SAT-Lancer: A Hardware SAT-Solver for Self-Verification

Buse Ustaoglu, Sebastian Huhn, Daniel Große, Rolf Drechsler
Cyber Physical Systems, DFKI GmbH, Bremen, Germany
Group of Computer Architecture, University of Bremen, Germany
buse.ustaoglu@dfki.de,{huhn,grosse,drechsle}@cs.uni-bremen.de

ABSTRACT

To close the ever widening verification gap, new powerful solutions are strictly required. One such promising approach aims in continuing verification tasks after production of a chip during its lifetime. This approach is called self-verification. However, for realizing self-verification tasks on-chip, verification packages have to be developed. In this paper, we propose the verification package SAT-Lancer. SAT-Lancer is a compact Boolean Satisfiability (SAT) solver and has been implemented entirely on HW with the capability of solving any arbitrary SAT-instance. At the heart of SAT-Lancer is a scalable memory model, which can be adjusted to given memory constraints and allows to store the SAT-instance most effectively. In comparison to previous HW SAT-solvers, SAT-Lancer utilizes significant less area and can handle order of magnitude larger SAT-instances.

ACM Reference Format:
Buse Ustaoglu, Sebastian Huhn, Daniel Große, Rolf Drechsler. 2018. SAT-Lancer: A Hardware SAT-Solver for Self-Verification. In *Proceedings of 2018 Great Lakes Symposium on VLSI (GLSVLSI '18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194554.3194643

1 INTRODUCTION

Electronic systems are used in various application domains ranging from cars to mobiles, from industrial control to IoT devices. In these domains very different design goals with respect to functionality, performance and power requirement have to be considered. However, the process of verification is fundamental for all electronic systems and it ensures that a design implementation meets its specification. To keep pace with the enormous dimension of freedom in producing chips, a variety of verification methods are available today. Essentially, three main directions can be distinguished: simulation, formal verification and emulation. A strong verification platform spanning from abstract high-level models down to the *Register-Transfer-Level* (RTL) implementation is formed by orchestrating these techniques in a clever way. Simultaneously, on the design side, *Intellectual Property* (IP) re-use has become an integral part of all design flows today. Unfortunately, despite all these advancements, the verification complexity continues to increase at approx. 4 times more than the rate of design creation [11]. To attack this widening verification gap, the new promising and complementary approach of *self-verification* has been recently proposed in [5, 6, 10]. Self-verification allows an electronic system to continue open verification tasks after fabrication. However, to bring self-verification to life, as major components verification packages are needed. The verification packages execute the remaining verification tasks directly on the chip.

In this paper, we target small to mid-sized electronic systems (e.g. from the IoT domain), and hence present a compact verification package for self-verification, which allows to heavily orchestrate SAT-based verification techniques like *Bounded Model Checking* (BMC) [1] directly on-chip. More precisely, we propose **SAT-Lancer: a complete HW SAT-solver IP-component** which can solve any arbitrary SAT-instance that has been generated on-chip. Such a SAT-instance represents a Boolean formula (coming from a verification task) and raises the question whether an assignment to all Boolean variables exists such that the overall formula is *satisfied* (SAT) or remains *unsatisfied* (UNSAT), respectively. Other existing approaches for SAT-solving either focus on accelerating a SW-based SAT-solver by outsourcing the solving process partially to HW [2, 8, 13, 14] or by introducing even for small SAT-instance sizes a large HW-overhead [9, 12]. In contrast to this, the proposed HW SAT-solver is very compact and can be easily integrated as an IP-component into an existing design. At the heart of our HW SAT-solver is a *scalable memory model*. Essentially, the memory model defines how to utilize the memory most effectively for storing the investigated SAT-instance as well as meta information of the SAT-solving process itself.

In an extensive experimental evaluation we demonstrate the advantages of our proposed HW SAT-solver. It shows that SAT-Lancer can handle much larger instances in comparison to existing HW-based approaches and outperform a SW-based SAT-solver, which adheres to area constraints for the underlying HW.

2 SCALABLE MEMORY MODEL

In this section we present the heart of SAT-Lancer, i.e., the scalable memory model. Please note that we assume that the reader is familiar with SAT-solving and the basic DPLL algorithm.

In general, to realize a HW SAT-solver the *Conjunctive Normal Form* (CNF) instance has to be stored in a HW memory. This memory is then accessed by the core of the HW SAT-solver for solving the instance, i.e., to determine a satisfying assignment for the clauses or to prove that no such assignment exists.

The basic idea of our proposed memory model is to store a literal of the current clause as well as meta information about the SAT-solving process itself in a memory word. Thereby, scalability is achieved by identifying, which information is essential and which information can be reduced to adhere given memory constraints (e.g. in terms of word size). Moreover, by following the principle of locality, the actual HW implementation of SAT-Lancer and in particular the interface between the SAT-solver core and the memory can be optimized. This allows to process even large SAT-instances.

2.1 Principle Memory Layout

he principle memory layout is shown in the middle of Fig. 1. As can be seen it is inspired by the DIMACs encoding. More precisely,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GLSVLSI '18, May 23–25, 2018, Chicago, IL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5724-1/18/05…$15.00
https://doi.org/10.1145/3194554.3194643

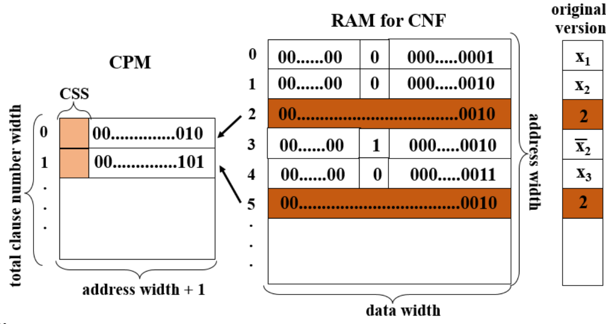

Figure 1: Example of principle memory layout for example CNF $f = (x_1 + x_2) \cdot (\overline{x}_2 + x_3)$

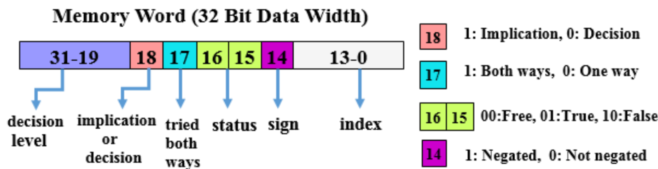

Figure 2: Meta Information and Literal Encoding

we utilize the lower bits of each memory word to store the index and add one further bit to store the sign. When assuming a 32 bit memory, obviously, many bits remain unused and, consequently, should be used to save HW resources. Due to this fact, we identify which information of the SAT-solving process can be stored in the upper bits of a memory word to effectively use the memory.

On the left of Fig. 1 an additional memory is shown which stores the address where a clause ends. We refer to this memory as *Clause Position Memory* (CPM). The CPM allows a fast and direct access to a specific clause. With this knowledge, instead of storing a "0" to mark the end of a clause, we can now use this memory location to store the number of free literals of the clause under the current assignment. Moreover CPM holds the decision level of a literal that has maximum value of the clause.

Finally, the most-significant bit of each CPM entry represents the *Clause SAT Status* (CSS), i.e., whether the current clause is already SAT or not (cf. Subsection 2.2).

2.2 Encoding of Meta Information

Staying with the assumption of a 32 bit memory, we performed a careful analysis of information to be stored in a memory word supplementary to the pure literal encoding. Fig. 2 shows the outcome: For a 32 bit memory word, the lower 14 bits encode the variable index, the 14th bit the sign, bit-15,16 define the status of the literal which can be free, assigned to "0" or assigned to "1", bit-17 defines whether the variable has been assigned to both values ("0" and "1") for conflict resolution at same decision level, bit-18 indicates whether the assignment to the current literal results from a decision of the SAT-solver or an implication and the upper 13 bits (19 to 31) represent the decision level, i.e., the number of decisions, which have so far been done during the solving process.

Based on the general presentation of our memory model, we want to emphasize that the scalability is a major challenge: This is due to the application-specific resource constraints, particularly, constraints concerning the memory size as well as memory width, i.e., the size of a (single) word. As becoming evident from the introduced SAT-instance encoding, our scheme is scalable: the bit

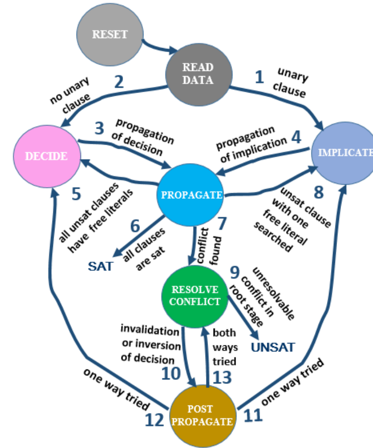

Figure 3: State Machine

width of a memory word can be varied and we can reflect this by adapting the number of bits to represent the index as well as the decision level while the other bits are kept (essential information).

3 DESIGN OF SAT-LANCER

This section describes the principle design of SAT-Lancer from a HW point of view and presents an exemplary application of the complete SAT-solving procedure on basis of a comprehensible SAT-instance.

3.1 Main Finite State Machine

In Fig. 3, the *Finite State Machine* (FSM) is represented, which is inspired by the well-known *DPLL*-algorithm [3, 4]. In the following we describe each state in detail by assuming an arbitrary CNF is represented according to the introduced memory model of Section 2.

Reset: This state initializes the complete SAT-solver and, thus, introduces a well-defined state to SAT-Lancer.

Read data: The complete SAT-instance is transferred to the device implementing SAT-Lancer and stored in the main memory. After the transfer is completed, the number of literals in every clause is determined and stored in the memory location (cf. Fig. 2). In the case that the processed CNF contains any unary clause, i.e., a clause hold just one positive or negative literal, the next state to continue is set to "implicate" (cf. Edge 1), and to "decide" (cf. Edge 2) otherwise.

Decide: In this state an assignment to a variable, which has not already been assigned, is determined. A particular clause which has not been yet satisfied and has free variables is identified by the relevant states. If such a literal is determined, an assignment to "0" or "1" to the corresponding variable is derived such that the literal evaluates to true. Furthermore, information concerning the index and the determined decision (assignment) is stored in a dedicated register and the decision level is increased by one. The determined assignment is then propagated to all clauses by entering the "propagate" state (cf. Edge 3).

Implicate: Here, all clauses are further investigated, which are not yet satisfied and which hold just one single unassigned literal. In particular, this literal has to be necessarily assigned in the way that it evaluates to true. These clauses can be easily detected by taking advantage of the collected information concerning the CSS and the remaining number of free variables. If such a variable is identified, this variable is introduced as an implication (set implication status

bit) and stored in a separate register. Analogously, the assignment is propagated by entering the "propagate" state (cf. Edge 4).

Propagate: In this state, every (positive as well as negative) literal of the complete SAT-instance holding the specific Boolean variable, which should be assigned, is identified by taking advantage of the determined index information. If a clause is already satisfied, the clause is skipped. Otherwise, for the case that the current assignment satisfies the clause, the SAT counter is increased and the decided bit for the CSS (cf. Fig. 2) is set. Furthermore, the number of free variables of this clause is decreased by one. During the evaluation procedure, the implication address is stored and the implication flag is set for all clauses which are not yet satisfied and which have just one remaining unassigned literal. If the SAT counter equals the total number of clauses, the overall instance becomes SAT (cf. Edge 6). If a clause, which is not yet satisfied, has no more free variables, i.e., every literal evaluates to false under the current assignment, a conflict occurs. This causes an immediate state transition to "resolve conflict" (cf. Edge 7). Otherwise, the state is changed to "implicate" (cf. Edge 8). If there is no conflict and the implication flag is set, the state is changed to "implicate" (cf. Edge 8). Otherwise the state returns to "decide" to assign a further literal (cf. Edge 5).

Resolve conflict: To resolve the conflict, the decision level is decreased by one, i.e., the last assignment to the decision variable is withdrawn and all derived implications are invalidated which are marked by the *implication status bit*. Hence, all previously affected clauses are reevaluated. This step also includes that the counter of free variables is updated for the effected clauses as well unsetting the implication status bit if necessary (cf. Edge 10). In case the conflict occurred at decision level 0 (root level) and it is not possible to flip any variable since all have been already tried in both ways ("0" or "1"), the SAT-instance is UNSAT (cf. Edge 9).

Post propagate: Mainly, this state is invoked by the conflict resolution to check whether a decision variable has already been tried in both ways, i.e., by an assignment with "0" as well as "1", to resolve the conflict. If this is not the case, this not yet tried assignment will be propagated towards "implication" or "decision" state based on the free variable numbers which is same procedure as in "propagate" state (cf. Edges 11 and 12). Otherwise, the status bit *tried both way* of the investigated variable is set and the procedure continues in state "resolve conflict" to further decrease the decision level (cf. Edge 13).

3.2 Example of Solving Process

To demonstrate the solving process of SAT-Lancer we use the following example.

Example 1. *Given the SAT-instance*

$$g = (\overline{x}_1 + x_4 + x_5) \cdot (x_1 + \overline{x}_2 + x_3) \cdot (x_1 + \overline{x}_2 + \overline{x}_3) \cdot (x_1 + x_2 + \overline{x}_3) \cdot (x_1 + x_2 + x_3)$$

As it can be seen, g consists of 5 *clauses,* 5 *variables and* 15 *literals. Table 1 lists the steps for solving g with SAT-Lancer. The first column gives the number of the current step. The second column shows the current state of SAT-Lancer (the used abbreviations can be found below the table). The column* Variable Status *shows the assignment to a variable in the form $x_i = a@d$, i.e. x_i is assigned to a in decision level d; note that during backtracking a variable may be unassigned and the decision level is reset to* 0 *and we denote this as $x_i = U@0$. The next column gives the status of the clause after the assignment,*

Table 1: Example of SAT-Lancer's solving process

Step	State	Variable Status	Clause Status	Free Literals	SC
1	D	$x_1 = 0@1$	(T, F, F, F, F)	[-, 2, 2, 2, 2]	1
2	D	$x_2 = 0@2$	(T, T, T, F, F)	[-, -, -, 1, 1]	3
	I	$x_3 = 0@2$	(T, T, T, T, C)	[-, -, -, -, 0]	4
3	R	$x_3 = U@0$	(T, T, T, F, F)	[-, -, -, 1, 1]	3
4	PP	$x_2 = 1@2$	(T, F, F, T, T)	[-, 1, 1, -, -]	3
	I	$x_3 = 1@2$	(T, T, C, T, T)	[-, -, 0, -, -]	4
5	R	$x_3 = U@0$	(T, F, F, T, T)	[-, 1, 1, -, -]	3
	PP	$x_2 = U@0$	(T, F, F, F, F)	[-, 2, 2, 2, 2]	1
6	PP	$x_1 = 1@1$	(F, T, T, T, T)	[2, -, -, -, -]	4
7	D	$x_4 = 1@3$	(T, T, T, T, T)	[-, -, -, -, -]	5

State: **D**ecide, **I**mplicate, **R**esolve Conflict, **P**ost **P**ropagate, **P**ropagate
Variable Status: $x_i = a@d$ means x_i is assigned to a in decision level d. If $a = U$, x_i is unassigned (decision level is reset to 0)
Clause Status: **T**rue, **F**alse or **C**onflict in the respective clause
Free Literals: - means not relevant since the clause is satisfied, otherwise i in the respective clause
SC: **S**AT **C**ount, i.e., total number of satisfied clauses

i.e., if the clause became True, False *or* Conflicting *(for instance, the assignment $x_1 = 0@1$ makes the first clause $(\overline{x}_1 + x_4 + x_5)$ true and, hence, the first entry in the vector (T, F, F, F, F) becomes T. The column* Free literals *reports the number of free literals in the clauses (using the same assignment $x_1 = 0@1$, the first clause becomes true, so we mark this one with '-' and for all the other clauses the first literal evaluates to false, so in each of them only 2 free literals are left and, finally, leading to $[-, 2, 2, 2, 2]$). The last column* SC *shows the SAT count, i.e., the total number of satisfied clauses. We now describe the solving process for g:*

- Step 1 *SAT-Lancer starts by assigning* 0 *to x_1 at decision level* 1 *as described above when introducing the notations.*
- Step 2 *With the assignment* 0 *to x_2 Clauses 2 and 3 become true while the Clauses 4 and 5 still remain false but have only one free literal left. As a consequence an implication is performed on Clause 4 by assigning x_3* 0*, which leads to a conflict on Clause 5 because of the assignments $x_1 = 0$, $x_2 = 0$ and $x_3 = 0$ and, hence, this clause is false.*
- Step 3 *Due to this conflict, we enter the* Resolve Conflict *state and, hence, x_3 is unassigned and the number of free literals for Clause 4 increases again to 2.*
- Step 4 *Now the decision value for x_2 is converted to the inverted value which makes last two clauses true. This leads to an implication for Clause 2 but also to a conflict occurring in Clause 3.*
- Step 5 *Since x_2 has been already tried in both ways, we need to backtrack to x_1. Hence, the implication on x_3 and then the decision on x_2 are undone. By this the number of free variables are increased.*
- Step 6 *By flipping the assignment to x_1 from* 0 *to* 1*, Clause 1 becomes false and the others are true.*
- Step 7 *After the decision on x_4 to* 1*, all clauses are true and, thus, SAT-Lancer returns SAT.*

4 EXPERIMENTAL RESULTS

This section describes the setup used as an implementation basis. Furthermore we present and discuss the conducted experimental results out of several benchmark runs.

SAT-Lancer has been written in Verilog. At first, the claimed compact character of SAT-Lancer is investigated by comparing the occupied HW-resources of our design against the implementation of [12]. Consequently, we implement our design on a *Xilinx Virtex-II Pro* FPGA as used in [12]. Our design implementation just occupies

Table 2: Results from SATLIB

Instance	#Var	#Cls	Status	Leon3[s]	SAT-Lancer [s]
hole6	42	133	UNSAT	5.98	0.06
hole7	56	204	UNSAT	200.30	0.61
hole8	72	297	UNSAT	739.21	7.02
uuf75-1	75	325	UNSAT	0.54	0.34
uuf75-2	75	325	UNSAT	0.55	0.16
uf100-7	100	430	SAT	0.38	0.03
uf100-10	100	430	SAT	0.59	2.80
cbs100-1	100	403	SAT	0.66	6.71
cbs100-10	100	403	SAT	0.55	0.07
ii8e2	870	6121	SAT	0.44	0.25
ii8b4	1068	8214	SAT	0.47	0.40
ii16e2	532	7825	SAT	26.87	6.10
ii32e1	222	1186	SAT	0.39	0.02

Table 3: Results from verification instances

Instance	#Var	#Cls	Status	Leon3[s]	SAT-Lancer [s]
alu8_and	345	839	UNSAT	0.01	0.42
alu8_or	349	851	UNSAT	0.01	0.34
alu8_orf	349	847	SAT	0.01	0.02
fifo_full	2120	4762	UNSAT	0.05	0.04
b3-26	1850	5236	SAT	0.04	0.47
b12-53	2425	14046	UNSAT	0.04	0.01
s38584-90	5999	15402	UNSAT	0.04	1.34

4% of the Slice LUTs. In contrast to this, [12] requires 20 times more resources compared to SAT-Lancer, i.e., it occupies 80% of the overall Slice LUTs. These results clearly show that the proposed design is suitable for environments with limited HW-resources.

In the actual setup, the whole design of SAT-Lancer has been synthesized on a *Xilinx ZedBoard Zynq* with an embedded *xc7z020clg484* FPGA core. This device consists of two components: a dual-core ARM Cortex-A9 microprocessor as well as a Programmable Logic unit - the FPGA as above mentioned. The SAT-instances are emitted by the host system (PetaLinux) running on the ARM microprocessor, which offers a suitable interface to the FPGA. More precisely, the state-of-the-art *Advanced eXtensible Interface* (AXI) is utilized in connection with some self-written middleware to realize all data communications. The current implementation of SAT-Lancer allows to process instances, which hold up to 65536 clauses and up to 32768 variables in the well-established *3-CNF* format, while occupying 1430 Slice LUTs and 136 BRAM Tiles. In fact, these boundaries can be even enhanced by increasing defined implementation parameters, however, further memory has to be allocated. The chosen parameters lead to an occupation of 2.69% Slice LUTs and 96.79% BRAM Tiles of the overall FPGA's resources, respectively.

To evaluate the actual SAT-solving capability of SAT-Lancer, different benchmark sets are considered and the results are shown in Table 2. In particular, the selected SAT-instances represent a wide variety of different problem classes including Pigeon Hole *(hole)*, Uniform Random-3-SAT *(uu)*, Random-3-SAT with Controlled Backbone Size *(cbs)* and Inductive Interference *(ii)*. Besides this, more application-specific SAT-instances (with respect to the intended domain of self-verification) are also investigated and shown in Table 3. These SAT-instances cover the field of sequential circuit problems containing a relevant high number of variables (clauses) up to approx. 6500 (18000). In particular, the unrolling of sequential circuits is covered, i.e., to determine a certain criteria spread over two (or more) time frames. In fact, unrolling is one elementary step in the most of the verification techniques.

To ensure the correctness of the result, all runs have been validated by the software-based SAT-solver *MiniSAT* [7] off-chip on a separate workstation. Besides this, the well-known microprocessor *LEON3* has also been implemented on the FPGA core of the ZedBoard for comparison. This processor executes a cross-compiled version of MiniSAT[1] to determine the resulting solving-time. In fact, this exhaustive microprocessor implementation allocates 7 times more HW-area in terms of Slice LUTs (SAT-Lancer with 2.69% and LEON3 with 16.93%) than the proposed SAT-Lancer and, furthermore, it takes benefit of the well-engineered MiniSAT solver.

Tables 2 and 3 include the instance name (1st col.) for the individual benchmark as sketched above, the total number of variables and clauses (2nd & 3rd col.) and, finally, the result in sense of {SAT,UNSAT} (4th col.) and the solving-time (5th col.) for LEON3 as well as for SAT-Lancer (6th col.) in seconds. It can be noted that the arbitrary SAT-instances of Table 2 are much harder to solve compared to the application-specific ones in Table 3. SAT-Lancer mostly finishes within seconds which is, in fact, negligible from a self-verification point of view.

5 CONCLUSIONS

This paper proposes SAT-Lancer, a complete HW SAT-solver which allows to solve arbitrary SAT-instances on-chip while occupying only moderate hardware resources. Furthermore, it utilizes an scalable memory model such that even on-chip generated SAT-instances can be processed in a dynamic fashion. Moreover, we have shown that SAT-Lancer is capable to process arbitrary benchmark in reasonable run-time. In summary, SAT-Lancer facilitates completely new application scenarios for functional self-verification, particularly, in systems with strictly limited resources like e.g. the rapidly spreading IoT devices.

Acknowledgments: This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grant no. 01IW16001, by the German Research Foundation (DFG) as part of Collaborative Research Center SFB1320 *EASE – Everyday Activity Science and Engineering* in subproject P04 and by the Collaborative Research Center SFB1232 *High Throughput Exploration for Evolutionary Structural Materials* in subproject P01 *Predictive function*.

REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. 1999. Symbolic model checking without BDDs. In *TACAS*. 193–207.
[2] John D. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. 2008. A Practical Reconfigurable Hardware Accelerator for Boolean Satisfiability Solvers. In *DAC*. 780–785.
[3] M. Davis, G. Logeman, and D. Loveland. 1962. A Machine Program for Theorem Proving. *Comm. of the ACM* 5 (1962), 394–397.
[4] M. Davis and H. Putnam. 1960. A computing procedure for quantification theory. *J. ACM* 7 (1960), 506–521.
[5] R. Drechsler, M. Fränzle, and R. Wille. 2015. Envisioning self-verification of electronic systems. In *ReCoSoC*.
[6] R. Drechsler, H. M. Le, and M. Soeken. 2014. Self-verification as the key technology for next generation electronic systems. In *SBCCI*. 1–4.
[7] N. Eén and N. Sörensson. 2004. An extensible SAT solver. In *Theory and Applications of Satisfiability Testing (LNCS)*, Vol. 2919. 502–518.
[8] K. Gulati, M. Waghmode, S. P. Khatri, and W. Shi. 2008. Efficient, scalable hardware engine for Boolean satisfiability and unsatisfiable core extraction. *IET Computers Digital Techniques* 2, 3 (2008), 214–229.
[9] T. Ivan and E. M. Aboulhamid. 2013. An Efficient Hardware Implementation of a SAT Problem Solver on FPGA. In *DSD*. 209–216.
[10] Christoph Lüth, Martin Ring, and Rolf Drechsler. 2017. Towards a Methodology for Self-Verification. In *ICRITO*.
[11] Walden C. Rhines. 2016. Design Verification Challenges: Past, Present and Future. In *DVCon US Keynote Address*.
[12] M. Safar, M. W. El-Kharashi, M. Shalan, and A. Salem. 2011. A reconfigurable, pipelined, conflict directed jumping search SAT solver. In *DATE*. 1–6.
[13] J. Thong and N. Nicolici. 2013. FPGA acceleration of enhanced boolean constraint propagation for SAT solvers. In *ICCAD*. 234–241.
[14] J. Thong and N. Nicolici. 2015. SAT solving using FPGA-based heterogeneous computing. In *ICCAD*. 232–239.

[1]Please note that the learning capability of MiniSAT has been disabled for a fair setup.