Multiple Decision Making in Conflict-Driven Clause Learning

Muhammad Osama and Anton Wijs
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

***Abstract*—Most modern and successful SAT solvers are based on the Conflict-Driven Clause-Learning (CDCL) algorithm. The CDCL approach is to try to learn from previous assignments, and based on this, prune the search space to make better decisions in the future. In the current paper, we propose the introduction of a multiple decision maker (MDM) into CDCL. Adhering to a number of rules, MDM constructs sets of decisions to be made at once. Experiments show MDM has a considerably positive impact on CDCL, for many different SAT application problems. Overall, about 50% of the benchmarks we considered were solved faster when MDM was enabled, and the total processing time of all benchmarks was reduced by 6%. Moreover, MDM allowed 31 extra problems to be solved. We introduce MDM, analyse its impact, and try to understand the cause of that impact.**

***Index Terms*—Satisfiability, CDCL, Multiple Decision Making**

I. Introduction

During the past decade, SAT solving has been used extensively for various applications, such as combinational equivalence checking [1], automatic test pattern generation [2], [3], automatic theorem proving [4], and symbolic model checking [5], [6]. Most modern and successful SAT solvers are based on the Conflict-Driven Clause-Learning (CDCL) algorithm [7]–[12]. The CDCL approach is to try to learn from previous assignments, and based on this, prune the search space to make better decisions in the future.

Many solvers have been introduced that employ CDCL, such as GRASP [7], CHAFF [8], BERKMIN [11], MINISAT [10], GLUCOSE [13], and LINGELING [12]. GRASP was the first tool applying CDCL, after which CHAFF introduced the so-called *two watched literals* optimisation and the VSIDS decision heuristic (more on these in Section IV). BERKMIN and MINISAT introduced further implementation and heuristics optimisations. The authors behind GLUCOSE presented robust clause deletion and restart heuristics. The LINGELING solver introduced the effective use of SAT simplifications [14]–[16] as an in-processing technique during the solving process [17].

One aspect of CDCL that has always remained the same is that to explore all possible assignments, a single decision is made at a time. In the current paper, we propose to extend CDCL with the ability to make and propagate multiple decisions at once. Our motivation for this is to further improve the runtime performance of CDCL. To ensure effective selection of non-singleton sets of decisions, we require that the decisions in such a set do not lead to implications. In this paper, we present our generalisation of CDCL, with support for what we call *Multiple Decision Making* (MDM). To experimentally test the extension, we implemented MDM in a new SAT solver developed by us, and compared runtimes with both MINISAT and our solver without MDM. In addition, we implemented MDM in GLUCOSE, and compared it to GLUCOSE without MDM, both with and without preprocessing. MDM turns out to be particularly effective for SAT problems stemming from applications, i.e., that are not random. Overall, about 50% of the benchmarks we considered were solved faster with MDM.

The paper is organised as follows: Sect. II introduces the preliminaries. In Sect. III, we present our generalisation of CDCL. In Sect. IV, it is explained how MDM can be implemented, focusing on heuristics and optimisations. In Sect. V, we integrate MDM into CDCL and address its correctness. Finally, benchmark results are given and discussed in Sect. VI, and conclusions are drawn in Sect. VII.

II. Preliminaries

All SAT formulas in this paper are in conjunctive normal form (CNF). A CNF φ is a conjunction of clauses $\bigwedge_{i=1}^{m} C_i$ where each clause C_i is a disjunction of literals $\bigvee_{j=1}^{k} \ell_j$ and a literal is a Boolean variable x or its negation $\neg x$. For a literal ℓ, $v(\ell)$ denotes the referenced variable, i.e., $v(x) = x$ and $v(\neg x) = x$. The domain of all literals is $\mathbb{L}$. We interpret a clause C as a set of literals $\{\ell_1, \ldots, \ell_k\}$ representing the clause $\ell_1 \vee \ldots \vee \ell_k$, and a SAT formula φ as a set of clauses $\{C_1, \ldots, C_m\}$ representing the formula $C_1 \wedge \ldots \wedge C_m$. We denote the set of all clauses of φ in which ℓ occurs by $\mathcal{C}_\ell = \{C \in \varphi \mid \ell \in C\}$. The domain of the Booleans is $\mathbb{B}$, **true** is represented by $\top$ and **false** by $\bot$, and we have $\neg\top = \bot$ and $\neg\bot = \top$. An *assignment* ℓ refers to assigning $\top$ to literal ℓ. During SAT solving, we keep track of a set σ consisting of all literals that have been assigned $\top$. When *applying* an assignment ℓ, σ is updated to $\sigma \cup \{\ell\}$.

With $\ell \models_\sigma \top$, we express that literal ℓ evaluates to **true**, i.e., it is satisfied, w.r.t. σ. This is defined as $\ell \models_\sigma \top \triangleq \ell \in \sigma$. With $\ell \models_\sigma \bot$, we refer to $\neg\ell \in \sigma$. The evaluation of ℓ is

This work is part of the GEARS project with project number TOP2.16.044, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

2375-0197/20/$31.00 ©2020 IEEE
DOI 10.1109/ICTAI50040.2020.00035

undetermined, denoted by $\ell \models_\sigma \uparrow$, in case neither $\ell \in \sigma$ nor $\neg\ell \in \sigma$. For a clause $C \in \varphi$, $C \models_\sigma \top$ expresses that C is satisfiable w.r.t. σ. We have $C \models_\sigma \top$ iff $\exists \ell \in C.\ell \models_\sigma \top$. If for all $\ell \in C$, we have $\ell \models_\sigma \bot$, then $C \models_\sigma \bot$. If neither $C \models_\sigma \top$ nor $C \models_\sigma \bot$, we have $C \models_\sigma \uparrow$. Formula φ is satisfiable w.r.t. σ, i.e., $\varphi \models_\sigma \top$, iff $\forall C \in \varphi.C \models_\sigma \top$. In that case, σ is a *model* for φ.

With $\mathit{Free}_\sigma(C)$, we refer to the set of free, unassigned literals in C, i.e., $\mathit{Free}_\sigma(C) = \{\ell \in C \mid \ell \notin \sigma \wedge \neg\ell \notin \sigma\}$. A clause C is called *unit* iff $C \models_\sigma \uparrow \wedge |\mathit{Free}_\sigma(C)| = 1$, i.e., its evaluation is undetermined and one literal is unassigned.

III. Generalised CDCL-based SAT Solving

A. The CDCL procedure

Given a CNF formula φ, a CDCL-based SAT solver tries to find a model for φ. The search is performed in three main steps: *decision making* or branching, *propagating* the effects of assignments, and *analysing* in case so-called conflicts arise. The CDCL procedure is described by Alg. 1. This description generalises the one given in [7] to support MDM. The extension we propose in this paper affects the DECIDE routine coloured blue at line 9. In the extension, DECIDE may not just make a single decision when called, as in standard CDCL, but it may make a set of decisions. How and when DECIDE makes multiple decisions should be ignored for now. This is covered in Sections IV, V.

The global variables of the CDCL procedure are φ, σ, a set σ' of assignments that were previously part of σ, but have been removed due to backtracking (the purpose of this is to apply progress saving, as proposed in [18]), and a *decision level function* δ, which records at which *decision/search level* (d) each literal has been added to σ. In addition, the function *source* is used to keep track of which implications are caused by which clauses. More on this later.

At lines 2-5 of Alg. 1, the CDCL procedure is given, which first calls the procedure BCP at the initial search level ($d = 0$). More on this procedure later. After that, SEARCH is called for $d = 0$, with the *forward jump level* set to 0 ($\vec{d} = 0$). The forward jump level defines the level that SEARCH needs to progress towards before making the next decisions.

Procedure SEARCH performs one step in searching for a σ that models φ. If the two given levels d and $\vec{d}$ are equal and we are not at the initial level, DECIDE is called (line 9), which implements the decision making step; it decides which decisions to make next, i.e., to which literals $\top$ should be assigned. The forward jump level $\vec{d}$ is subsequently increased by the number of decisions made (for instance, $\vec{d}$ is incremented if one decision is made). At the initial level 0, we simply increment $\vec{d}$ (line 12), to ensure that the first decision will be made by SEARCH$(1, 1)$, called at line 15.

If no decisions could be made, then φ is satisfied by the current σ, and SEARCH returns d together with the conclusion SAT (line 10). Otherwise, at line 11, the new decisions in $\mathcal{L}$ are added to σ, and the **while** loop at lines 13-21 is entered.

The propagation of a decision and all its resulting *implications* is done by the *Boolean Constraint Propagation* (BCP) procedure (called at line 14). The description of BCP is given at lines 22-28. As long as there are unit clauses (line 24), a unit clause C is picked (line 25), and its unassigned literal is added to σ (line 26). This assignment is called an *implication*. If the update does not make φ unsatisfied (line 27), the procedure is repeated. After every update of σ, the function *source* is updated to record the fact that C caused $\mathit{Free}_\sigma(C)$ to be added to σ. This is relevant when clause learning needs to be applied, which is discussed later. The BCP procedure identifies all implications, unless a *conflict* is detected.

Algorithm 1 CDCL, generalised to support MDM

```
1:  global vars: formula φ, assigns. σ, saved-assigns. σ', level function δ, implication function source
2:  procedure CDCL( )
3:      σ ← ∅, σ' ← ∅, δ ← ∅
4:      BCP(0)
5:      if SEARCH(0, 0) = (0, UNSAT) then return UNSAT else return SAT
6:  procedure SEARCH(d, d⃗)
7:      if d = d⃗ then
8:          if d > 0 then
9:              L ← DECIDE(d), d⃗ ← d⃗ + |L|
10:             if L = ∅ then return (d, SAT)
11:             σ ← σ ∪ L
12:         else d⃗ ← d⃗ + 1
13:     while true do
14:         if d < d⃗ − 1 ∨ BCP(d) then
15:             (d̄, sat) ← SEARCH(d + 1, d⃗)
16:             if sat = SAT then return (d, SAT)
17:             else if d > d̄ then BACKJUMP(d), return (d̄, UNSAT)
18:             else d⃗ ← d + 1
19:         else
20:             (Ĉ, d̄) ← ANALYSE(d), φ ← φ ∪ Ĉ, BACKJUMP(d)
21:             return (d̄, UNSAT)
22: procedure BCP(d)
23:     source ← ∅
24:     while {C ∈ φ | |Free_σ(C)| = 1} ≠ ∅ do
25:         pick a clause C ∈ φ for which |Free_σ(C)| = 1
26:         σ ← σ ∪ Free_σ(C); source ← source ∪ {(Free_σ(C), C)}
27:         if φ ⊨_σ ⊥ then return false
28:     return true
29: procedure BACKJUMP(d)
30:     σ ← σ \ {ℓ | (ℓ, d) ∈ δ}
31:     σ' ← σ' ∪ {ℓ | (ℓ, d) ∈ δ} \ {¬ℓ | (ℓ, d) ∈ δ}
32:     δ ← δ \ {(ℓ, d) ∈ δ}
33: procedure ANALYSE(d)
34:     Ĉ ← LEARNCCLAUSE (d)
35:     d̄ ← MAX({0} ∪ ({δ(ℓ̂) | ℓ̂ ∈ Ĉ} \ {d}))
36:     return (Ĉ, d̄)
37: procedure LEARNCCLAUSE(d)
38:     Ĉ ← C with C ∈ φ and C ⊨_σ ⊥
39:     while |{ℓ ∈ Ĉ | δ(ℓ) = d}| > 1 do
40:         pick a literal ℓ ∈ Ĉ for which ∃C.(¬ℓ, C) ∈ source
41:         Ĉ ← Ĉ ⊗_ℓ source(¬ℓ)
42:     return Ĉ
```

Definition 1 (Conflict). *Given a formula φ and a set of assignments σ, there is a* conflict *iff there exists a $C \in \varphi$ with $C \models_\sigma \bot$.*

Recall that at the start of CDCL, BCP is called (line 4). The purpose of this call is to propagate implications initially present in φ. Furthermore, note that the BCP procedure does not need to be called at line 14 when $d < \vec{d}-1$. When DECIDE makes a single decision, $d < \vec{d} - 1$ does not hold, hence BCP is called. When n decisions are made ($n > 1$), SEARCH progresses $n - 1$ levels before calling BCP again.

If BCP returns **true** at line 14 or we are forward jumping ($d < \vec{d} - 1$), SEARCH is called recursively, moving on to the next decision level. If this invocation of SEARCH returns that the formula is satisfied, the result is propagated back (line 16). Else, if a *backtrack level* $\overleftarrow{d}$ is returned that is smaller than d, backtracking is needed, and the backtrack level is returned together with the conclusion UNSAT (line 17). A backtrack step is done in the BACKJUMP procedure, given at lines 29-32. All assignments made at the current level d are removed from σ and δ, and added to σ'. Any contradictory information is removed from σ'. In this way, σ' can be used to recall the last value assigned to a variable, if one exists. This is relevant in DECIDE; in Section IV, we specify how σ' is used in decision making. When $\overleftarrow{d}$ is returned together with UNSAT at line 17, $\overleftarrow{d}$ is propagated back to the calling SEARCH procedure (line 15), which will then in turn backtrack if $\overleftarrow{d}$ is smaller than its level d (line 17), and so on, until the backtrack level has been reached, in which case the forward jump level is set to the next level to move forward again (line 18).

In case BCP returns **false** at line 14, there is a conflict, and this must be analysed (line 20). The ANALYSE procedure, described at lines 33-36, tries to identify all assignments causing the conflict and backtracks the required number of levels to undo those assignments. The negations of those assignments are recorded by the LEARNCCLAUSE procedure (line 34), in a new clause, called the *conflict clause* ($\hat{C}$), to avoid making that combination of assignments in the future. How such a clause is constructed is not essential here, but for the sake of completeness, we provide a possible procedure.

The LEARNCCLAUSE procedure is given at lines 37-42. The conflict clause can be constructed by a sequence of resolution steps [7], starting with the clause that caused the conflict (line 38). Each step produces an intermediate new clause (*resolvent*). A resolvent is the result of applying the *resolution rule* [19] w.r.t. some implication ℓ at level d. For this, we use the *resolving operator* $\otimes_\ell$ on clauses C_1 and C_2 with $\ell \in C_1$, $\neg\ell \in C_2$. It is defined as $C_1 \otimes_\ell C_2 = C_1 \cup C_2 \setminus \{\ell, \neg\ell\}$.

As long as more than one literal in $\hat{C}$ was assigned a value at d (line 39), i.e., as long as some values are the result of implications at d, $\hat{C}$ is rewritten. This is done by selecting a literal ℓ that obtained a value due to an implication at d ($source(\neg\ell)$ is defined) at line 40, and combining the clause that caused that implication ($source(\neg\ell)$) with $\hat{C}$ using the resolving operator, resulting in a new clause in which ℓ no longer appears (line 41). Given $\hat{C}$, the backtrack level $\overleftarrow{d}$ is defined as MAX($\{0\} \cup (\{\delta(\ell) \mid \ell \in \hat{C}\} \setminus \{d\})$) (line 35): the highest level involved in $\hat{C}$ that is smaller than d is selected. In case no such level exists ($\hat{C}$ contains a single literal at d), 0 is selected. Both $\hat{C}$ and $\overleftarrow{d}$ are returned at line 36. Next, $\hat{C}$ is added to the input formula, the backtracking to $\overleftarrow{d}$ is initiated, and $\overleftarrow{d}$ is returned with UNSAT (lines 20-21).

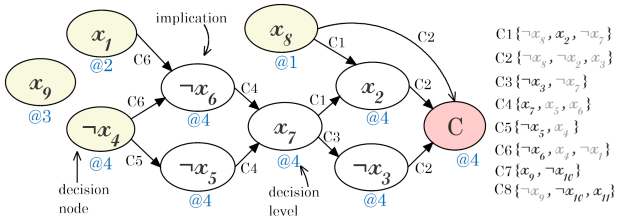


Fig. 1: A visualization of CDCL solving on a small example

Example 1. *A small example of applying* standard *(single-decision) CDCL is illustrated in Fig. 1 by an implication graph [7]. Consider a formula φ containing, among others, the following clauses:*

$$\varphi = \{\{\neg x_8, x_2, \neg x_7\}, \{\neg x_8, \neg x_2, x_3\}, \{\neg x_3, \neg x_7\}, \{x_7, x_5, x_6\}, \{\neg x_5, x_4\}, \{\neg x_6, x_4, \neg x_1\}, \{x_9, \neg x_{10}\}, \{\neg x_9, \neg x_{10}, x_{11}\}, \ldots\}$$

In addition, consider x_8, x_1, x_9 and $\neg x_4$ as the decisions made so far. The decisions and their levels are indicated in Fig. 1 by the green ovals and blue numbers, respectively. Once these decisions are made, BCP *identifies implications. Each implication (white ovals in Fig. 1) takes the highest level among its parents. The procedure first sets $\neg x_5$, $\neg x_6$ and x_7 to $\top$, before there are a number of possible scenarios that all lead to a conflict. The order in which unit clauses are analysed by* BCP *determines which scenario occurs. In the figure, both x_2 and $\neg x_3$ are set to $\top$, before C_2 causes a conflict.*

At this point, LEARNCCLAUSE *can produce the clause $\hat{C} = \{\neg x_7, \neg x_8\}$, as the result of $(C_2 \otimes_{x_2} C_1) \otimes_{x_3} C_3$. Finally, the backtrack level $\overleftarrow{d}$ is determined by the highest decision level other than the conflict level in $\hat{C}$; in this example, $\overleftarrow{d} = 1$.*

B. Multiple Decision Making

Next, we reason about making multiple decisions simultaneously with DECIDE. Making a decision can always lead to conflicts, and making more than one decision at once increases the likelihood of a conflict occurring. To avoid repeatedly selecting sets of decisions that cause conflicts, we wish to construct multiple decisions sets, i.e., non-singleton sets of decisions, in such a way that it is guaranteed that no conflicts will occur. For this, we define multiple decisions set as follows.

Definition 2 (Multiple decisions set). *Given a formula φ and a set of assignments σ, we call a set $\mathcal{M} \subseteq \mathbb{L} \setminus \{\ell, \neg\ell \mid \ell \in \sigma\}$ with $|\mathcal{M}| > 1$ a set of* multiple decisions *iff $\{C \in \varphi \mid |Free_{\sigma \cup \mathcal{M}}(C)| = 1\} = \emptyset$.*

In words, a set of multiple decisions $\mathcal{M}$ can be selected, i.e., is *valid*, iff $\mathcal{M}$ does not result in unit clauses. Note that $\mathcal{M}$ only contains new decisions, and not previously selected literals or ones that contradict σ. Def. 2 cannot be efficiently used to construct a set of multiple decisions, since it refers

to $\mathcal{M}$ as a whole. The basic approach to construct $\mathcal{M}$ is to iteratively select a decision ℓ and add it to $\mathcal{M}$ iff it does not lead to a violation of the condition in Def. 2.

Example 2. *By Def. 2, the set of literals $\{x_3, x_7, \neg x_9\}$ for the formula φ in Ex. 1 is initially, with $\sigma = \emptyset$, not a valid multiple decisions set, as it results in C_3 being unsatisfied and C_7 being a unit clause. On the other hand, $\{x_1, x_8, x_9\}$ is valid.*

Since a valid multiple decisions set does not produce any unit clauses, note that DECIDE cannot always select multiple decisions. The solving of a formula can only terminate if at some point unit clauses are produced and implications are processed. Therefore, DECIDE should not always, but periodically select multiple decisions. In the next section, we discuss a possible mechanism to achieve this.

IV. Implementing MDM

So far, we have presented MDM mathematically. In this section, we discuss how to make it efficient in practice, and we address the correctness of CDCL with MDM.

A. Decision heuristics

The DECIDE routine in Alg. 1 determines which literals should be selected and assigned **true** for the next decisions. Actually, in existing solvers, where DECIDE makes a single decision at a time, a *variable* is selected and specific heuristics are used to set it either to **true** or **false**, i.e., a literal is selected in two steps. Regarding variable selection, a robust heuristic is Variable State Independent Decaying Sum (VSIDS) [8]. In VSIDS, each variable has a counter (sum), called α, which initially has the value 0. Once a conflict clause is deduced, the α-values of all variables it refers to are incremented, and the α-values of all variables in the formula are divided by some constant (implementing so-called decay). Each time a decision must be made, the variable x with the highest α is selected. VSIDS does not address whether to assign x **true** or **false**. Other heuristics are used for that. The BERKMIN [11] solver improves VSIDS by incrementing the α-values of those variables referenced by *any* clause involved in the conflict analysis. This has been adopted in many SAT solvers, and we adopt it as well.

Inspired by the above heuristics, we also apply VSIDS for the selection of multiple decisions. Initially, after φ has been parsed, a sorted list L of the *unassigned* variables in φ is created. For each two variables $x, y \in L$, we define the following order for the sorting.

$$x \prec y \triangleq \alpha(x) \leq \alpha(y) \implies h(x) \times h(\neg x) > h(y) \times h(\neg y)$$

In the formula, h is a histogram function ($h(\ell) = |\mathcal{C}_\ell|$). From L, a set of variables can be obtained for decision making, starting from the highest ranking variable, and stopping once a bound B for the ranking has been reached. The sorted list is updated each time new variables have to be selected.

B. 2-WL optimisation

When constructing a multiple decisions set, checking whether the condition of Def. 2 still holds every time a literal ℓ is selected can be optimised by using the so-called *two-watched literals* (2-WL) optimisation [8]. When using it, in each clause $C \in \varphi$, two unassigned literals $\ell_1, \ell_2 \in C$ are marked as *watched*, and as soon as one is set to $\bot$, another unassigned literal is selected for watching, unless there are no unassigned, unwatched literals left. This optimisation is particularly suitable to check for violations of the condition of Def. 2, as it allows us to only consider the clauses in $\mathcal{C}_{\neg\ell}$ in which $\neg\ell$ is watched; if it is not in some clause $C \in \mathcal{C}_{\neg\ell}$, then there are at least two other, unassigned literals in C, hence C cannot become unit when $\neg\ell$ is assigned a value. If there are no more than two watched literals in C, then $\neg\ell$ has to be watched, since it was unassigned before being selected. We formalise the predicate that a literal ℓ is watched in a clause C with $\mathcal{W}_C(\ell)$.

C. Decision freezing

During the construction of a multiple decisions set, we wish to avoid the repeated selection of literals that cause the condition of Def. 2 to be violated. For this reason, we introduce the notion of *freezing*. With it, we over-approximate the potential to produce unit clauses. If we add a valid literal ℓ to $\mathcal{M}$, the clauses in $\mathcal{C}_{\neg\ell}$ have $\bot$ assigned to $\neg\ell$, and thereby have more potential to become unit. Subsequently selecting a literal ℓ' that also appears in any of those clauses in $\mathcal{C}_{\neg\ell}$ can possibly produce a unit clause. To avoid this, we freeze all unassigned literals in $\mathcal{C}_{\neg\ell}$ after the selection of ℓ, thereby not allowing them to be selected for the multiple decisions set.

Checking whether a literal is frozen or not is simpler than repeatedly checking for a violation of the condition of Def. 2. A literal ℓ' is frozen if either ℓ' or $\neg\ell'$ *depends* on a previously selected decision:

Definition 3 (Decision dependency relation). *We call a relation* $\mathsf{D}\colon \mathbb{L} \times \mathbb{L}$ *a* decision dependency relation *iff for all $\ell, \ell' \in \mathbb{L}$, we have ℓ' D ℓ iff there exists a $C \in \mathcal{C}_{\neg\ell}$ such that $\ell' \in C \vee \neg\ell' \in C$.*

Given a multiple decisions set $\mathcal{M}$, the set of frozen decisions is defined as follows.

Definition 4 (Frozen decisions). *Given a formula φ, a set of assignments σ, and a multiple decisions set $\mathcal{M}$, the set of* frozen decisions *$\mathcal{F}$ is defined as $\mathcal{F} = \{\ell' \mid \ell' \in \mathbb{L} \setminus \{\ell'', \neg\ell'' \mid \ell'' \in \sigma\} \wedge \exists \ell \in \mathcal{M}.\ell'$ D $\ell\}$.*

In practice, we actually freeze variables, not literals, due to the two steps variable selection procedure described at the beginning of this section. For consistency, though, we reason about literals in the remainder of this section.

D. Integrating heuristics and freezing in MDM

Next, we explain how the MDM procedure can be implemented. As input, Alg. 2 requires the current decision level d. At line 3, this is stored in d', which is used to assign

Algorithm 2 Multiple Decision Maker

```
1: global vars: formula φ, saved-assigns. σ′, dec. level function δ
2: procedure MDM(d)
3:     d′ ← d, M ← ∅, F ← ∅
4:     L ← SORT(ASSIGNSCORES(φ), ≺)
5:     for all x ∈ L do
6:         if x ∉ F then
7:             if x ∈ σ′ ∨ ¬x ∈ σ′ then ℓ ← LIT(x, x ∈ σ′)
8:             else ℓ ← LIT(x, h(¬x) ≥ h(x))
9:             if ∀C ∈ C_¬ℓ.W_C(¬ℓ) ⟹ |Free_σ(C)| > 1 then
10:                if DEPFREEZE (ℓ) then
11:                    M ← M ∪ {ℓ}
12:                    δ(x) ← d′, δ(¬x) ← d′
13:                    d′ ← d′ + 1
14:    return M
15: procedure DEPFREEZE(ℓ)
16:    for all C ∈ C_¬ℓ do
17:        if W_C(¬ℓ) then
18:            for all ℓ′ ∈ C do
19:                if ℓ′ ∈ M then return false
20:                if v(ℓ′) ≠ v(ℓ) ∧ v(ℓ′) ∈ L then F ← F ∪ v(ℓ′)
21:    return true
```

consecutive levels to the decisions selected by MDM. Sets $\mathcal{M}$ and $\mathcal{F}$ are initially empty. At line 4, we create the list of variables L using ASSIGNSCORES, and sort it using the $\prec$-order. Next, we iterate over L (line 5).

For each unfrozen variable (line 6), a value is picked to select a literal. The LIT function at lines 7 and 8 takes a variable x and a predicate p, and produces x if p and $\neg x$ otherwise. If for x, a literal has been recorded in σ' (progress saving [18], see Alg. 1, line 31), then this literal is selected (line 7), otherwise, the histogram function h determines the value (line 8). At line 9, it is checked whether the selected literal ℓ is valid, according to Def. 2, restricting the check to clauses in $\mathcal{C}_{\neg\ell}$ in which $\neg\ell$ is watched. If ℓ is valid, it is attempted to freeze dependent variables (line 10). If successful, ℓ is added to $\mathcal{M}$, the assignments are recorded with δ at level d', and d' is incremented (lines 11-13).

The procedure DEPFREEZE is given at lines 15-21. It tries to freeze all variables dependent on ℓ, according to Defs. 3 and 4. We apply the 2-WL optimisation to speed up iterating over clauses in $\mathcal{C}_{\neg\ell}$. This has the drawback that some literals that have to be frozen may be ignored, if they do not appear in a clause in which $\neg\ell$ is watched. To remedy this, we check whether a literal has already been added to $\mathcal{M}$ before freezing it (line 19). If it was added, then clearly, ℓ should not have been added, and the latter's selection is canceled.

Example 3. *Consider again Ex. 1. We assume that x_8 is picked first and is being watched. Since $\neg x_8$ appears more frequently than x_8, $\top$ is assigned to x_8 (line 8, Alg. 2), hence the* DEPFREEZE *procedure freezes all unassigned variables in $\mathcal{C}_{\neg x_8} = \{C_1, C_2\}$, that is x_2, x_3, and x_7. Similarly, x_1 and x_9 can be chosen for a decision, causing x_4, x_6, x_{10}, and x_{11} from clauses C_6, C_7, and C_8 to be frozen, respectively. Finally, the set $\mathcal{M} = \{x_1, x_8, x_9\}$ is constructed. As all remaining variables are frozen, this set is not extended any further.*

Algorithm 3 CDCL with integrated MDM

```
1: global vars: assigns. σ, MDM rounds, conflicts limit maxConflicts
2: procedure CDCL( )
3:     R ← rounds, status ← UNSOLVED, restarts ← 0
4:     while status = UNSOLVED do
5:         maxConflicts ← RESTART(restarts)
6:         (0, status) ← SEARCH(0, 0), restarts ← restarts + 1
7:         if restarts mod maxConflicts = 0 then R ← rounds
8:     return status
9: procedure DECIDE(d)
10:    L ← ∅
11:    if R ≠ 0 then L ← MDM(d), R ← R − 1
12:        if σ = ∅ then PUMPFROZEN(M)
13:    if L = ∅ then L ← FOLLOWUPDEC(d)
14:    return L
```

V. INTEGRATING MDM INTO CDCL

As already noted in Section III, DECIDE cannot always select multiple decisions, as the production of implications cannot indefinitely be avoided. The main question is therefore when MDM should be applied. To regulate this, we involve the *restarting mechanism* of SAT solvers, in particular the one used by MINISAT, as we use it in our own SAT solver as well.

In Alg. 3, at lines 2-8, an extended version of the CDCL procedure is given, with restarts. The RESTART procedure applies the chosen restart strategy, providing an upper-bound for the total number of conflicts that is allowed to occur in the SEARCH procedure before the next restart should happen. We have implemented both the geometric [20] and Luby [21] restarts, as used in MINISAT. In Section VI, we discuss their effect on solving when MDM is applied.

Although not listed here, SEARCH is extended to terminate once the conflicts limit *maxConflicts* is reached, and whenever this is the case, SEARCH returns $(0, \texttt{UNSOLVED})$. Every time SEARCH returns a value (line 6), the number of restarts is incremented. At line 7, we use *maxConflicts* again to periodically reset *R* to the constant *rounds*.

The DECIDE procedure (lines 9-14) calls MDM up to *rounds* times (line 11) after every reset of *R* (line 7). At line 13, if $\mathcal{L}$ is empty, either due to $R = 0$ or MDM producing an empty set, the procedure FOLLOWUPDEC is called, which implements standard decision making (i.e., single decision selection). If the overall solving procedure has just started ($\sigma = \emptyset$), the PUMPFROZEN procedure is called at line 12, to mitigate what we call the *overjump effect*. More on this below.

The intuition behind the periodic execution of MDM for a *rounds* number of times after a restart, is that there is maximum potential for selecting a large set of multiple decisions at the start of solving. After a number of MDM selections, the solver should start considering decisions that cause implications. The use of *maxConflicts* to regulate the resetting of R (line 7) may be less intuitive. We experience that not applying MDM after *each* restart is effective.

A. The overjump effect

Consider the multiple decisions set $\{x_1, x_8, x_9\}$ selected in Ex. 3. Assume that after MDM has selected this set, FOLLOWUPDEC is called by DECIDE, and that this procedure selects $\neg x_4$, as Fig. 1 shows. Since this leads to a conflict, assume that the binary clause $\{\neg x_7, \neg x_8\}$ is learned. It is concluded that all assignments up to level 1 must be undone (line 35 of Alg. 1), and therefore also decision x_9 at level 3 (see Fig. 1), even though it is not related to the conflict at all (in other words, x_9 is jumped over). Unnecessarily undoing decisions can in general negatively affect the solving procedure. Although this can occur in standard CDCL as well, suitable heuristics, such as VSIDS, mitigate this effect. They tend to favour sequences of decisions that are closely related, due to how α evolves over time, making it unlikely that a sequence such as the one in Ex. 3, with x_9 not being related to the other decisions at all, is made. Moreover, in standard CDCL, sequences of decisions without implications are not specifically stimulated, making it more likely that conflicts occur earlier (see Section VI), which in turn influences α.

On the other hand, when MDM makes a set of decisions, α can only evolve after the complete set has been selected, hence α has much less influence on the subsequent selection of the decisions when that set is constructed. A subsequent call of FOLLOWUPDEC may then select a decision that is (possibly transitively) dependent on any of the previously made parallel decisions, and at the start of the solving procedure, when all variables have $\alpha = 0$, the order in which parallel decisions are made is independent of α, and only depends on h.

To mitigate the overjump effect, we therefore have to stimulate at the start of solving that after making multiple decisions, the subsequent standard *follow up* decisions (FUDs) are dependent on the most recently made parallel decision, i.e., the one with the highest decision level. If no such decisions can be made, the ones dependent on the decision with the second highest level must be stimulated, and so on.

The prioritisation algorithm in Alg. 4 plays a vital role in this. Recall that it is called by DECIDE after the initial selection of decisions (line 12 in Alg. 3). It gives priority to the FUDs implied by the most recent parallel decisions. The **for** loop at lines 3-7 in Alg. 4 iterates over the decisions in $\mathcal{M}$, ordered by $\geq_\delta$, which is defined as $\ell \geq_\delta \ell' \triangleq \delta(\ell) \geq \delta(\ell')$. At line 4, a normalised value $0 < \beta \leq 1$ is computed for decision ℓ, based on the decision level of ℓ and the number of decisions made. We ensure that this value does not exceed 1, to prevent this prioritisation from interfering with the standard VSIDS activity. In the **for** loop at lines 5-7, the α of every variable referred to by a literal in a clause in which $\neg\ell$ is watched is increased (bumped) by β, if no activity has been recorded yet (line 8). In practice, the use of PUMPFROZEN is very effective: experiments on a random set of 200 formulas reveal a boost in performance by speedups of 21× at best, and 1.8× on average. This is remarkable, considering that PUMPFROZEN is only called at the start of the solving procedure. After that, the use of VSIDS is sufficient to mitigate overjumping.

Algorithm 4 Follow-Up Decisions Prioritisation

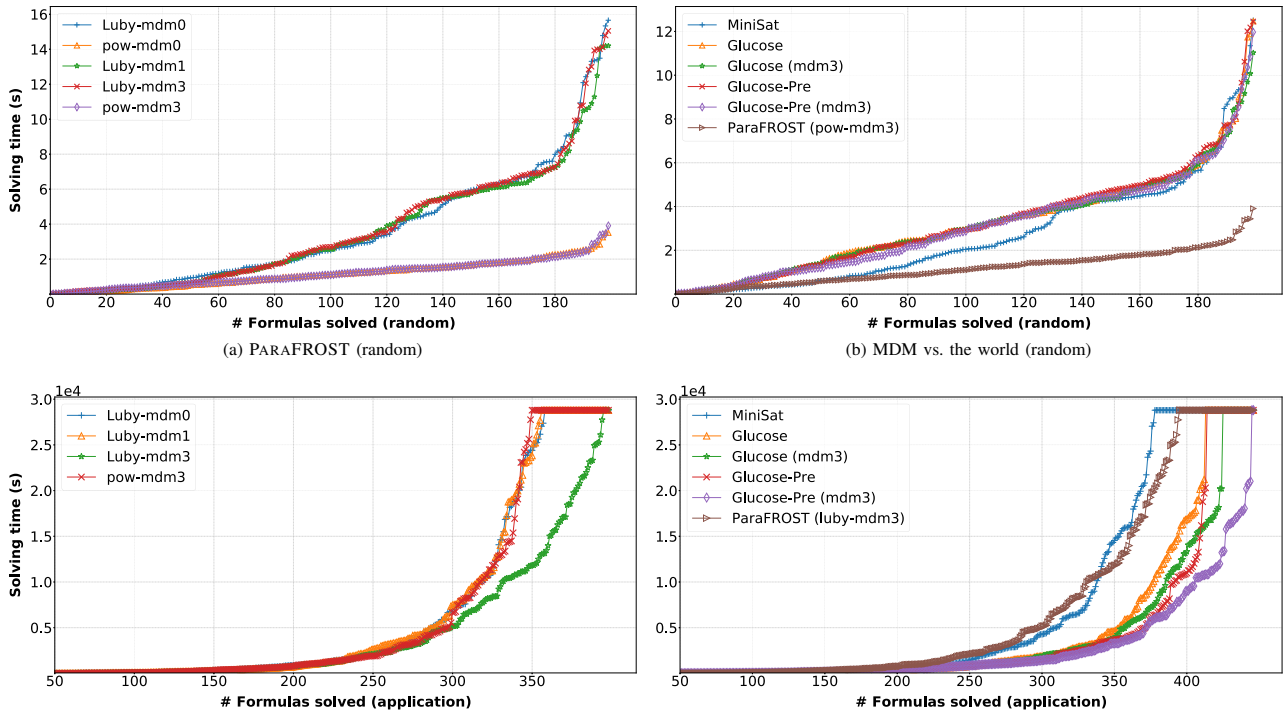
```
1: global vars: VSIDS variable activity α, dec. level function δ
2: procedure PUMPFROZEN(M)
3:     for ℓ ∈ SORT(M, ≥_δ) do
4:         β ← δ(ℓ)/|M|
5:         for all C ∈ C_¬ℓ with W_C(¬ℓ) do
6:             for all ℓ' ∈ C do
7:                 if v(ℓ') ≠ v(ℓ) ∧ α(v(ℓ')) = 0 then α(v(ℓ')) ← β
```

B. Correctness of applying MDM in CDCL

Finally, we can reason about the correctness of applying MDM in CDCL, i.e., that CDCL as in Alg. 3 returns SAT iff φ is satisfiable, and returns UNSAT iff φ is unsatisfiable. We argue that this is the case, by showing that each execution of CDCL with MDM, which, for clarity, we refer to as MDCL (Multiple Decision Clause Learning), can be matched by an execution of CDCL without MDM, or CDCL for short. MDCL and CDCL only differ in the fact that MDCL sometimes calls MDM, leading to a forward jump to a level higher than the current level. Clearly, every time DECIDE executes FOLLOWUPDEC in MDCL, CDCL can match the decision made. When DECIDE executes MDM in MDCL, multiple decisions $\ell_1, \ldots, \ell_k$ are made, each decision not leading to any implications, by Def. 2. Note in Alg. 2 that each time a decision ℓ_i ($1 \leq i \leq k$) is added to $\mathcal{M}$ at line 11, the condition at line 9 is satisfied, which implies that there are no unit clauses, but also that there is no clause C for which $C \models_{\sigma \cup \{\ell_i\}} \bot$. Selecting k decisions in Alg. 1 results at line 9 in $\vec{d}$ possibly being increased to some value higher than $\vec{d}+1$, which directly leads, by the conditions of lines 14 and 7, to $k-1$ calls of SEARCH, after which we have $d = \vec{d} - 1$, and BCP is called to check for conflicts at line 14. In CDCL, this can be matched by successively making the same decisions $\ell_1, \ldots, \ell_k$ in k calls of DECIDE at line 9 of Alg. 1. After each of the first $k-1$ decisions, BCP is called at line 14. and since no clause is unsatisfied and no unit clauses exist, the **while** loop in BCP is skipped, resulting in BCP returning **true**, and SEARCH being called again. After decision ℓ_k, BCP detects conflicts iff MDCL does as well, and if no conflicts occur, SEARCH is called again, which in turn calls DECIDE.

VI. BENCHMARKS

We implemented CDCL with MDM into a new solver called PARAFROST. It has data structures designed from scratch and implemented in C++, with the intention of parallelising SAT solving in the future. To check the $\mathcal{W}_C$ predicate, a *watch list* is maintained for each literal ℓ that stores pointers to clauses in which ℓ is watched. Similar to the LINGELING implementation, binary clauses are only referenced in the watch lists, and not in the clause database, to save memory. The VSIDS settings and the clause deletion policy for the reduction of conflict clauses are the same as in MINISAT. Additionally, we integrated MDM into GLUCOSE version 3 [13], to demonstrate the effectiveness of MDM in a state-of-the-art solver with and without preprocessing. The same GLUCOSE heuristics and settings have been used for all



(a) PARAFROST (random)

(b) MDM vs. the world (random)

(c) PARAFROST (application)

(d) MDM vs. the world (application)

Fig. 2: Runtime results for CDCL with and without MDM

experiments, both when using and not using MDM. When configuring the preprocessing, we noticed that GLUCOSE skips clause eliminations on problems having more than 4,800,000 clauses. We disabled this limitation to avoid redundant results w.r.t. non-preprocessing experiments. We refer to GLUCOSE with preprocessing as GLUCOSE-PRE. Both PARAFROST and GLUCOSE with MDM, together with all experimental results, are available online.[1]

While preparing the GLUCOSE experiments, we observed that controlling the frequency of calling MDM in DECIDE as in line 7 of Alg. 3 is not effective, since the restart strategy of GLUCOSE does not work with a fixed maximum number of conflicts. Therefore, in GLUCOSE-PRE, we have configured the periodic calling of MDM differently: whenever *restarts* mod $(nrConflicts/m) + 1 = 0$, R is reset to *rounds*. Since the total number of conflicts (*nrConflicts*) monotonically increases, we divide it by a monotonically growing divisor m before comparing *restarts* with it. After every 100,000 conflicts, m is incremented. Initially, $m = 1$. Tuning those parameters to an optimum is left for future work.

All experiments were conducted using the compute nodes of the DAS-5 cluster [22]. Each problem was analysed in isolation on a separate node. Each node has an Intel Xeon E5-2630 CPU running at 2.4 GHz with 128 GB of system memory, and runs on the CentOS 7.4 operating system. We performed the equivalent of 4 years of uninterrupted processing on a single node to measure how MDM impacts SAT solving when applied in state-of-the-art CDCL solvers.

We selected all the application problems from the 2013-2019 SAT competitions[2], excluding redundancies. These problems encode roughly 60+ different real-world applications, with various logical properties. The problem size ranges in this set from 14 KB up to 2 GB. Moreover, we selected 200 random problems ((u)uf-250) from the SATLIB library [23].

In the experiments, we compared MINISAT, version 2.2, with PARAFROST (with and without MDM), and we analysed the impact of MDM on GLUCOSE, with and without preprocessing. The time out for solving a formula in PARAFROST and MINISAT was set to 8 hours, while GLUCOSE was run with a timeout of 6 hours per formula.

Figure 2 gives runtime results for the various solvers. Cactus plots (a) and (c) show the impact of different modes in PARAFROST. Each mode in the legend is named `<restart>-mdm<rounds>`, with `restart` referring to the restart policy, which is either geometric (`pow`) or `luby`, and `rounds` defining MDM rounds per search (see Alg. 3). With `mdm3`, we refer to MDM with *rounds* $= 3$. Note that `mdm0` refers to not applying MDM at all. In practice, we observed that setting *rounds* to a value higher than 3 does not help to improve performance. Other settings, such as the ones for VSIDS, were kept the same for all experiments. Plots (b) and (d) compare PARAFROST with(out) MDM3 with

[1] https://gears.win.tue.nl/software/mdm.

[2] http://satcompetition.org.

TABLE I: SOLVABILITY OF MDM WHEN INTEGRATED TO PARAFROST AND GLUCOSE

Measurement	MiniSat		ParaFROST(MDM0)		ParaFROST(MDM3)		Glucose		Glucose(MDM3)		Glucose-Pre		Glucose-Pre(MDM3)	
	r	a	r	a	r	a	r	a	r	a	r	a	r	a
Solved	200	379	200	359	200	395	200	413	200	425	200	414	200	445
	(579) 65%		(559) 63%		**(595) 67%**		(613) 69%		**(625) 70%**		(614) 69%		**(645) 73%**	
MDM3 faster by	159	115	102	182	—		104	170	—		118	190	—	
	(274) 48%		**(284) 51%**				**(274) 45%**				**(308) 51%**			
MDM3 excess by	0	16	0	36	—		0	12	—		0	31	—	
	(16) 3%		**(36) 7%**				**(12) 2%**				**(31) 5%**			
Total time (hr)	2846.66		3026.79		**2822.07**		2532.35		**2450.55**		2457.21		**2306.62**	

TABLE II: STATISTICAL EVALUATION OF MDM IMPACT ON GLUCOSE-PRE FOR DIFFERENT FORMULAS

CNF	Glucose-Pre (MDM0)				Glucose-Pre (MDM3)					
	# FUDs	restarts	conflicts	t(s)	# FUDs	MDM Calls	# MDs	restarts	conflicts	t(s)
blockpuzzle_9x9_s1_free10	3,978,815	5,367	3.5479E+10	94	210,979	3	1,357	269	2.3934E+10	**3**
toughsat_26bits_2	10,629,037	18,140	9.1326E+11	2,960	1,419,209	3	209	2,704	2.0983E+12	**155**
q_query_3_L200_coli.sat	13,821,925	33,817	7.3463E+06	1,121	1,021,637	7	913,832	1,948	5.7789E+05	**71**
sha1r17m75a_p	4,110,551	2,566	3.6750E+13	696	826,056	3	1,309	996	1.7752E+11	**54**
mp1-klieber2017s-0300-034-t12	13,704,827	7,933	3.4034E+15	3,707	5,131,866	3	17,991	1,184	3.8672E+12	**1,056**
gss-20-s100	1,927,134	908	1.1796E+13	1,303	251,631	3	22,550	159	2.1659E+12	**141**
mp1-9_49	9,076,146	17,438	2.3723E+12	1,025	1,799,351	9	187	2,116	7.2529E+11	**114**
partial-10-19-s	125,449,003	135,634	6.1101E+13	20,300	28,704,006	57	986,934	24,578	2.1182E+12	**2,550**
mp1-rubikcube912	42,716,017	157,177	3.7654E+07	10,180	30,377,794	19	7,067	52,701	2.7644E+07	**5,911**
cube-11-h13-unsat	287,186	399	8.3493E+04	146	208,103	3	444,667	257	5.2369E+04	**86**
size_5_5_5_i223_r12	38,684,114	69,426	2.1244E+13	6,055	15,004,798	52	8,623	27,859	2.5824E+14	**977**
transport-1city-35n-40	14,777,489	1,483	1.5311E+15	1,240	4,825,914	3	1,309,549	456	1.5641E+15	**387**
Mickey_out250_known_last146	2,893,224	2,725	1.3714E+10	54	97,517	3	49,320	229	2.3611E+10	**11**
hwmcc15deep-oski15a14b04s-k16	30,081,389	27,092	7.3587E+06	869	7,873,390	9	525,385	6,065	1.4810E+06	**190**
UCG-20-5p1	1,810,808	838	6.1661E+14	535	728,566	3	95,894	436	2.7468E+12	**126**
GracefulGraph-K04-P04_c18	2,554,285	5,449	3.3113E+12	104	930,211	9	5,851	1,869	3.4781E+12	**25**
reachsafety.newton_2_6	334,056	424	1.4703E+12	19	136,621	3	64,006	142	1.5128E+10	**6**
mizh-md5-48-2	30,817,468	29,698	1.4822E+12	592	16,993,726	67	241,779	13,964	1.2828E+12	**206**
reachsafety.triangular	5,650,490	9,018	1.9239E+11	187	3,849,193	5	25,179	3,024	1.9951E+13	**99**
test_v7_r17_vr5_c1_s25451	39,280,415	140,081	1.8042E+07	19,716	20,668,772	79	6,787,242	76,248	8.2700E+06	**10,670**
9vliw_m_9stages_iq3_C1_bug9	4,456,766	195	2.8021E+12	72	1,033,139	3	403,335	30	1.7197E+10	**40**
mp1-klieber2017s-0490-023-t12	389,796	154	4.5573E+12	74	229,568	3	28,870	35	1.9371E+11	**41**
MASG0_72_keystream76_0	16,316,939	5,551	4.8628E+15	6,193	12,636,873	3	3,181	11,778	2.5498E+15	**3,682**
sokoban-p20.sas.cr.21	2,206,126	2,142	4.1048E+05	230	1,639,150	8	96,892	1,494	2.9162E+05	**146**
ibm-2002-23r-k90	3,737,246	4,437	1.3415E+12	130	2,894,517	13	371,158	3,122	3.3715E+12	**83**
slp-synthesis-aes-top28	61,329,961	154,464	2.4132E+14	9,707	44,840,257	187	1,875,092	110,562	7.7041E+15	**6,229**
cruxmiter29seed4	21,114,991	19,908	1.9758E+07	3,287	15,143,324	3	387	21,212	1.4036E+07	**2,126**
transport-1city-35n-50	21,593,311	1,923	1.0441E+15	1,708	12,258,223	3	1,634,247	984	1.0932E+15	**1,125**
ak128boothbg2asisc	47,192,431	35,696	1.1411E+10	3,083	33,471,429	11	484,943	25,086	4.4297E+10	**2,037**
ASG_72_keystream76_2	8,038,821	4,527	2.5829E+13	2,247	5,871,604	3	3,439	2,722	1.0757E+14	**1,525**
ACG-20-5p1	922,937	1,059	7.8870E+10	317	717,435	3	105,373	444	2.2829E+12	**223**
g2-modgen-n200	31,960,054	40,581	1.3276E+15	12,032	32,133,297	23	5,937	39,840	1.1062E+14	**8,539**
9dlx_vliw_at_b_iq2	9,899,952	1,843	6.3482E+05	37	8,098,849	9	100,323	1,602	4.3828E+05	**27**
test_v7_r7_vr5_c1_s14675	1,120,842	851	1.5723E+11	35	887,532	5	248,969	613	8.4923E+11	**26**
CarSequencing-90-02_c18	10,676,128	3,206	1.4912E+13	59	7,671,459	17	212,123	1,920	8.5121E+12	**45**
Eternity-10-05_c18	899,714	1,030	2.5314E+09	24	815,214	5	3,737	729	1.4429E+10	**18**
T50.2.0	18,452,981	2,350	7.0158E+05	2,465	15,674,218	31	31,740,848	1,909	6.1158E+05	**1,915**
hwmcc15deep-intel065-k11	44,751,749	39,042	2.2911E+07	1,533	39,218,889	14	205,636	30,459	2.0185E+07	**1,288**

MINISAT, and GLUCOSE with(out) MDM3 with GLUCOSE-PRE with(out) MDM3. For PARAFROST, we used the best restart modes as indicated by plots (a) and (c).

In the majority of cases, enabling MDM3 positively affects the runtime in PARAFROST. When solving random problems, the geometric restart policy is most effective. These problems are very small in size, resulting in a small number of conflicts, which favours long restart intervals. In contrast, the `luby` policy works well for the application problems due to frequent restarting. Using the best restart policy together with MDM3 causes PARAFROST to outperform MINISAT (plots (b), (d)). When used in GLUCOSE and GLUCOSE-PRE, MDM3 has little impact on solving random problems (plot (b)). However, for application problems, it clearly impacts the performance of both GLUCOSE and GLUCOSE-PRE very positively (plot (d)).

Table I summarises the solvability of all solvers. The letters `r` and `a` refer to *random* and *application*, respectively. The `Solved` row gives per solver the number of solved problems per problem type, followed by the total number of solved problems and their percentage. For example, PARAFROST(MDM3) solved 395 application problems and a total of 595, which is 67% of the 895 problems. The `MDM3 faster by` row evaluates the number of problems solved faster when MDM3 is used in PARAFROST, compared to not using MDM3 and MINISAT, and when using MDM3 in GLUCOSE, compared to not using it in GLUCOSE. The same applies to GLUCOSE-PRE. For instance, GLUCOSE-PRE(MDM3) solved 308 problems faster than GLUCOSE-PRE, which is 51% (308/614) of problems solved by GLUCOSE. Similarly, the third row indicates how many *extra* problems were solved when MDM3 was used. The last row accumulates the solving time of all problems (including timeout cases) in hours for each solver. For instance, the total time of GLUCOSE-PRE(MDM3) is the minimum among all solvers, saving roughly 150 hours (6%) compared to GLUCOSE-PRE. Overall, it is clear that MDM positively affects solving times and works very well when preprocessing is enabled.

Table II presents the impact of MDM3 on various aspects

of CDCL on solving a sample of 38 formulas, with MDs referring to Multiple Decisions, i.e., decisions that were part of a multiple decisions set. In general, the number of restarts and the number of conflicts are drastically reduced by MDM3, which seems to be the main cause for its effectiveness in CDCL. The conflict clauses produced seem to be of higher quality, better helping CDCL to make good FUDs, and reduce the number of implications and potential conflicts.

VII. Conclusion

We have proposed how to generalise CDCL to allow making multiple decisions at once, and have integrated *multiple decision making* (MDM) into the CDCL algorithm. Furthermore, we have demonstrated the effectiveness of MDM in practice, in particular when solving structured application problems.

Concerning future work, our results motivate us to study the impact of different heuristics and restart policies. We believe there is room for improvement to boost the performance of CDCL solving, and will continue investigating this.

References

[1] J. Marques-Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, March 1999, pp. 145–149.

[2] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1167–1176, 1996.

[3] M. Osama, L. Gaber, A. I. Hussein, and H. Mahmoud, "An Efficient SAT-Based Test Generation Algorithm with GPU Accelerator," *Journal of Electronic Testing*, vol. 34, no. 5, pp. 511–527, Oct 2018.

[4] C. E. Brown, "Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems," *Journal of Automated Reasoning*, vol. 51, no. 1, pp. 57–77, Jun 2013.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS 1999*. Springer, 1999, pp. 193–207.

[6] A. R. Bradley, "SAT-based model checking without unrolling," in *VMCAI 2011*. Springer, 2011, pp. 70–87.

[7] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[8] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC 2001*. ACM, 2001, pp. 530–535.

[9] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *ICCAD 2001*. IEEE Press, 2001, pp. 279–285.

[10] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *SAT*, ser. LNCS, vol. 2919. Springer, 2004, pp. 502–518.

[11] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver," *Discrete Applied Mathematics*, vol. 155, no. 12, pp. 1549 – 1561, 2007.

[12] A. Biere, "Lingeling, plingeling, picoSAT and precoSAT at SAT race 2010," Johannes Kepler University, FMV Report 1, 2010.

[13] G. Audemard and L. Simon, "Predicting Learnt Clauses Quality in Modern SAT Solvers," in *IJCAI 2009*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[14] N. Eén and A. Biere, "Effective Preprocessing in SAT Through Variable and Clause Elimination," in *SAT*, ser. LNCS, vol. 3569. Springer, 2005, pp. 61–75.

[15] M. Osama and A. Wijs, "Parallel SAT Simplification on GPU Architectures," in *TACAS*, ser. LNCS, vol. 11427. Cham: Springer International Publishing, 2019, pp. 21–40.

[16] ——, "SIGmA: GPU Accelerated Simplification of SAT Formulas," in *Integrated Formal Methods*, W. Ahrendt and S. L. Tapia Tarifa, Eds. Cham: Springer International Publishing, 2019, pp. 514–522.

[17] M. Järvisalo, M. J. Heule, and A. Biere, "Inprocessing Rules," in *IJCAR*, ser. LNCS, vol. 7364. Springer, 2012, pp. 355–370.

[18] K. Pipatsrisawat and A. Darwiche, "A Lightweight Component Caching Scheme for Satisfiability Solvers," in *SAT 2007*, ser. LNCS, vol. 4501. Springer, 2007, pp. 294–299.

[19] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[20] T. Walsh, "Search in a small world," in *IJCAI 1999*. ACM, 1999, pp. 1172–1177.

[21] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal speedup of Las Vegas algorithms," *Information Processing Letters*, vol. 47, no. 4, pp. 173–180, 1993.

[22] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *IEEE Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[23] H. H. Hoos and T. Stützle, "SATLIB: An Online Resource for Research on SAT," in *SAT 2000*. IOS Press, 2000, pp. 283–292.