulative average by selecting an arbitrarily small smoothing coefficient in case this assumption turns out to be incorrect. In addition, by multiplying our definition of stability with the total number of epochs we arrive at approximately the same notion of stability as defined in [20].

As an example how stability can inform us about the behavior of variables let us consider a hypothetical solver execution. Figure 4.1 shows the trail length over time for a solver. Because the trail behaves like a stack, every entry on the trail length axis represents an assigned literal. Literals on the trail are always assigned to be true, otherwise their complement would be used instead. Only if the trail shrinks to below the entry of a literal can this entry change to store a different, newly assigned literal. It follows, that a literal l which is assigned earlier and is therefore lower in the trail will gradually increase its $S^t(l)$ as it gets repeatedly sampled. Whereas literals that are higher in the trail will be unassigned and changed more often which results in a less stable S^t. The colored areas in Figure 4.1 indicate literals that are assigned at the end of the plot. The literals within the lowest (blue) area are continuously assigned for the longest time which will be reflected in their S^t value.

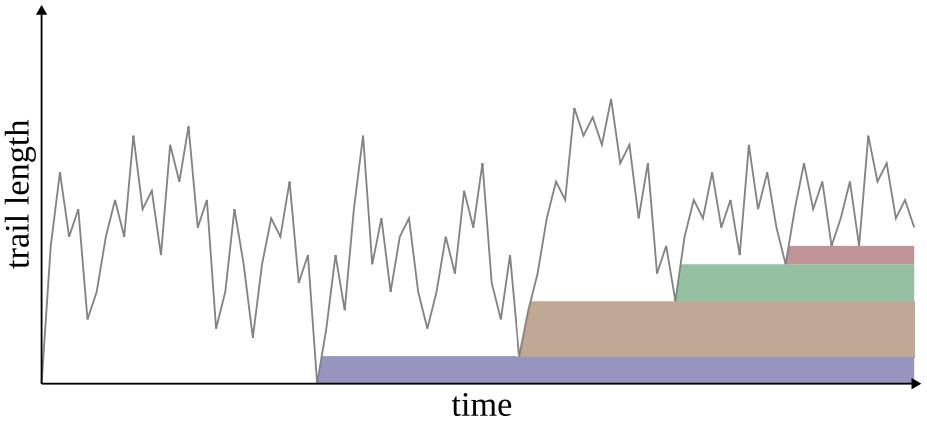

Figure 4.1: Plot of the trail length over time for a hypothetical solver execution. The colored bars indicate assigned literals that have not changed since their assignment.

As we move up the different colored areas, their contained literals generally will have decreasing S^t values. Naturally, due to the unpredictable nature of SAT solving, it is possible that certain literals are often repeatedly assigned and unassigned in a short timeframe. Such literals will have a higher S^t value even though they are recently assigned on the trail. This simple analysis of stability based on trail position is further complicated in state-of-the-art solvers which use restarts and phase saving to improve their performance.

4.2 Trail Sampling

Conceptually, the stability information is collected by inspecting the assignment value of each variable in regular intervals. We choose every conflict to perform trail sampling. Inbetween two conflicts the trail is only appended with new literals. These literals are either inferred by unit propagation or they are decision literals. Blocks of propagated literals are separated by a single decision literal. After each decision, the unit propagation engine finds all literals, that are implied by the new decision in addition to all previous literals on the trail, and appends them to the trail. Within such a block of propagated literals the order is mostly arbitrary and an implementation detail of the solver. Sampling multiple times while such a block is in the process of being appended would bias the stability of literals that appended earlier. Additionally, there is a difference of several magnitudes in the number propagations compared to the number of conflicts. Sampling every propagation would hurt performance significantly. Sampling less frequently than every conflict can miss the assignment a variable entirely reducing the accuracy of the gathered data. Only after a conflict occurs does backtracking remove literals from the trail. We therefore decide to sample the trail during every conflict, before any literals are removed so that they sampled at least once.

A naive implementation of trail sampling might iterate *every* variable on every conflict.

For each variable it updates its stability value. We must not only observe the literals on the trail because unassigned literals affect the stability value as well by decreasing both S^f and S^t. As described in Section 4.1 we are interested in three values S^t, S^f and S^u. We need to sample at least two of these three values to calculate the third. In order to correctly calculate these values, unassigned variables (i.e. variables that do not appear on the trail) need to be updated as well.

The performance of this naive implementation can be surprisingly good, which can be explained by beneficial cache behavior where the assignment value of each variable is stored in a compact array. However, it still is linear in the number of variables. In Subsection 4.2.3 we will present an implementation that runs significantly faster. This faster implementation requires that we can update an exponential moving average repeatedly with the same value efficiently. Therefore, before we can explain the faster implementation we need to derive an unbiased exponential moving average with this capability.

4.2.1 Exponential Moving Average with Cumulative Correction

In this section we derive an unbiased exponential moving average which supports updating the current EMA with the same value k times in $O(\log k)$.

The common formula for the exponential moving average is

$$E_n = \alpha \cdot Y_n + (1 - \alpha) \cdot E_{n-1} \tag{4.2.1}$$

Y is the series of values we are averaging and α the smoothing factor with $0 < \alpha < 1$. Note that this formula is missing its base case.

Substituting E_n in the body of the formula we can determine the series of weights for each value.

$$\begin{aligned} E_n &= \alpha \cdot Y_n + (1 - \alpha) \cdot E_{n-1} && (4.2.2)\\ &= \alpha \cdot Y_n + (1 - \alpha)(\alpha \cdot Y_{n-1} + (1 - \alpha)(\dots)) && (4.2.3)\\ &= \alpha \cdot Y_n + \alpha(1 - \alpha)Y_{n-1} + \alpha(1 - \alpha)^2 Y_{n-2} + (\dots) && (4.2.4) \end{aligned}$$

Given n values, their individual weights W_i^n are

$$W_i^n = \alpha \cdot (1 - \alpha)^{n-i}$$

Based on W_i^n, we derive the recursive formula for bulk updates, where we update with

the value y repeatedly, k times.

$$E_{n+k} = \sum_{i=1}^{n+k} W_i^{n+k} Y_i \tag{4.2.5}$$

$$= \sum_{i=1}^{n+k} \alpha(1-\alpha)^{n+k-i} Y_i \tag{4.2.6}$$

$$= \sum_{i=n+1}^{n+k} \alpha(1-\alpha)^{n+k-i} y + \sum_{i=1}^{n} \alpha(1-\alpha)^{n+k-i} Y_i \tag{4.2.7}$$

$$= \alpha y \sum_{i=0}^{k-1} (1-\alpha)^i + (1-\alpha)^k E_n \tag{4.2.8}$$

$$= (1-(1-\alpha)^k) y + (1-\alpha)^k E_n \tag{4.2.9}$$

$$= y + (1-\alpha)^k (E_n - y) \tag{4.2.10}$$

Since $0 < \alpha < 1$ we can use the closed form of the finite geometric series to get rid of the summation.

Unbiased Exponential Moving Average

In its current state, E_0 remains an issue to calculate the EMA. Simply choosing an initial value x would have the same effect as adding infinitely many x initially before beginning to update with values in Y. The EMA for the first values would be distorted by this initialization bias. Instead, we would prefer that the EMA of few early values is representative of their true average. This is referred to as an *unbiased* exponential moving average.

For example, CaDiCal uses the unbiased EMA presented in the ADAM method [22]. While this method is more accurate than CaDiCals previous implementation it is unsuitable for us because of our bulk update requirement. Therefore, we derive our own bias correction in the following paragraphs.

The exponential moving average assigns a non-zero weight for each value in an infinite sequence of values. Since we are only given a finite amount of values there is a sum of residual weights left. We distribute these residual weights evenly between all values. This uniform distribution is equivalent to the cumulative moving average. Given a sufficiently small smoothing coefficient α for the exponential moving average, we will mostly observe the cumulative averaging between all values to begin with. As the number of values increases the residual weight converges to zero and the weights of the exponential moving average begin to dominate. If we were to ignore the residual weights we would end up with a biased exponential moving average where the early values are distorted. We refer to this cumulative corrected exponential moving average as CEMA.

In order to determine the residual weight R_n given n values we subtract the sum of W_i^n from 1.

$$R_n = 1 - \sum_{i=1}^{n} W_i^n \tag{4.2.11}$$
$$= 1 - \sum_{i=1}^{n} \alpha \cdot (1-\alpha)^{n-i} \tag{4.2.12}$$
$$= 1 - \alpha \sum_{i=0}^{n-1} (1-\alpha)^i \tag{4.2.13}$$
$$= 1 - \alpha \frac{1-(1-\alpha)^n}{\alpha} \tag{4.2.14}$$
$$= (1-\alpha)^n \tag{4.2.15}$$

Finally, we can write the formula for the exponential moving average which corrects its bias using cumulative averaging:

$$E'_n = \sum_{i=1}^{n} \left(W_i^n Y_i + \frac{R_n}{n} Y_i \right) \tag{4.2.16}$$
$$= \sum_{i=1}^{n} W_i^n Y_i + \sum_{i=1}^{n} \frac{R_n}{n} Y_i \tag{4.2.17}$$

This formula consists of the sum of the *exponential part* $E_n = W_i^n Y_i$ and the *cumulative part* $C_n = \frac{R_n}{n} Y_i$.

To efficiently calculate E'_n as new values Y_n become available we need to convert this formula back into a recursive formula. Additionally, we want to be able to add the same value y multiple times efficiently.

We already derived the recursive formula for the exponential part in Equation 4.2.5. Likewise, for the cumulative part, we can also derive a recursive formula for bulk updates:

$$C_{n+k} = \sum_{i=1}^{n+k} \frac{R_{n+k}}{n+k} Y_i \tag{4.2.18}$$

$$= \frac{R_{n+k}}{n+k} \left(k \cdot y + \sum_{i=1}^{n} Y_i \right) \tag{4.2.19}$$

$$= \frac{R_{n+k}}{n+k} \left(k \cdot y + \frac{n}{R_n} C_n \right) \tag{4.2.20}$$

$$= \frac{(1-\alpha)^{n+k}}{n+k} \left(k \cdot y + \frac{n}{(1-\alpha)^n} C_n \right) \tag{4.2.21}$$

$$= \frac{(1-\alpha)^{n+k}}{n+k} \left(k \cdot y + \frac{n+k}{(1-\alpha)^n} C_n - \frac{k}{(1-\alpha)^n} C_n \right) \tag{4.2.22}$$

$$= (1-\alpha)^k \left(\frac{(1-\alpha)^n}{n+k} k \cdot y + C_n - \frac{k}{n+k} C_n \right) \tag{4.2.23}$$

$$= (1-\alpha)^k \left(C_n + k \frac{(1-\alpha)^n y - C_n}{n+k} \right) \tag{4.2.24}$$

As formulated in Equation 4.2.16, the sum of the exponential E_n and cumulative part C_n result in the final value. An implementation of this unbiased exponential moving average needs to store at least three values. The time n which remembers how many values were already aggregated, the exponential and cumulative part.

Because Equation 4.2.5 and Equation 4.2.18 are recursive formulas, we need to define the initial values for C_n and E_n. We define $C_0 = 0$ and $E_0 = 0$. To justify this decision, consider the expected result of C_1 and E_1. When only adding a single value x, we expect the EMA value to equal x: $E'_1 = C_1 + E_1 = x$. In the exponential part, x is weighted by α as is implied by formula for W_1^1. Therefore, the cumulative part needs to apply $1 - \alpha$ as weight to x in order for the sum of both result in x. This is only the case if $C_n = 0$. Finally, it can be shown that for every n the sum of all weights of C_n and E_n add up to one.

4.2.2 Efficient Trail Sampling

In Section 4.2 we described how a naive trail sampling implementation looks like. While simple, this method requires an $O(n)$ operation every conflict. This subsection describes how we can improve upon that implementation to reduce the performance costs.

The core idea is to only update the stability values of a variable when its assignment is modified. If a variable v changes its assignment from a to b, where $a, b \in \mathbb{S}$, we repeatedly update the stability values for v with a repeatedly k times, where k is the number epochs (i.e. conflicts) that occurred between when v was assigned with a and the current time. For the most part, assignments of variables change only during propagation, decision and

backtracking. To facilitate such an update, every variable remembers the epoch of when it was last changed. Once the assignment changes again we can calculate the difference in epochs and update the stability values accordingly. To efficiently update the EMA of a variables stability value we use the CEMA implementation described in Subsection 4.2.1. Algorithm 1 defines the function UPDATEVARIABLE which shows the corresponding pseudocode.

The type StabilityData stores the last epoch a variable was changed previously together with the CEMA data to store S^f and S^t. After updating the last epoch of the input variable v and calculating the number of elapsed epochs, both CEMA for true and false are updated. In the algorithm, we once again use σ as the indicator function.

The correct use UPDATEVARIABLE requires it to be called every time *before* the assignment of a variable changes, as it needs the original assignment. Finally, every time the gathered stability information is used in order to calculate clause heuristics (Section 4.4), we need to synchronize the epochs of all variables by calling UPDATEVARIABLE beforehand. Otherwise, variables that have not changed for a significant amount of time, will not reflect their current assignment in their stability value.

Algorithm 1: UPDATEVARIABLE($v : V$)

```
1 Type: StabilityData = (lastEpoch: ℕ, trueStability: CEMA, falseStability: CEMA)
  Data: stability: V → StabilityData
  Data: assignment: V → 𝕊
2
3 oldEpoch ← stability[v].lastEpoch
4 currentEpoch ← GETCURRENTEPOCH()
5 stability[v].lastEpoch ← currentEpoch
6 epochDiff ← currentEpoch − oldEpoch
7
8 stability[v].trueStability.UPDATEBULK(v, σ(assignment[v] = T), epochDiff)
9 stability[v].falseStability.UPDATEBULK(v, σ(assignment[v] = F), epochDiff)
```

The authors of [20] use a similar strategy to efficiently collect stability information. The main difference is that they are collecting the sum of epochs where a literal is true. This is equivalent to aggregating the cumulative moving average and significantly simplifies the formula.

4.2.3 Optimized Implementation

The straightforward implementation of these formulas leaves room for performance improvements which are described in this subsection. The first three tips focus on the implementation of CEMA, while the last two are tips to improve the efficiency of the entire trail sampling algorithm.

The cumulative part requires the value $(1-\alpha)^n$ where n is the total number of values added to the CEMA. Calculating this value each time is undesirable since it becomes increasingly expensive over time. The implementation should instead store this value, so it only needs to multiply it by $(1-\alpha)^k$ each time k new values are added.

The cached $(1-\alpha)^n$ monotonically decreases and will eventually reach and remain zero due to floating point imprecision. Once it and C_n are zero, the cumulative part will remain zero forever. Therefore, we can skip the cumulative formula avoiding the expensive division and multiplications. The branch predictor on modern CPUs can almost entirely eliminate the costs for the cumulative part at this point. It is important to check both the cumulative part and $(1-\alpha)^n$. If either one is larger than zero it will influence the new C_{n+k}.

The α parameter is the same for each CEMA of each variable. Therefore, it need not be stored inside the CEMA implementation. It can be passed to the corresponding procedure every time a CEMA is updated. Removing α reduces the size of the CEMA structure and therefore improves cache efficiency.

We can reduce cache misses further by placing the time, true- and false-stability values next to each other in memory for each variable (array-of-structs). If these three values are stored in separate arrays for each variable (struct-of-arrays) they incur up to three cache misses if their variable changes. On the other hand, when placed next to each other they are likely placed on the same cache line.

Both the cumulative and exponential part require the factor $(1-\alpha)^k$ where k is the number of epochs since the last update. We observe that the order of variables on the trail coincides with their respective assignment epoch. We can make use of this behavior during backtracking, by starting with the most recently assigned variable v_1 with an epoch difference of k_1 and calculating $(1-\alpha)^{k_1}$. The next variable v_2 on the trail has an epoch difference of k_2 where $k_1 \leq k_2$. Therefore, we can calculate $(1-\alpha)^{k_2}$ by reusing $(1-\alpha)^{k_1}$ by multiplying it with $(1-\alpha)^{k_2-k_1}$. We can continue this trick for the entire backtracking process. In the actual implementation we always cache the most recent $(1-\alpha)^k$ result independent of whether the solver is currently backtracking. Any subsequent updates where the new epoch difference is larger the k use the same cheaper calculation. Backtracking causes around half of all assignments. Even besides backtracking on assignments to true or false the probability that a new epoch difference is larger than the previous is around 50% and can, therefore, make use of this optimization.

4.3 Clause Weights

A clause sharing parallel SAT solver will instruct its children to share their learned clauses with each other. Not every received clause is useful for a sequential solver and accurately determining whether a specific clause will help a different solver make progress is difficult. Therefore, we rely on heuristics that approximate the usefulness of a clause.

Usually, parallel SAT solvers are careful with the total amount clauses shared as it can have detrimental effects unit propagation performance. For our approach we increase the number of shared clauses, but the individual sequential solvers do not import all of them. Instead, they evaluate each clause weight and only import a fraction of the clauses with the best clause weight. In this thesis we focus on heuristics that use literal stability.

A good clause weight heuristic is expected to improve the performance of the parallel SAT solver by selecting clauses that are more likely to cause a conflict or trigger unit propagation, thereby furthering the progress of the entire solving process. The justification of each of the presented heuristics will involve some reason why this particular heuristic might improve the rate at which conflicts or unit propagations occur.

A heuristic is a function $h : C \to \mathbb{R}$ which maps a clause to a real number. Whether a higher or lower evaluation of a clause by a heuristic indicates usefulness is left to the definition of the heuristic.

Regarding evaluation of clauses our approach is similar to GPUShareSAT [32] which was introduced in Section 3.1. Whereas GPUShareSAT collects and maintains a discrete set of recent assignments from a solver, we statistically aggregate recent assignments. The GPUShareSAT approach is more precise as it can always accurately determine if a clause would have triggered under a recent assignment. Our approach instead can only make an approximate guess based on if the literal stability lines up. As multiple assignments are merged, details can get lost.

Consider an example with two variables v_1 and v_2. Clauses encode that both variables have to be equal: $\{v_1, \overline{v_2}\}$, $\{\overline{v_1}, v_2\}$. During the CDCL loop, they switch between the true and false assignments. Therefore, the S^f value for each variable might be around 0.5. Note that the both variables have the same stability as unit propagation ensures the same assignment before the next conflict where their assignments are sampled. GPUShareSAT remembers some exact trails where the variables are both true and other trails where they are both false. Now, let us consider what happens when the clause $\{v_1, v_2\}$ is learned. This is a very useful clause as it proves that both v_1 and v_2 have to be true.

Our approach can only see that both variables have changed their assignment recently. It does not matter whether v_1 and v_2 are always equal or always different their stability information looks the same. Whether the clause would be imported is up to the exact heuristic, but more fundamentally no heuristic can make a distinction here. GPUShareSAT, on the other hand, can detect the conflict with a trail where both variables are false and import it. The core issue is that stability does not retain any information about the relationship between different variables. Here, it can not realize that the new clause is effectively a unit clause. In GPUShareSAT such relationships are implicitly encoded inside the discrete trails. This example is very small and, in practice, both methods are almost guaranteed to import this clause. Yet we consider it sufficient to illustrate a core difference in approach.

The benefit of our approach is simplicity in implementation, not requiring additional hardware. We incur performance overhead as each sequential solver needs to maintain its own stability and perform clause evaluation, which is outsourced to the GPU by GPUShare-

SAT. We do not require a master thread for coordination, which allows this approach to scale more easily to an arbitrary number of sequential solvers, a property we inherit from Mallob.

4.4 Clause Weight Heuristics

In this section we present the heuristics that were explored as part of this thesis and why they were chosen. This section does not claim that these heuristics do improve solver performance. Their evaluation is discussed later in Section 5.3.

Size

The size heuristic is the default heuristic used by most parallel SAT solver.

$$h_{\text{size}}(c) = |c|$$

It is easy to determine and independent of the solver that learned the clause. Additionally, it is easy to justify why smaller clauses are more useful than larger clauses. Smaller clauses cover larger parts of the entire search space. They are in general more likely to trigger as, per definition, fewer literals need to be assigned false. Since the size heuristic is also the default for the Mallob parallel SAT solving engine we use this heuristic as the baseline for the experiments in this thesis.

Product

The product heuristic is the product of all S^f values of each literal:

$$h_{\text{product}}(c) = \prod_{l \in c} S^f(l)$$

The $S^f(l)$ can be viewed as the probability that l is false in the recent history. In order to determine the probability that a given clause would trigger we multiply the probability that each literal is false. For this heuristic a higher value represents a clause that is more likely to trigger. However, this assumes that each $S^f(l)$ is pairwise independent which is not true. For example, consider the following three clauses $\{\neg x, y\}, \{x, \neg y\}, \{x, y, z\}$. The first two clauses guarantee that x and y are always assigned to the same value due to unit propagation. Therefore, if z is independent of x and y, the product heuristic would result in $S^f(x) \cdot S^f(y) \cdot S^f(z)$ while the correct probability is $S^f(x) \cdot S^f(z)$. Despite this limitation, it may have turned out that the actual error introduced by this false assumption is not super relevant in practice, which is why we added it to the set of heuristics to explore.

Mean

The mean heuristic calculates the average of the S^f of each literal:

$$h_{\text{mean}}(c) = \frac{\sum_{l \in c} S^f(l)}{|c|}$$

The mean heuristic averages out the likelihood for each literal to be false. Naturally, a higher rating implies a that the given clause is more likely to trigger under the current assignment.

A notable feature of the mean is heuristic that its evaluation is independent of the clause size. For example, unlike the product heuristic, where each additional $S^f(l)$ almost always decreases its overall rating because $S^f(l) \in [0, 1]$.

Lukasiewiecz

This heuristic is based on the equally named T-norm used in fuzzy logic.

$$h_{\text{lukas}}(c) = \max\left(\sum_{l \in c} S^f(l) - |c| + 1, 0\right)$$

Usually a T-norm is defined as a binary operator. We can generalize it to an n-ary function because T-norms are required to be commutative and associative. In order for the Lukasiewiecz norm to be larger than zero, the sum of all S^f values must be larger than $|c| - 1$. This is a strong requirement and there is little doubt that such clauses are likely to trigger. However, all clauses that do not meet this bar are considered equally bad with an evaluation of zero.

Minimum

This heuristic evaluates to the lowest false stability value of all literals in a clause.

$$h_{\text{min}}(c) = \min_{l \in c}(S^f(l))$$

The lowest $S^f(l)$ serves as an upper bound on the probability for the actual probability of the clause to cause a conflict. l is the literal that is most likely to not be false. Therefore, we can use it to filter clauses that have a literal that is very unlikely to be false. Like the mean heuristic, the minimum heuristic discards the size information entirely when assessing a clause.

Second Minimum

The second minimum is the second-lowest value of a set. Similar to the minimum heuristic we define this heuristic to be the second minimum of the false stability values of each literal in the clause.

$$h_{\text{2min}}(c) = \min_{l_0 \neq l_1}(\max(S^f(l_0), S^f(l_1)))$$

Instead of optimizing for conflicts like the minimum heuristic, the second minimum additionally allows unit propagations. If all but one literal l are likely to become false in the near future, this will cause unit propagation to assign l. Such clauses might be completely disregarded by the minimum heuristic. On the other hand, if l is consistently true, the usefulness of this clause drops without changing its weight under second minimum heuristic. Per definition, the second minimum heuristic will evaluate all clauses better than the minimum heuristic.

Weighted Literal Score

The Weighted Literal Score (WLS) heuristic is a generalization of the size heuristic incorporating stability information.

$$h_{\text{WLS}}(c) = \sum_{l \in c} \left(S^u(l) + S^t(l) \cdot \omega + S^f(l) \cdot 0 \right)$$

Each literal is evaluated and given a score in isolation. The sum of all literal scores is the resulting clause weight. To motivate the heuristic consider the three extreme cases of stability for a literal. If a literal has never been assigned, we lack any knowledge about its behavior. In this case we score it with a value of one. Indeed, if all literals have never been assigned, for example, at the very beginning, this heuristic evaluates to exactly the size. If a literal is always false we can pretend the clause does not even contain this literal and its size is therefore reduced by one. On the other hand, if a literal is always true, we can not expect the clause to trigger. We therefore discourage this clause by penalizing the entire clause weight by multiplying S^t with ω where $\omega \geq 1$. We refer to ω as the *true literal penalty*. This heuristic has an immediate correlation with the size heuristics, since it is monotonically growing with the number of literals in a clause.

Discrete Weighted Literal Score

The Discrete Weighted Literal Score (DWLS) is similar to WLS.

$$h_{\text{DWLS}}(c) = \sum_{l \in c} \left(\delta_{\tau_t}(S^t(l)) \cdot \omega + \delta_{\tau_f}(S^f(l)) \cdot 0 + (1 - \delta_{\tau_t}(S^t(l)))(1 - \delta_{\tau_f}(S^f(l))) \right)$$

δ_α is the indicator function whether the argument is greater than α. The justification for the Weighted Literal Score and Discrete Weighted Literal Score are similar. The difference is that the Discrete Weighted Literal Score makes a binary choice on the role a literal serves. Instead of continuously interpolating between the three stability values for each literal, DWLS categorizes each literal and then applies its respective score. To understand the definition of the Discrete Weighted Literal Score consider three cases.

(i) $S^t(l) > \tau_t$: The literal has been mostly true in recent history, therefore we weight this literal with the true literal penalty ω.

(ii) $S^f(l) > \tau_f$: The literal has been mostly false in recent history, therefore we act like the clause doesn't contain this literal and give it weight of zero.

(iii) else: Finally, if the literal is neither mostly true nor mostly false, it acts as a literal without a clear behavior. Mostly unassigned literals fall into this category. These literals get weight a of one.

Each of the summand in the formula correspond to one of the cases in order. We refer to τ_t (τ_f) as the *true stability threshold* (*false stability threshold*).

5 Evaluation

We now turn to evaluating our work. We pose three question which we will try to answer empirically in the subsequent sections.

(i) What are the performance implications of trail sampling?

(ii) What are the best parameters for the heuristics?

(iii) How well do the heuristics perform?

The last question will be split into a preliminary benchmark which compares all heuristics on a weaker machine and a benchmark on the SuperMUC NG supercomputer which explores the best heuristics using higher parallelization.

Trail sampling has been implemented inside the CaDiCal [5] SAT solver.[1] We use Mallob [33] as the parallel SAT solver. For this thesis, very few modifications were required to get Mallob to work for our purposes. Indeed, we only added a pass-through mechanism to be able to define configuration flags for CaDiCal which is linked as a static library inside Mallob.

Before we discuss any results, however, we elaborate on our benchmark methodology.

Our benchmark selection consists of 349 SAT problem instances from a total of 400 SAT problems used in the 2022 SAT Competition. The 51 removed instances were removed because no participating solver in the 2022 competition was able to solve them. This reduction significantly reduces the execution time for the benchmarks in this chapter. With a timeout of five minutes, we save more than four hours on each configuration per benchmark run.

We use the PAR-2 score as our performance measure for all benchmarks. The PAR-2 score is calculated as the sum of runtimes for all solved instances plus two times the timeout for every unsolved instance. It follows that for the PAR-2 score a lower score is better. In addition to the PAR-2 score we also show the number of solved instances which is divided into the number of solved SAT instances and UNSAT instances. Based on the definition of the PAR-2 score, there is a strong correlation between the PAR-2 score and the number of solved instances.

Where needed, we distinguish between the heuristics h_{size} and h_{size^*}. h_{size} represents the size heuristic with trail sampling disabled, while h^*_{size} has it enabled.

[1] The code can be found at `https://github.com/neuring/cadical`.

strategy	PAR-2 score	# solved
NAIVE	30996.32	69
OPTIMIZED	14318.99	104
NO-SAMPLING	10867.66	109

Table 5.1: Benchmark results for trail sampling performance. NAIVE is the strategy which samples every variable on every conflict. The OPTIMIZED strategy incorporates all optimizations described in Subsection 4.2.2

.

Evaluation Challenges

The variance of results has proven itself to be a major challenge during evaluation. Without care, results prove to be unreproducible and therefore unreliable. Parallel SAT solving can be highly chaotic in runtime performance. Even the runtime of a single sequential SAT solver can differ drastically, given slightly different starting conditions or small random disturbances during execution, like small differences in the selection of decision variables. This is exacerbated by the non-deterministic nature of a parallel programs, where small differences in scheduling can have a multitude of effects.

In order to counteract this problem, we have chosen several strategies. First and foremost we have increased the benchmark set multiple times until we ended up with the current set. Additionally, we randomize the order of configuration and instances. Concretely, for a benchmark with a set of configurations we want to test, we generate a list of all pairs of configurations and instances and shuffle it. We traverse this list and execute the instance with its configuration in order. This randomization helps to average out any temporal effects on the computer like temperature. Finally, we did not use hyperthreading as it can introduce additional disturbances during execution. However, even with these countermeasures one needs to remain prudent when drawing conclusions from any gathered data.

The issue with variance can be observed in the tuning benchmarks for ϵ (Section 5.2) and τ (Section 5.2). They both include two distinct benchmark runs and their average score is noticeably different.

5.1 Trail Sampling Efficiency

The performance analysis of the optimized trail sampling was executed on an Intel(R) Xeon(R) CPU E5-4640 0 @ 2.40GHz running Ubuntu 20.04 LTS using Linux 5.4.0. For this benchmark we are only interested in the performance overhead of trail sampling. Therefore, we run each instance entirely independent of each other and there is no parallel solver coordinating multiple CaDiCal instances. Indeed, a large benefit over the remaining benchmarks is that a single threaded CaDiCal is deterministic when executed with the same seed.

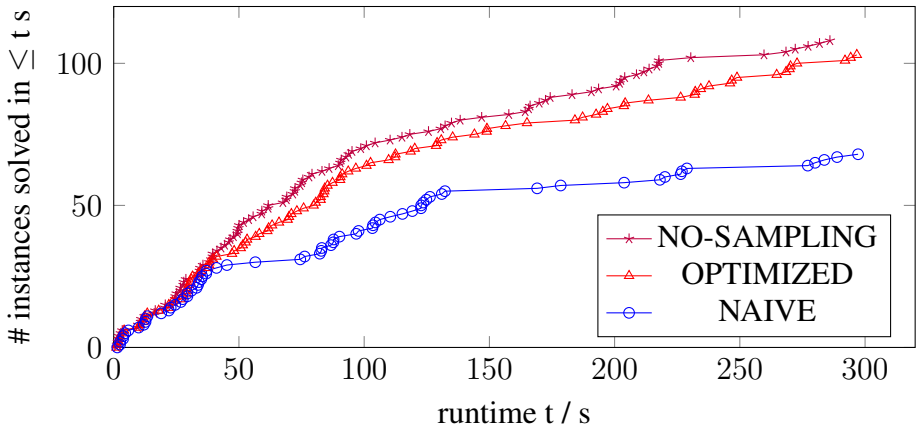

Figure 5.1: Performance of trail sampling approaches.

The results are shown in Table 5.1 and Figure 5.1. We compare three categories. The baseline NO-SAMPLING uses default CaDiCal without any modification on our part. NAIVE is the naive implementation of trail sampling where we sample every variable on every conflict. Finally, OPTIMIZED is the improved trail sampling implementation including all tricks described in Subsection 4.2.2. As is expected, NO-SAMPLING is the fastest category. OPTIMIZED is fairly close with a score that is a third larger, while NAIVE trails behind with a score that is a three times worse than NO-SAMPLING. When we restrict us to the instances that were both solved by OPTIMIZED and NAIVE, OPTIMIZED shows a speedup of 1.51. While NO-SAMPLING only has a speedup of 1.2 over OPTIMIZED.

5.2 Parameter Tuning

In this section we optimize the parameters for the heuristics WLS and DWLS as defined in Section 4.4 and present the results.

Preliminary results have shown that h_{WLS} and h_{DLWS} to be the best performing heuristics. Therefore, we focus on them when tuning parameters. We optimize the following parameters for these heuristics in their presented order: EMA window size ϵ, true literal penalty ω, stability threshold τ and buffer size β.

After a parameter is optimized, the subsequent tuning benchmarks adopt the optimized parameter where applicable. Whenever we are tuning for a parameter with an open-ended domain, we increase the value exponentially. If the parameter domain is bounded we linearly interpolate its domain.

The tuning benchmarks were executed on computer a with two CPU sockets with each containing AMD EPYC 7713 64-Core Processor running Ubuntu 20.04 LTS using Linux 5.4.0. This results in a total of 128 physical cores. Mallob was run with 32 processing units, each running four threads.

	WLS		DWLS	
ϵ	PAR-2 score	# solved (SAT + UNSAT)	PAR-2 score	# solved (SAT + UNSAT)
10^3	84753.70	232 (131 + 101)	83349.67	234 (133 + 101)
10^4	83416.13	235 (132 + 103)	**82218.05**	236 (133 + 103)
10^5	83968.98	235 (130 + 105)	82664.05	234 (134 + 100)
10^6	**82534.23**	**236 (134 + 102)**	82397.68	**237 (132 + 105)**
10^6	**79123.02**	**242 (134 + 108)**	**79870.80**	**241 (133 + 108)**
10^7	79583.44	240 (133 + 107)	80703.70	239 (132 + 107)

Table 5.2: The parameter tuning results for the EMA coefficient. The ema column specifies the tested window size. The results of lower two rows were generated by a seperate benchmark run. Other parameters are: $\omega = 4$, $\beta = 4500$, $\tau = 0.9$

	WLS		DWLS	
ω	PAR-2 score	# solved (SAT + UNSAT)	PAR-2 score	# solved (SAT + UNSAT)
1	80220.38	239 (132 + 107)	81643.63	238 (131 + 107)
2	79217.68	242 (133 + 109)	82294.03	235 (129 + 106)
4	**78461.89**	**244 (136 + 108)**	**79373.19**	**242 (132 + 110)**
8	79872.21	241 (130 + 111)	80419.94	238 (132 + 106)
16	80811.13	238 (132 + 106)	81127.14	238 (132 + 106)

Table 5.3: The parameter turning results of the true literal penalty parameter. Other parameters are: $\epsilon = 10^6$, $\tau = 0.9$, $\beta = 4500$

Tuning EMA

We begin by tuning the EMA parameter ϵ. The values in the EMA column are comparable to the window size in the simple moving average. The EMA coefficient α is calculated using the formula $\alpha = \frac{2}{\epsilon+1}$ [6]. In the first benchmark run we only tried values from 10^3 to 10^6. The results are shown in the first four rows of Table 5.2. For the most part, the best results occur when $\epsilon = 10^6$. Naturally, the question arises if larger values for ϵ improve the results further. In order to answer this question we ran an additional tuning benchmark for the EMA parameter. This time only with the value 10^6 and 10^7. These results are shown in the two lower rows, separated by a horizontal line. Again, $\epsilon = 10^6$ shows the best result. However, it is worth pointing out that all results of the second run are better than all results of first run. This discrepancy highlights the importance to shuffle all individual runs which ameliorate such temporal artifacts and allows for comparison within the same benchmark run.

Tuning True Literal Penalty

We now turn to the true literal penalty ω. As a reminder, the true literal penalty is the factor by which literals that have been true in the recent past are penalized and, thereby,

	DWLS	
τ	PAR-2 score	# solved (SAT + UNSAT)
0.5	81910.86	236 (129 + 107)
0.6	**79606.30**	**241 (134 + 107)**
0.7	81385.81	237 (132 + 105)
0.8	80900.41	238 (131 + 107)
0.9	81140.19	239 (132 + 107)
0.99	80099.90	240 (133 + 107)

Table 5.4: The parameter tuning results of the stability threshold. This parameter is only relevant for the DWLS heuristic. Other parameters are: $\epsilon = 10^6$, $\omega = 4$, $\beta = 4500$

		DWLS	
τ_t	τ_f	PAR-2 score	# solved (SAT + UNSAT)
0.6	0.6	52397.77	287 (143 + 144)
0.6	0.95	50776.60	291 (147 + 144)
0.95	0.6	**50298.02**	**292 (145 + 147)**
0.95	0.95	51353.11	290 (146 + 144)

Table 5.5: Benchmark results for the experiment where the stability threshold is split. Other parameters are: $\epsilon = 10^6$, $\omega = 4$, $\beta = 4500$

discouraging clauses that contain such literals. The results are shown in Table 5.3. Here, the results clearly indicate that a true literal penalty of 4 is the best choice, both for h_{WLS} and h_{DWLS}.

Tuning Stability Threshold

The next parameter is the stability threshold τ. This parameter is only used in h_{DWLS}. The definition of this heuristic in subsubsection 4.4 uses two parameters: true stability threshold τ_t and false stability threshold τ_f. During the initial conception of h_{DWLS}, both of these parameters were merged as a single stability threshold parameter where $\tau_t = \tau_f = \tau$. Indeed, the split into the two parameters τ_t and τ_f was motivated by the results of this benchmark.

The timings are shown in Table 5.4. The best performing stability threshold is 0.6. However, it is closely followed by 0.99. These two values for the stability threshold are the two peaks of these results. The question that arises is if 0.6 is simply an outlier or if there is a more fundamental reason for these two peaks.

One hypothesis we have formulated is that the different thresholds optimize the behavior for different literals. Since true literals make clauses useless to further solver progress, as they can never trigger under the current assignment, such clauses are penalized by the true literal penalty. Likewise, literals that are mostly false reduce the effective size of the

	WLS		DWLS	
β	PAR-2 score	# solved (SAT + UNSAT)	PAR-2 score	# solved (SAT + UNSAT)
750	85607.04	230 (132 + 98)	86519.16	230 (131 + 99)
1500	84461.12	232 (129 + 103)	83424.89	235 (131 + 104)
3000	80802.06	239 (134 + 105)	83413.01	234 (128 + 106)
6000	79881.18	240 (132 + 108)	80212.89	**241 (131 + 110)**
12000	**79293.60**	**241 (134 + 107)**	81014.10	237 (131 + 106)
24000	80438.19	237 (132 + 105)	**79087.70**	**241 (133 + 108)**

Table 5.6: The parameter tuning results of the buffer size. Other parameters are: $\epsilon = 10^6$, $\omega = 4$, $\tau = 0.6$

clause and are therefore are rewarded with a cost of zero. Based on the results of Table 5.4 we hypothesized that the two peaks indicate the optimal threshold for each of these two thresholds respectively.

To test this hypothesis we ran a new experiment which splits the stability threshold and tests every combination of the two peak values 0.6 and 0.95. The results for experiments are presented in Table 5.1. The best results were achieved with a true threshold of 0.95 and a false threshold of 0.6. While this fact in isolation might confirm our hypothesis, the second-best results swap the two values. We can not justify this fact and therefore disregard our hypothesis. Additionally, it is troubling to see that in the cases where the true threshold and false threshold are equal, both being 0.95 performs slightly better than 0.6. This is in contradiction to the initial stability threshold tuning results and casts doubt on the overall validity of this and the previous benchmarks.

Tuning Buffer Size

The final parameter we analyzed is the buffer size β. This parameter represents the number of literals that are aggregated by Mallob and shared between all CaDiCal instances. Each CaDiCal instance selects the best clauses using its heuristic until a third of all literals are imported. The results are shown in Table 5.6. There is no clear value that appears to perform best. All best results appear in the buffer sizes that are equal or larger than 6000. Indeed, this result confirms preexisting knowledge about the optimal buffer size. Mallob inherits its default buffer size of 1500 from HordeSAT. When multiplying the import ratio with the buffer size, we arrive at approximately the same conclusion, that there is little to gain by surpassing this default.

5.3 Evaluation of Heuristics

We evaluate every proposed heuristic in Section 4.4 by running it on the same machine used for parameter tuning in the previous section. Again we use all 128 physical cores to

heuristic	PAR-2 score	# solved (SAT + UNSAT)
h_{size^*}	**49864.17**	**292 (146 + 146)**
h_{WLS}	51748.62	288 (145 + 143)
h_{DWLS}	52447.99	286 (148 + 138)
h_{product}	52639.58	286 (146 + 140)
h_{mean}	54406.01	285 (148 + 137)
h_{2min}	56315.82	279 (143 + 136)
h_{min}	58318.98	276 (144 + 132)
h_{lukas}	62730.68	270 (145 + 125)

Table 5.7: The benchmark results of every proposed heuristics, ordered by PAR-2 score. Parameters are $\epsilon = 10^6$, $\omega = 4$, $\tau = 0.6$, $\beta = 6000$

run Mallob.

For this benchmark all runs perform trail sampling. This includes the size heuristics which does not require literal stability to be computed. However, including trail sampling allows us to focus on the effects of the import heuristics since the trail sampling overhead should cancel out when comparing their relative performance.

The results for this benchmark are presented in Table 5.7. The default h_{size^*} heuristic is best performing, followed by h_{WLS}, h_{DWLS}, h_{product} and h_{mean}. h_{lukas} is the worst performing heuristic.

5.4 Cloud Evaluation

As Mallob scales its shared clause buffer size with the number of used solvers, more solvers imply more potentially useful clauses to select from. Indeed, it is possible that more cores will allow our heuristics to select more effectively.

To explore if increased parallelism magnifies strengths of literal stability based heuristics, we will run a final benchmark on the powerful SuperMUC-NG computer using six times the number of cores. The SuperMUC-NG is a supercomputer which features a total 6336 compute nodes, each a 24-core Xeon Platinum 8174 processor with 96 GB of RAM and an OmniPath network interconnection.

As the usage of a supercomputer is costly in terms of energy and money we only benchmark the best performing heuristics from Section 5.3. The resulting PAR-2 score is presented in Table 5.8. Figure 5.2 shows the corresponding performance plot.

The resulting order is mostly the same as in Table 5.7. As h_{size} is clearly the best performing heuristic we need to conclude that our stability heuristics do not result in a performance benefit. Furthermore, h_{size^*} which unnecessarily performs trail sampling, is still slightly better than the second-best best heuristic h_{wls}. Therefore, it is not just the overhead introduced by trail sampling that prevents h_{wls} from surpassing h_{size}. Neither

heuristic	PAR-2 score	# solved (SAT + UNSAT)
h_{size}	**29741.76**	**319 (158 + 161)**
h_{size^*}	32903.85	316 (156 + 160)
h_{wls}	33227.80	315 (156 + 159)
h_{product}	35191.09	311 (155 + 156)
h_{dwls}	36183.01	309 (154 + 155)
h_{mean}	38003.49	308 (156 + 152)

Table 5.8: The SuperMUC-NG benchmark results of the best heuristics, ordered by PAR-2 score. Parameters are $\epsilon = 10^6$, $\omega = 4$, $\tau = 0.6$, $\beta = 6000$

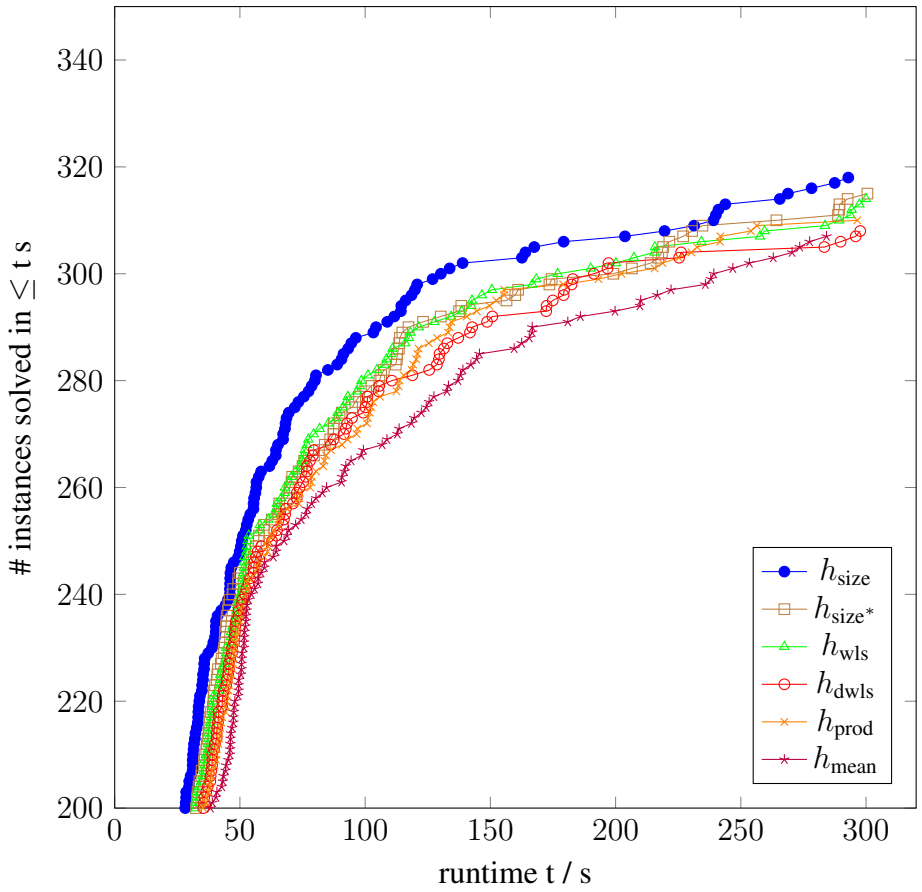

Figure 5.2: Performance of the best heuristics on 764 cores.

were we able to find classes of problems where stability based heuristics showed a clear advantage over the default size heuristic.

We observe that heuristics that correlate with the size of the clause appear to perform most strongly. Whereas the heuristics which discard information like h_{mean}, h_{min} and h_{lukas} perform the worst overall. This supports the efficacy of the h_{size} heuristic further.

In conclusion, we recognize that our presented clause sharing heuristics do not benefit a parallel SAT solver.

6 Conclusion

In this thesis we explored the use of literal stability to improve clause sharing. First, we provided an overview starting from SAT solving first principles up to modern parallel SAT solvers. We introduced the notion of literal stability which can be calculated by sampling the trail in regular intervals. Using a custom derived unbiased exponential moving average with support for efficient repeated updates we were able to optimize trail sampling significantly. We also discussed additional techniques to improve its performance further. Continuing with literal stability, we constructed multiple heuristics for clause sharing which reprioritize foreign clauses to import. Finally, we optimized the needed parameters and empirically evaluated all heuristics. Unfortunately, we were unable to show any benefit over the default size heuristic.

Future Work

Although we were unable to improve the performance of a parallel SAT solver by integrating literal stability into the clause import procedures, there are potentially other areas that could benefit from literal stability information. Stability has already shown to improve the performance of cryptographic instances when it is applied to select blocking literals [20]. Likewise, there might be other areas that where literal stability could aid. For example, when selecting a decision literal, literals with a certain stability profile might perform better.

Bibliography

[1] Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In *Theory and Applications of Satisfiability Testing–SAT 2012: 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings 15*, pages 200–213. Springer, 2012.

[2] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. On freezing and reactivating learnt clauses. In *Theory and Applications of Satisfiability Testing-SAT 2011: 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings 14*, pages 188–200. Springer, 2011.

[3] Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel sat solvers. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 197–205, Cham, 2014. Springer International Publishing.

[4] Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio sat solver. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 156–172, Cham, 2015. Springer International Publishing.

[5] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[6] Armin Biere and Andreas Fröhlich. Evaluating cdcl restart schemes. *Proceedings of Pragmatics of SAT*, pages 1–17, 2015.

[7] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[8] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[9] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Cud@ sat: Sat solving on gpus. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3):293–316, 2015.

[10] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201—215, july 1960.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[12] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Sat competition 2020. *Artificial Intelligence*, 301:103572, 2021.

[13] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[14] Youssef Hamadi, Said Jabbour, and Jabbour Sais. Control-based clause sharing in parallel sat solving. *Autonomous Search*, pages 245–267, 2012.

[15] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *JSAT*, 6:245–262, june 2009.

[16] Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo, and Nils Froleyks. Sat competition 2021. `https://satcompetition.github.io/2021/results.html`. Accessed: 2023-03-17.

[17] Marijn Heule, Matti Järvisalo, Martin Suda, and Tomáš Balyo Markus Iser. Sat competition 2022. `https://satcompetition.github.io/2022/results.html`. Accessed: 2023-03-17.

[18] Marijn Heule, Oliver Kullmann, and Victor W Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 228–245. Springer, 2016.

[19] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving sat in grids. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 430–435, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[20] Markus Iser and Tomáš Balyo. Unit propagation with stable watches. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021). Ed.: L. D. Michel*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik GmbH (LZI), 2021.

[21] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European conference on Artificial intelligence*, pages 359–363, 1992.

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

[23] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[24] Inês Lynce and Joao Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9*, pages 136–141. Springer, 2006.

[25] Panagiotis Manolios and Yimin Zhang. Implementing survey propagation on graphics processing units. In *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9*, pages 311–324. Springer, 2006.

[26] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a sat problem. *Journal of Automated Reasoning*, 24(1-2):165, 2000.

[27] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530—535. Association for Computing Machinery, 2001.

[28] Chanseok Oh. Between sat and unsat: the fundamental difference in cdcl sat. In *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*, pages 307–323. Springer, 2015.

[29] Muhammad Osama, Anton Wijs, and Armin Biere. Sat solving with gpu accelerated inprocessing. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part I 27*, pages 133–151. Springer, 2021.

[30] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings 10*, pages 294–299. Springer, 2007.

[31] S Plaza, I Markov, and Valeria Bertacco. Low-latency sat solving on multicore processors with priority scheduling and xor partitioning. In *International workshop on logic and synthesis*, 2008.

[32] Nicolas Prevot, Mate Soos, and Kuldeep S. Meel. Leveraging gpus for effective clause sharing in parallel sat solving. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 471–487, Cham, 2021. Springer International Publishing.

[33] Dominik Schreiber and Peter Sanders. Scalable sat solving in the cloud. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 518–534. Springer, 2021.

[34] Laurent Simon and Gilles Audemard. Predicting learnt clauses quality in modern

sat solver. In *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, 2009.

[35] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: A distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4–6):543—560, june 1996.