SAT-Hard: A Learning-based Hardware SAT-Solver

Buse Ustaoglu[1] Sebastian Huhn[1,2] Frank Sill Torres[1,2] Daniel Große[1,2] Rolf Drechsler[1,2]

[1]Cyber Physical Systems, DFKI GmbH, Bremen, Germany
[2]Group of Computer Architecture, University of Bremen, Germany
buse.ustaoglu@dfki.de huhn,frasillt,grosse,drechsle@cs.uni-bremen.de

Abstract—**Within the last decades, tremendous research work has been carried out on the development of software-based algorithms to solve the Boolean Satisfiability Problem. These SAT-solvers have then been heavily orchestrated for addressing complex computational tasks like the verification of circuits. In this field, most of the applied techniques focused only on the design phase of the circuit. Due to this fact, new approaches have been published in the literature solely focusing on online verification as well as self-verification. These kind of solutions strictly require Hardware (HW) SAT-solvers that can be integrated into a system while introducing only low hardware overhead and still providing high flexibility. By following these observations, this work presents SAT-Hard: In contrast to the state-of-the-art, SAT-Hard takes advantage of learning techniques to support features like clause learning and non-chronological backtracking, and combines them within a lightweight and standalone HW device. By this, a run-time speed-up of 2,000x can be achieved. Furthermore, the experimental evaluation clearly demonstrates that those complex problems can be solved in less than 20 seconds. Particularly due to its compactness, SAT-Hard is suitable for self-verification that enables the continuous verification of an integrated system during its lifetime.**

Index Terms—**Hardware SAT-Solver; hardware based conflict-driven clause learning; online-verification; on-chip self-verification package**

I. Introduction

The *Boolean Satisfiability Problem* (SAT-problem) is used in many areas of computer science such as computer-aided design, circuit design, automated reasoning as well as formal verification. The verification of circuit designs is an essential task and ensures that the functionality of the design meets its specification. Verification is of uttermost importance for safety-critical applications. The steadily increasing design complexity of electronic systems paves the way for the long process of verification. Due to tight time-to-market constraints, most of the designs are deployed without being fully verified. Self-verification has been introduced as a solution in order to close this verification gap [1], [2], enabling to continue the design verification tasks on-chip after deployment by making use of an embedded verification package. Online verification is a similar approach that verifies produced results at run-time in order to continuously guarantee the integrity of the system. Both solutions require fast and powerful solvers with low costs in terms of area and power. Only if these criteria are both fulfilled, such a solver can be considered as part of the verification package. Since there are fundamental differences in the development of hardware (HW) and software (SW) solutions for specific problems, the transfer of existing software SAT-solvers to the hardware domain is a non-trivial task. In previous work, a basic HW SAT-solver [3] has been developed as a verification package and been implemented entirely on hardware in order to solve arbitrary SAT-instances which are dynamically generated on-chip. However, it suffers from long run-times when solving highly complex instances and, hence, turning it inappropriate for several applications.

This work presents SAT-Hard, a learning-based and standalone HW SAT-solver. SAT-Hard significantly extends the HW SAT-solver [3] by enabling conflict-driven clause learning in HW. SAT-Hard is still completely implemented in HW and allows solving arbitrary instances solely in HW. By this, SAT-Hard achieves a run-time speed-up of 2,000x while keeping the required HW costs low. Furthermore, the orchestration of these learning-based techniques allows solving SAT-instances which could not be handled by the state-of-the-art approach. Consequently, SAT-Hard provides the required performance to tackle the upcoming challenges in the domain of self-verification.

The remainder of this paper is organized as follows: Section II gives the background on SAT-problem and SAT solving algorithms. Section III reviews and classifies existing hardware SAT-solvers. Section IV introduces SAT-Hard: A learning-based HW SAT-solver. The experimental results are evaluated in Section V. Finally, Section VI concludes this work.

II. Background in SAT solving

This section reviews the general SAT-problem and presents algorithms that enable SAT solving. In particular, learning-based approaches are discussed in more details, which form the basis for this work.

A. SAT-problem

The SAT-problem is one of the central $\mathcal{NP}$-complete problems. In fact, it was the first known $\mathcal{NP}$-complete problem [4]. Despite its proven complexity, solving the SAT-problem is a fundamental aspect of countless optimization methods, many of them in the field of computer-aided design.

The SAT-problem is defined as follows: Let f be a Boolean function in *Conjunctive Normal Form* (CNF), i.e., a product-of-sum representation. Then, the SAT-problem is to find an assignment for the variables of f such that f evaluates to '1', or to prove that no such assignment exists.

The CNF consists of a conjunction of clauses. A *clause* is a disjunction of literals and each *literal* is a Boolean variable or its negation. A CNF formula is satisfied iff for at least one variable assignment all clauses are satisfied. A clause is satisfied if at least one of its literals is satisfied and a literal is satisfied if it evaluates to ‘1’, which is demonstrated by the following example.

Example 1: Let $f = (x_1 + x_2) \cdot (\overline{x}_2 + x_3)$. Then, $x_1 = 1$ and $x_2 = 0$ is a satisfying assignment for f, i.e., f is satisfiable.

The *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm [5], [6] is a backtracking-based SAT algorithm that acts as the basis for state-of-the-art SAT-solvers like [5]–[9]. A fundamental aspect of the DPLL-algorithm concerns the use of the *Unit Propagation*, also known as *Boolean Constraint Propagation* (BCP) [8]. By definition, a clause is a *unit clause* if only one of its literals has no satisfying assignment. Then, the variable of the literal is implied or forced to a logic value such that the clause is satisfied. A conflict occurs if this implication is contradictory to the previous variable assignment. The DPLL algorithm tries to solve such a conflict by reverting the assignments from previous decisions, i.e., via chronological backtracking.

Fig. 1a details the basic flow of DPLL. In general, the SAT-problem is satisfiable, iff all variables could be assigned to a valid value (Step II). If there are free variables left (Step I) a free variable is assigned to a value (Step III), which is also called *Decision*. In parallel, the *Decision Level*, i.e., the number of decisions during the solving process, is increased. Next, a value is implied to the variable based on the previous *Unit Propagation* (Step IV). If this step does not cause a conflict (V), then the procedure restarts at (Step I). In case of a conflict, it is checked whether the decision level is higher than 0 (Step VI). If this is the case, the algorithm executes a backtrack to the immediate level and forces an opposite value to the specific variable of the decision level. If the conflict still exists at the decision level 0, then the problem is said to be unsatisfiable, usually denoted as UNSAT (Step VIII).

B. Conflict-Driven Clause Learning

The plain DPLL algorithm only allows chronological backtracking might be leading to a state in which the HW SAT-solver gets trapped in non-solution regions of the search space. The *Conflict-Driven Clause Learning* (CDCL) is an extension of the DPLL algorithm, which tries to prevent this issue by determining the reason for the conflict using implication graphs, clause learning and non-chronological backtracking [8].

An *Implication Graph* (IG) is a skew-symmetric directed graph containing the variable assignments associated with the corresponding decision level. The vertexes of an IG represent a variable assignment, while the directed edges indicate the implications and the related clause. Furthermore, decisions have no incident edges. The IG is constructed during the decision and unit propagation phases (see Fig. 1).

A significant share of solving time is spent on the backtracking of unsatisfying variable assignments. CDCL allows learning of new clauses, which help to avoid unfavorable partial variable assignments and restrict the search space in areas, where no solutions can be found. The principle flow of the *Clause Learning* is depicted in Fig. 1b, which starts after a conflict has been identified. First, the IG is cut into two partitions such that one side contains the conflict while the other one holds all decisions and implications leading to this conflict (2). Next, the partial assignment, formed by all variable assignments that have an outgoing edge crossing the cut, is extracted (3). Then, the partial assignment is negated and added as a newly learned clause to the problem instance (4). Finally, the backtrack level is determined (5). In contrast to the plain DPLL algorithm, more than one level can be returned, which is also called *Non-chronological Backtracking*. Here, the appropriate decision level (to be considered for backtracking) is the highest decision level, except the current one, in the learned clause. This strategy has an advantage since the learned clause becomes unit (or assertive) under the resulting partial assignment [10].

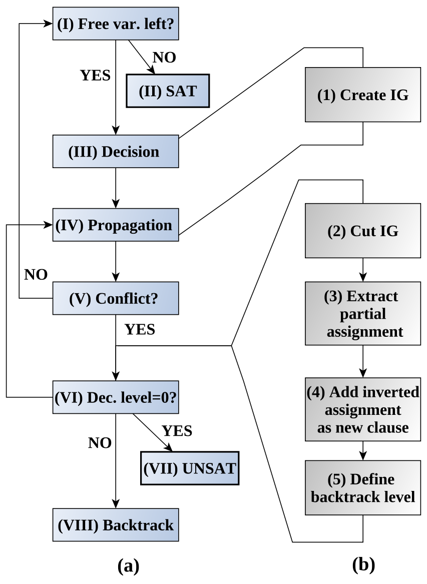


Fig. 1. (a) Basic steps of DPLL flow and (b) enhanced with Conflict-Driven Clause Learning

From a functional point of view, a sequence of resolution operations is performed on clauses during the clause learning procedure [11]. A new temporary clause is generated at each step. Let $\odot$ represents the resolution operator. For two clauses w_j and w_k, for which a unique variable x exists such that one clause has a positive literal x and the other one has negative literal $\overline{x}$, $w_j \odot w_k$ contains all the literals of w_j and w_k with the exception of x and $\overline{x}$. If the current decision level contains just one variable (in any generated clause), the resolution operation will terminate with the learned clause as the result. The literals of the learned clause are the reason for the conflict; the so-called *reason literals*.

The following example shall give a more detailed insight into the CDCL:

Example 2: Fig. 2a depicts a CNF consisting of the five clauses w_1 to w_5. In order to represent the decision and propagation process, the following notation is used: $x_i = b@n$ with x_i is the variable, the logic value b is assigned to at decision level n. For example, $x_1 = 0@1$ means that at decision level 1 the variable x_1 was assigned to the value '0'. Fig. 2b shows the resulting implication graph. The graph indicates that the implications on x_5 in w_1 and x_6 in w_2 at decision level 4 and on x_7 in w_5 at decision level 2 turn the clause w_3 unsatisfied, i.e., all literals evaluate to '0'. The resulting conflict is denoted by *C*.

The conflict analysis starts with the unsatisfied clause w_3. In the first step, the resolution operation is executed on w_3 and w_2, which is one of the clauses at decision level 4 as follows:

$$\underbrace{(\overline{x}_5 \vee \overline{x}_6 \vee \overline{x}_7)}_{w_3} \odot \underbrace{(x_4 \vee x_6)}_{w_2} = \underbrace{(x_4 \vee \overline{x}_5 \vee \overline{x}_7)}_{w_T} \qquad (1)$$

The resulting temporary clause w_T includes all literals of both clauses, except x_6, since x_6 occurs negated in w_3 and non-negated in w_2. In the next step, the resolution operation is executed on w_T and w_1 as follows:

$$\underbrace{(x_4 \vee \overline{x}_5 \vee \overline{x}_7)}_{w_T} \odot \underbrace{(x_1 \vee x_4 \vee x_5)}_{w_1} = \underbrace{(x_1 \vee x_4 \vee \overline{x}_7)}_{w_L} \qquad (2)$$

The resulting clause w_L contains all literals of both clauses except x_5 and, similarly, x_5 occurs negated in w_T and non-negated in w_1. The resulting clause is learned clause because it holds the reason literals and it has just one literal at the decision level 4. This clause is added to the CNF problem (see Fig. 2c). The corresponding cut of the implication graph is depicted in Fig. 2b.

The variable assignments at the decision levels 3 and 4 are invalidated during the final backtracking since x_7 has the highest decision level apart from the current decision level in the learned clause.

In the remainder of the SAT solving process, x_4 is implied to '1', decisions made on x_3 with the assignment '0' and x_5 with the assignment '1' proving that this CNF problem is satisfiable. The final implication graph is depicted in Fig. 2d.

III. Hardware SAT-solvers

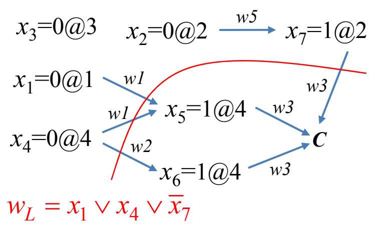
This section gives an overview of existing SAT-solvers, which are implemented in hardware. Furthermore, it classifies existing approaches and distinguishes these against SAT-Hard.

HW SAT-solvers can be broadly grouped into *Instance Specific* and *Application Specific* solutions [12]. In the case of the former, each and every instance is contained in the hardware design in a hard-coded fashion. This means that the solver has to be recompiled and reconfigured for each problem individually before the solver can be executed. In contrast to this, application specific SAT-solvers are configured only once and allow to process arbitrary problem instances. The solving process of such a SAT-solver can be solely implemented in hardware or it is separated in a hardware (HW) as well as in a software (SW) part.

The authors of [13], [14] present an instance specific solver implementing an improved DPLL algorithm. Here, the variables are randomly sorted during the automatic generation of the HDL code [9]. However, a principal drawback of such

$$\begin{aligned} w_1 &= (x_1 \vee x_4 \vee x_5)\\ w_2 &= (x_4 \vee x_6)\\ w_3 &= (\overline{x}_5 \vee \overline{x}_6 \vee \overline{x}_7)\\ w_4 &= (x_3 \vee x_{10})\\ w_5 &= (x_2 \vee x_7) \end{aligned}$$

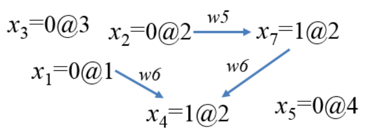
(a) Initial CNF instance



(b) Implication graph after a conflict has occurred

$$\begin{aligned} w_1 &= (x_1 \vee x_4 \vee x_5)\\ w_2 &= (x_4 \vee x_6)\\ w_3 &= (\overline{x}_5 \vee \overline{x}_6 \vee \overline{x}_7)\\ w_4 &= (x_3 \vee x_{10})\\ w_5 &= (x_2 \vee x_7)\\ w_L &= (x_1 \vee x_4 \vee \overline{x}_7) \end{aligned}$$

(c) CNF with learned clause



(d) Resulting implication graph

Fig. 2. Exemplary CNF problem with related implication graphs

a SAT-solver concerns the limitation of the problem size that can be solved with respect to the available hardware resources.

In the case of application specific SAT-solvers, one can distinguish between SW-driven solvers that require certain pre-processing steps that have to be executed in software, and stand-alone solvers, that execute all tasks solely on hardware. The authors of [15]–[18] present several SW-driven HW solvers, which are able to solve arbitrary problems of a large problem size. However, the solutions are based on incomplete search algorithms, like Walk SAT [19], which do not guarantee to determine whether the problem is unsatisfiable. This limitation does not apply for the SW-driven HW SAT-solver as presented in [20], which invokes a complete search algorithm. During the pre-processing of the CNF problem, clause literals are transformed into constants by exploiting the constant propagation optimization capabilities of the HDL compiler.

The authors of [3] present a stand-alone HW-solver, which introduces only very low costs in terms of area and power dissipation and, hence, can be easily integrated as a co-processing unit in complex systems. Besides the classification in instance and application specific solutions, one can also distinguish between SAT solvers with and without learning capabilities. As discussed in the previous section, the orchestration of learning-based techniques is essential for solving complex problems while occupying only manageable resources in terms of time and space. This means that any HW SAT-solver, which shall also be applied for more difficult problems should, or even has to, support learning techniques.

In [21], an instance specific solver is presented, which implements the learning of clauses by reverting the implication process. Each newly learned clause is then integrated as a new circuitry into the hardware. The authors of [22], [23] propose an instance specific solver that uses so-called banks to store the clauses and the implication graph. Furthermore, non-chronological backtracking and clause learning are supported but with considerably high area costs, i.e., an application-specific integrated circuit with a size of $2.25cm^2$ using a $100nm$ technology node. In [24], this solver has been expanded to a SW-driven application specific solver. Here, the SAT problem is heuristically partitioned into smaller CNFs, so-called *bins*, before it is stored into on-chip memory but again at high area costs. A further instance specific SAT solver is presented in [25], which uses conflict-directed jumping. The solver supports non-chronological backtracking and clause learning but is limited to smaller problems with nearly 500 literals at maximum.

All reported HW SAT-solvers, with exception of the one presented in [3], have high costs in terms of area and power and, consequently, are appropriate candidates for on-chip accelerators if there are no sharp limitations concerning the available hardware resources. In the field of self-verification as well as online verification, tough limitations in terms of hardware resources exist and, thus, the discussed HW SAT-solvers, again expect [3], are not applicable at all. Furthermore, in-field applications require fast solutions that are instance-agnostic. Hence, we propose in this work a lightweight application specific stand-alone HW SAT-solver, which supports both clause learning as well as non-chronological backtracking.

IV. SAT-Hard: CDCL-based Lightweight Hardware SAT-solver

This section introduces a lightweight application specific stand-alone HW SAT-solver, named *SAT-Hard*, which supports CDCL. In contrast to the state-of-the-art, this solver has notably low area requirements, and thus, can easily support self-verification and similar tasks. SAT-Hard is inspired by an existing basic HW SAT-solver that does not support CDCL and is introduced in the first part of this section. Next, the memory model as well as the control of SAT-Hard is presented.

A. Basic HW SAT-solver

A basic application specific and stand-alone HW SAT-solver has been presented in work [3], which is briefly introduced in the following. One fundamental component of this basic SAT-solver is the scalable memory model as used for storing the SAT-instance, which is formulated as a CNF (see Fig. 3). As commonly done, the literals are encoded in the DIMACs scheme, which is introduced in the following example.

Example 3: The DIMACs encoding of the two clauses of function f of example 1 is as follows:
$f = (x_1 + x_2) \cdot (\overline{x}_2 + x_3) \rightarrow$ `1 2 0, -2 3 0`.
This means that a variable x_i is represented by its index i, which is, in case of negation, multiplied by -1. Furthermore, a '0' encodes the end of a clause.

The memory model of the basic HW SAT-solver is shown in Fig. 3a. On the left-hand side, the SAT-instance is given in its original form. The *CNF memory* is shown in the middle part and, finally, the *Clause Position Memory* (CPM) is presented on the right-hand side of Fig. 3a. The CNF memory contains the clauses and their literals, whereat every literal is encoded following the scheme of Fig. 3b. Furthermore, the clauses are separated by a wordline, which stores the number of literals and *maximum decision level* of the preceding clause, i.e., $max\_level(x_i)$ with $x_i \in w$. The CPM contains the maximum decision level of the clause, the *Clause SAT Status* (CSS), indicating whether the clause is already satisfied or not. Furthermore, the CPM contains the clause address, i.e., the memory address pointing beyond the last literal of the clause.

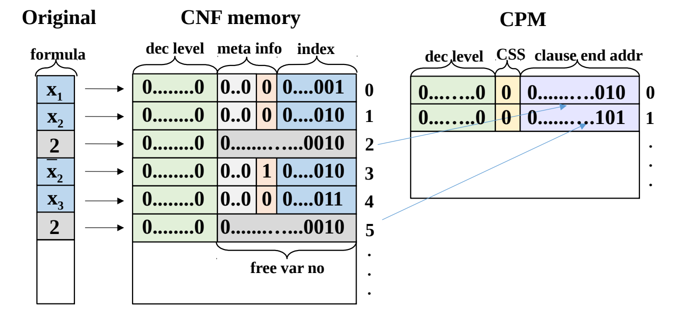
The encoding of every literal in the CNF memory is based on a 32 bit word as illustrated in Fig. 3b. The lower bits 0 to 13 indicate the index of the variable that belongs to the literal, bit 14 keeps the sign, bits 15 and 16 define whether the literal is free or assigned to '0' or '1'. Bit 17 shows whether the variable has been assigned already to both values ('0' and '1') for conflict resolution, bit 18 contains the information whether the assignment to the literal results from a decision of the SAT-solver or an implication. The remaining bits 19 to 31 represent the decision level.

The design of the basic HW SAT-solver employs the DPLL algorithm and invokes a finite state machine. After initialization of the solver, a complete SAT-instance is transferred to the device and stored in the CNF memory. Concurrently, the literal number of each clause is determined and written to the location of the wordline separator and, additionally, this address is also written to the CPM.

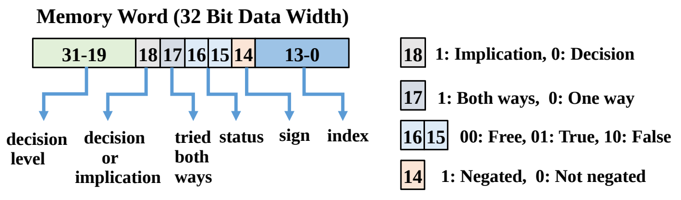
When the transfer of the SAT-instance is completed, a literal is determined from a clause, which is not yet satisfied and still holds free variables. Then, a decision assignment is made on this literal such that it evaluates to true and the decision level is increased by 1. If a clause has just one unassigned literal and is not yet satisfied, the free literal will be assigned by a newly derived implication. After an assignment has been conducted by a decision or an implication, the value is propagated to all clauses that include the literal and, if so, the counter for free variables is decreased within the specific clause. If the current assignment satisfies a clause, the CSS bit in the CPM is set to 1. If all the clauses are satisfied, the instance is said to be SAT. If no conflict has occurred and a clause is found, which holds only one free literal (and not yet satisfied), an implication is derived after current propagation has finished. Otherwise, a decision is made for a further literal. If a clause has no free variables and is not yet satisfied, a conflict occurs. To resolve the conflict, the SAT-solver flips the latest decision assignment and invalidates all implications of the current decision level. The affected clauses are identified by checking the maximum decision level information. If a clause has a literal at the current decision level, its number of free variables will be updated. As long as the conflict has not yet been resolved, the decision level is repeatedly decreased by 1 and the chronological backtracking principle is invoked. If the root level (decision level 0) has been reached and the conflict remains, the instance is said to be UNSAT.

B. Advanced Memory Model

The advanced memory model is developed to enable the conflict-driven clause learning, depicted in Fig. 4, and extends the basic memory model by an *Implication Graph Memory* (IGM) as well as a *Reason Literal Memory* (RLM). The



(a) Memory model containg the clauses of Example 1


Memory Word (32 Bit Data Width)
31-19
18
17
16
15
14
13-0
decision level
decision or implication
tried both ways
status
sign
index
18 1: Implication, 0: Decision
17 1: Both ways, 0: One way
16 15 00: Free, 01: True, 10: False
14 1: Negated, 0: Not negated

(b) Literal encoding

Fig. 3. Memory structure and literal encoding of basic HW SAT-solver

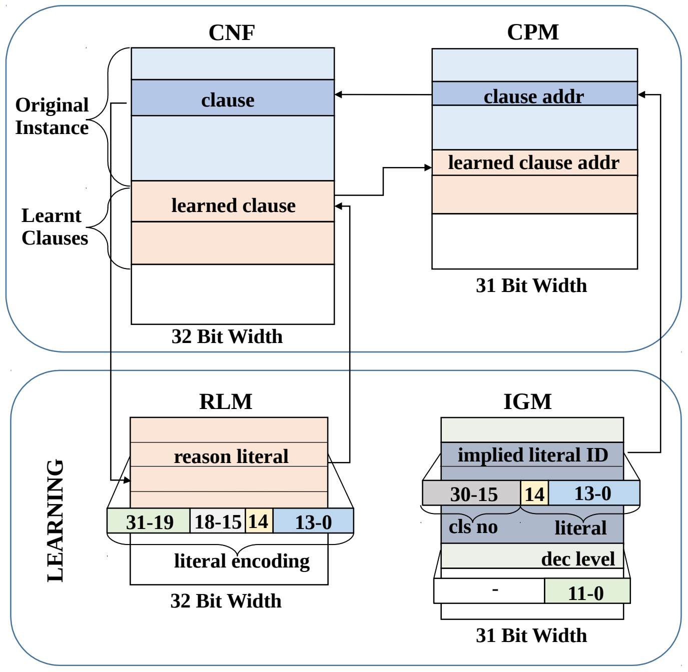


Fig. 4. Advanced memory model supporting CDCL

IGM represents the implication graph that stores all implications. This means that each wordline contains a variable index, a sign (bits 14 to 0) and the clause number, to which the implied literal belongs to (bits 30 to 15). Furthermore, a wordline of the IGM stores the decision level such that all implications of the same decision level are covered.

The RLM stores the literals of the clauses that contain reason literals causing a conflict. These literals, which follow the same encoding as shown in Fig. 3b, can be assigned by an implication or by a decision. During the learning process, as detailed in Section IV-C, all implied literals at the current decision level–except one– will be eliminated by the so-called resolution operation. Especially, the CNF memory is extended such that it can store learned clauses, whose addresses are then stored in the CPM.

C. Clause Learning

The solving process is quite similar to the methodology of [3] provided that no conflict occurs. One deviation concerns the fact that further information is gathered for every implication, i.e., the implied literal is written to the IGM. The significant extension of this paper mainly addresses the conflict resolution of SAT-Hard, which now involves sophisticated learning-based techniques. Fig. 5 presents an activity diagram, which describes the procedure after a conflict has been detected as follows: After detection of a conflict (1), all literals of the clause that contains the conflict, in the remainder named *Conflict Clause*, are stored in the RLM (2). Next, using the clause number, which has been just stored in the IGM and the address of the clause in the CPM, the implication clause, in which the literal is implied, is accessed in the CNF memory (3). This means that all literals of the implication clause are read from the CNF memory sequentially (4) and then are compared with all literals stored in the RLM using the literal index (5). If the literal is not yet stored in the RLM, then it is written to the RLM (7). Furthermore, if the corresponding decision level of the literal is equal to the highest decision level, then the counter will be incremented, which monitors the number of literals of the highest decision level. If the literal is already included in the RLM, it will be checked whether it is an implied one and whether its negation is already stored in the RLM (8). In the positive case, both literals eliminate each other (9), based upon the principle of the resolution operations (see also Example 2). Additionally, the counter for the corresponding decision level is decremented. If the literal cannot be eliminated but is already part of the RLM, the algorithm will proceed with the next step (10). The reading from the implication clause is repeated until all literals have been processed (11). If after this process more than one literal of the current decision level remains in the RLM, then the IGM will be used to access the address of the clause, in which the next literal of the current decision level is included and, subsequently, the process is repeated (12). However, if there is just one literal in the RLM at the current decision level, then the literals will be stored in the CNF memory as a newly learned clause. In parallel, the RLM is erased such that the RLM is prepared for the next conflict that occurs (13). Next, the backtrack level is identified by checking the decision level of all literals in the learned clause (14). After the backtrack level is determined, all literals with a decision level higher than the backtrack level get unassigned in the CNF memory, i.e., the free literal number of each affected clause is updated and, consequently, removed from the IGM (15). Finally, an implication is made on the learned clause, having in mind that this is an asserting clause (16). Then, the normal solving process continues.

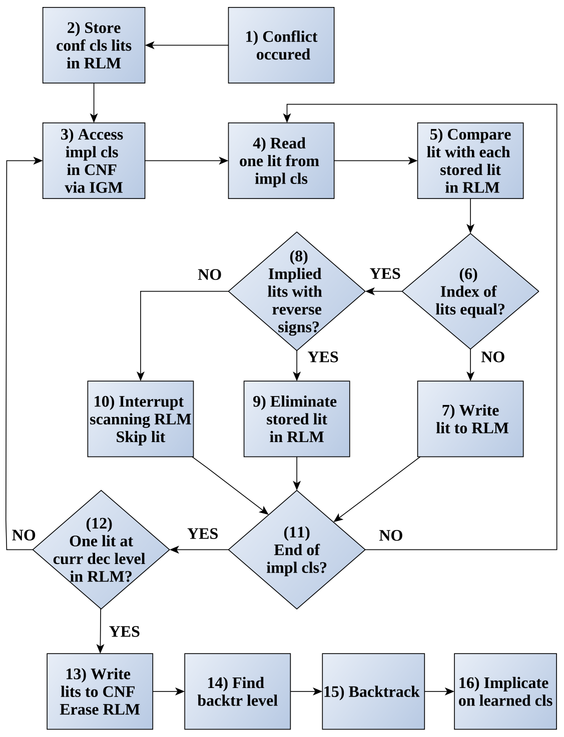

2) Store conf cls lits in RLM
1) Conflict occured
3) Access impl cls in CNF via IGM
4) Read one lit from impl cls
5) Compare lit with each stored lit in RLM
(8) Implied lits with reverse signs?
(6) Index of lits equal?
NO
YES
YES
NO
10) Interrupt scanning RLM Skip lit
9) Eliminate stored lit in RLM
7) Write lit to RLM
(12) One lit at curr dec level in RLM?
(11) End of impl cls?
NO
YES
NO
YES
13) Write lits to CNF Erase RLM
14) Find backtr level
15) Backtrack
16) Implicate on learned cls

Fig. 5. Clause learning procedure

D. Example of Clause Learning in SAT-Hard

In order to demonstrate the newly developed clause learning procedure of this work, Example 2 is reconsidered. The steps during the clause learning are detailed below as well as the linkage to the state numbers (in parenthesis) of the activity diagram shown in Fig. 5:

Example 4: Fig. 6 shows the symbolic representation of the literals and clauses in the utilized memories. If a conflict occurs, the implied literals x_7, x_{10}, x_5 and x_6 with their corresponding clause numbers and decision levels will already be stored in the IGM. The arrows and circled numbers show the steps as follows:

1- The literals $\overline{x}_5$, $\overline{x}_6$ and $\overline{x}_7$ of the unsatisfied clause w_3 are written to the RLM (2).

2- The literal x_6 is implied in w_2, which points to the CPM and the end of clause address 6, is accessed in the CNF (3). The index of the literals x_4 and x_6 in w_2 are compared with the ones in RLM (4). x_6 is eliminated because it is implied literal at the current decision level and its negation is already in the RLM (8). x_4, that is not yet in the RLM, is added to the RLM (7). The literals $\overline{x}_5$, $\overline{x}_6$ and $\overline{x}_7$ form the temporary clause $w_T = (x_4 \vee \overline{x}_5 \vee \overline{x}_7)$. This is due to the fact that 2 literals at current decision level 4 in the RLM (12) exist and the next implied literal is referenced by the IGM (3).

3- The clause w_1 is accessed by IGM due to the location of literal x_5 (3). Similarly, the literals x_1, x_4 and x_5 are compared one by one with the literals of the temporary clause (4). Subsequently, x_5 is removed (8), x_1 is written (7) and x_4 has been written before and, hence, it is skipped at this time (10). This is due to the fact that there is *only one literal* at decision level 4 in the RLM, Consequently, the literals $\overline{x}_7$, x_1, x_4 form the learned clause $w_L = (x_1 \vee x_4 \vee \overline{x}_7)$.

4- The literals of the learned clause w_L are transferred from the RLM to the CNF (13).

TABLE I
RESOURCE UTILIZATION ON ZYNQ

Elements	[3]	SAT-Hard	Increase Rate
Slice LUTS	1479	1894	1.28
Slice Registers	568	765	1.35
Block RAM Tile	79	109	1.38

V. EXPERIMENTAL RESULTS

This section presents the obtained experimental results by considering different benchmark instances. At first, the resource usage is shown, which proves the lightweight characteristic of SAT-Hard. Additionally, the solving run-times of several SAT-instances are presented, which clearly demonstrate the overall performance of SAT-Hard and the outperform against state-of-the-art approaches. Especially, detailed statistics are presented to emphasize the benefit of the introduced learning capabilities.

A. Area Utilization

SAT-Hard has been implemented Verilog HDL. In order to compare the utilized area against [25] and [24], SAT-Hard has been implemented on a Xilinx Virtex-II Pro device. Results indicate that the proposed design utilizes solely 9% of the Slice LUTs. In comparison, [25] occupies 82% of the LUTs and, consequently, is roughly 9 times larger than SAT-Hard. Furthermore, [24] uses 70% of the available LUTs on the same device, and thus, has nearly 8 times higher costs in terms of area compared to SAT-Hard.

The *Xilinx Zedboard Zynq* is used as the evaluation platform for all experiments. This device combines a dual-core ARM Cortex-A9 processor with the programmable logic unit (*xc7z020clg484* FPGA core). All SAT-instances are transferred by the host system (Petalinux) –running on ARM microprocessor– via the *Advanced eXtensible Interface* (AXI) to the FPGA core, which implements SAT-Hard.

SAT-Hard is able to solve instances of up to 16,384 clauses and with the same number of literals in the format of *3-CNF*[1]. The current implementation occupies 3.56% of the Slice LUTs, 0.72% of the Slice Registers and 77.86% of the available Block RAMs.

Table I compares the synthesis results on this FPGA with [3]. The occupied hardware resources are negligible, which clearly proves that the lightweight characteristic of SAT-Hard (over basic HW SAT-solver) remains and, hence, it is well-suited for an application like self-verification, which has to deal with strictly limited hardware resources.

[1]Note that SAT-Hard is not limited to a format that has the same number of literal in every clause.

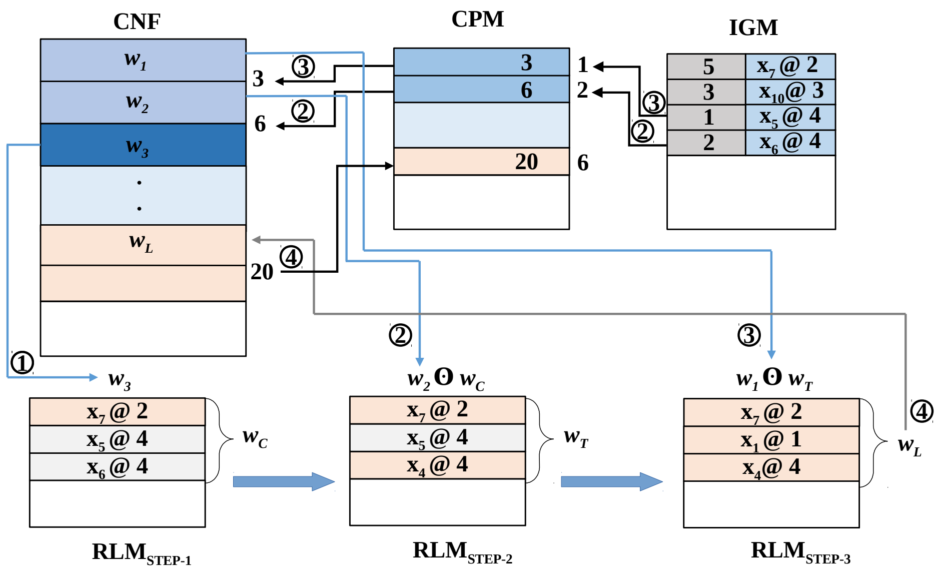


Fig. 6. Exemplary generation of learned clause

B. Run-time Comparison

TABLE II
RESULTS FROM SATLIB

Instance	#Var	#Cls	Status	[3][s]	SAT-Hard[s]	Speed-Up
hole7	56	204	UNSAT	0.61	0.33	1.85
hole8	72	297	UNSAT	7.02	2.27	3.09
hole9	90	415	UNSAT	79.45	15.29	5.20
uf100-10	100	430	SAT	2.80	0.58	4.83
uuf100-2	100	430	SAT	10.81	4.94	2.19
uf125-1	125	538	SAT	5.65	1.16	4.87
uuf125-5	125	538	UNSAT	15.32	4.90	3.13
uf150-8	150	645	SAT	20.08	3.92	5.12
cbs100-1	100	403	SAT	6.71	2.34	2.87
aim-200-3_4	200	680	SAT	timeout	1.20	-
aim-200-1_6	200	320	UNSAT	timeout	0.01	-
ii16e2	532	7825	SAT	6.10	5.76	1.06
ii32e1	222	1186	SAT	0.02	0.02	1.00

TABLE III
RESULTS FROM BMC

Instance	#Var	#Cls	Status	[3][s]	SAT-Hard[s]	Speed-Up
b4-38	2005	4892	SAT	23.66	3.26	7.26
b4-40	1700	4123	SAT	timeout	7.26	-
b4-90	1151	2967	SAT	1692.35	0.89	1901.86
b12-2	9490	14549	SAT	timeout	15.59	-
b12-12	2452	5947	UNSAT	1024.44	3.27	313.28
s38584-69	5201	13088	SAT	2100.37	4.90	428.65

TABLE IV
RESULTS FROM SELF VERIFICATION INSTANCES

Instance	#Var	#Cls	Status	[3][s]	SAT-Hard[s]	Speed-Up
simple	161	539	UNSAT	21.08	1.48	14.24
smart-6	163	335	UNSAT	8.28	0.02	414.00
multip-conf	214	429	UNSAT	14.88	0.06	248.00
simple-2	388	777	UNSAT	100.14	0.07	1430.57
mult8-17	583	1549	UNSAT	34.14	2.85	11.98
mult8-66	532	2392	UNSAT	50.59	6.17	8.20
mult-10-2	267	1015	UNSAT	39.43	5.26	7.50

The considered CNF instances have been clustered into different benchmark sets and been distinguished against [3], which is presented in the tables as follows:

- Table II presents the solving run-times for different classes of application specific instances from Satlib [26].
- Table III considers *Bounded Model Checking* (BMC) instances, which are used in the field of circuit test [27].
- Table IV illustrates the results of the instances that are used in the context of self-verification. These instances are generated by configuring variables leading to a significant reduction of the search space [28].

For all experiments, a *timeout* was assumed if the solver could not succeed to solve an instance within 5 hours.

The performance is highly dependent on the structure of the problem. Notwithstanding, all results show that SAT-Hard successfully solves each and every of the evaluated instances in less than 20 seconds. In contrast to this, [3] is only capable to solve those instances, whose conflicts can be resolved by invoking basic search procedures. Otherwise, [3] fails to solve instances, which require more sophisticated resolve procedure to, e.g., avoid being trapped in the non-solution space.

C. Detailed Solving Statistics

To discuss the advantage of the introduced learning capability in a more extensive way, detailed solving statistics[2] for some SAT-instances are presented in the following.

These numbers have been conducted by recording the total number of decisions, implications, and conflicts during the solving process. More precisely, the improvement factor is determined by dividing each measurement of basic HW SAT-solver by SAT-Hard, i.e., *number_of_decisions([3] / SAT-Hard)*. For example, the number of decisions of the instance *hole9* when using basic HW SAT-solver is 322,559 and when using SAT-Hard is 2,547. Consequently, the improvement factor is 126.6. All improvement factors are shown in Fig. 7 and in Fig. 8 with a logarithmic scale. It can be realized that

[2]Note that the numbers of instances, which led to a timeout while applying [3], could not have been determined and, therefore, these instances are not included in the graphics.

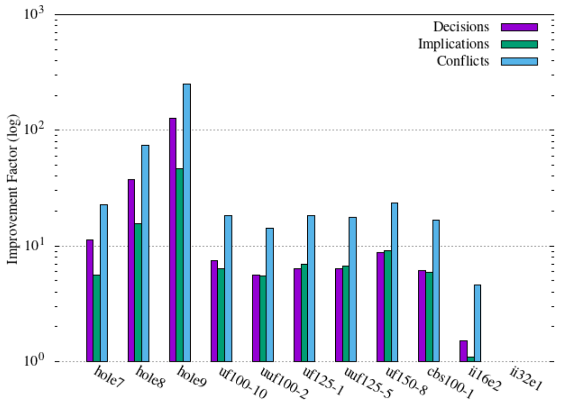


Fig. 7. Statistics of Satlib instances

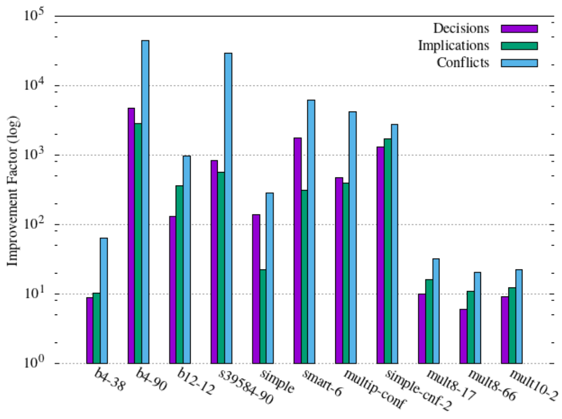


Fig. 8. Statistics of BMC and self-verification instances

SAT-Hard decreases the number of computations by orders of magnitude. Moreover, during the solving of instance *ii32e1*, no conflict occurs and, hence, the same number of decisions as well as implications are measured for SAT-Hard and for [3].

VI. Conclusions

This paper proposed SAT-Hard, a *learning-based* HW SAT-solver, i.e., SAT-Hard enables conflict-driven clause learning (CDCL) completely in HW. To realize CDCL, a sophisticated memory model has been developed and implemented within a lightweight and stand-alone HW SAT-solver. As shown in the experiments, SAT-Hard achieves a speed-up up to 2,000x compared to the existing state-of-the-art approach. Moreover, SAT-Hard is able to solve high complex SAT-instances which are out of reach for the existing HW SAT-solver. Since the lightweight characteristic in terms of area is clearly met, SAT-Hard enables new emerging approaches in the domain of verification and test.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grant no. 01IW16001, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Collaborative Research Center SFB1320 *EASE – Everyday Activity Science and Engineering* in subproject P04, as part of the Collaborative Research Center SFB 1232 – project number 276397488 – in subproject P01 Predictive function, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

References

[1] R. Drechsler, M. Fränzle, and R. Wille, "Envisioning self-verification of electronic systems," in *ReCoSoC*, 2015, pp. 1–6.
[2] R. Drechsler, H. M. Le, and M. Soeken, "Self-verification as the key technology for next generation electronic systems," in *SBCCI*, 2014, pp. 15:1–15:4.
[3] B. Ustaoglu, S. Huhn, D. Große, and R. Drechsler, "SAT-Lancer: a hardware SAT-solver for self-verification," in *GLSVLSI*, 2018, pp. 479–482.
[4] S. A. Cook, "The complexity of theorem proving procedures," in *STOC*, 1971, pp. 151–158.
[5] M. Davis and H. Putnam, "A computing procedure for quantification theory," *JACM*, vol. 7, pp. 506–521, 1960.
[6] M. Davis, G. Logeman, and D. Loveland, "A machine program for theorem proving," *Comm. of the ACM*, vol. 5, pp. 394–397, 1962.
[7] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *TC*, 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, pp. 530–535.
[9] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT*, ser. LNCS, vol. 2919, 2004, pp. 502–518.
[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *DAC*, 2001, pp. 530–535.
[11] J. Marques-Silva and K. Sakallah, "Invited tutorial: Boolean satisfiability algorithms and applications in electronic design automation," in *CAV*, 2000, pp. 3–3.
[12] I. Skliarova and A. de Brito Ferrari, "Reconfigurable hardware SAT solvers: a survey of systems," *TC*, vol. 53, no. 11, pp. 1449–1461, 2004.
[13] Z. Chen, J. Wu, H. Guo, J. Xiong, and A. He, "A FPGA based SAT solver with random variable selection," in *ICICM*, 2016, pp. 329–333.
[14] A. He, L. Yu, H. Zhang, L. Li, and J. Wu, "A fpga based sat solver with high random and concurrent strategies," in *QRS-C*, 2018, pp. 221–228.
[15] K. Bousmar, "A pure hardware k-SAT solver for FPGA," in *CIST*, 2018, pp. 481–485.
[16] K. Bousmar, F. Monteiro, Z. Habbas, S. Dellagi, and A. Dandache, "A pure hardware k-SAT solver architecture for FPGA based on generic tree-search," in *ICM*, 2017, pp. 1–5.
[17] M. Kefan, X. Liquan, Z. Jianmin, and L. Tiejun, "An FPGA SAT solver based on enhanced constraint," in *FPGA4GPC*, 2017, pp. 25–30.
[18] K. Kanazawa and T. Maruyama, "An approach for solving SAT/MaxSAT-encoded formal verification problems on FPGA," *IEICE*, vol. E100.D, no. 8, pp. 1807–1818, 2017.
[19] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *AAAI*, 1994, pp. 337–343.
[20] T. Ivan and E. M. Aboulhamid, "An efficient hardware implementation of a SAT problem solver on FPGA," in *DSD*, 2013, pp. 209–216.
[21] P. Zhong, M. Martonosi, and P. Ashar, "FPGA-based SAT solver architecture with near-zero synthesis and layout overhead," *CDT*, vol. 147, no. 3, pp. 135–141, 2000.
[22] M. Waghmode, K. Gulati, S. P. Khatri, and W. Shi, "Efficient, scalable hardware engine for boolean satisfiability," in *ICCD*, 2006, pp. 326–331.
[23] K. Gulati, M. Waghmode, S. P. Khatri, and W. Shi, "Efficient, scalable hardware engine for boolean satisfiability and unsatisfiable core extraction," in *IET*, vol. 2, no. 3, 2008, pp. 214–229.
[24] K. Gulati, S. Paul, S. P. Khatri, S. Patil, and A. Jas, "FPGA-based hardware acceleration for boolean satisfiability," in *TODAES*, vol. 14, no. 2, Apr. 2009, pp. 33:1–33:11.
[25] M. Safar, M. W. El-Kharashi, M. Shalan, and A. Salem, "A reconfigurable, pipelined, conflict directed jumping search SAT solver," in *DATE*, 2011, pp. 1–6.
[26] "SATLIB - benchmark problems," https://www.cs.ubc.ca/ hoos/SATLIB/.
[27] G. Fey, A. Sulflow, S. Frehse, and R. Drechsler, "Effective robustness analysis using bounded model checking techniques," in *TCAD*, vol. 30, no. 8, 2011, pp. 1239–1252.
[28] M. Ring, F. Bornebusch, C. Lüth, R. Wille, and R. Drechsler, "Better late than never: Verification of embedded systems after deployment," in *DATE*, 2019.