A Circuit-Based SAT Solver for Logic Synthesis

He-Teng Zhang Jie-Hong R. Jiang
Department of Electrical Engineering
National Taiwan University, Taiwan
superpi152@gmail.com jhjiang@ntu.edu.tw

Alan Mishchenko
Department of EECS
University of California, Berkeley
alanmi@berkeley.edu

***Abstract*—In recent years SAT solving has been widely used to implement various circuit transformations in logic synthesis. However, off-the-shelf CNF-based SAT solvers often have suboptimal performance on these challenging optimization problems.**

This paper describes an application-specific circuit-based SAT solver for logic synthesis. The solver is based on Glucose, a state-of-the-art CNF-based solver and adds a number of novel features, which make it run faster on multiple incremental SAT problems arising in redundancy removal and logic restructuring among others. In particular, the circuit structure of the problem instance is leveraged in a new way to guide variable decisions and to converge to a solution faster for both satisfiable and unsatisfiable instances. Experimental results indicate that the proposed solver leads to a 2-4x speedup, compared to the original Glucose.

I. Introduction

Boolean satisfiability (SAT) solving is a key component of modern logic synthesis and verification tools. In verification, SAT is used to prove combinational and sequential properties in the design, such as functional equivalence of outputs or some conditions, which should always hold. The use of SAT in synthesis is less common and is gradually becoming mainstream when SAT solvers replace less scalable computation engines, in particular, binary decision diagrams (BDDs), for the task of proving correctness of circuit transformations, such as removing redundancies, merging of equivalent nodes, and performing functional decomposition. [1].

In most cases, the SAT solver used is an off-the-shelf solver, such as MiniSAT [2] or Glucose [3]. These solvers perform well on large isolated problem instances, as witnessed by the fact that they won SAT solver competitions. However, when it comes to logic synthesis, a strong single-instance SAT solver is not needed. A flexible, robust, light-weight solver performs better on a sequence of incremental circuit-based problems generated when checking properties in logic synthesis, such as detecting redundancies, validating structural choices in technology mapping, or proving the existence of a decomposition, etc.

The use of application-specific SAT solving has a long history. The early work on efficient implementation of automatic test-pattern generation (ATPG) [4] coupled with the progress in conflict-driven SAT solving [5][6] developed in the context of CNF-based SAT solving, led to the development of circuit-based SAT solvers [7][8][9]. In the last two decades, the technology for application-specific SAT solving has matured, resulting in numerous improvements, such as [10], as well as hybrid solvers, such as [11][12] focusing on cryptography.

The existing circuit-based solvers have several limitations:

- They often leave out some practical features of CNF-based solvers because these cannot be readily transferred to work on the circuit.
- They rely on circuit-based J-frontier, which is often at odds with variable activity-based decision heuristics such as VSIDS used in CNF-based solvers [6].
- They are often stand-alone and require special effort to integrate with the circuit representation used in a logic synthesis application.

The SAT solver of this paper addresses the above limitations and differs from state-of-the-art circuit-based solvers as follows:

- It is based on the award-winning CNF-based SAT solver Glucose [3], thus leveraging the efficient infrastructure for constraint propagation and conflict analysis along with other features.
- It uses a known data-structure called J-frontier while offering new ways of making it work with activity-based variable decisions.
- It has novel APIs for sharing the circuit structure with a logic synthesis application. (Appendix of [13])

The proposed solver is developed in the context of SAT sweeping, which is a scalable way of detecting functionally equivalent nodes in a combinational logic circuit. To understand the role of SAT solving in this context, the reader is referred to [14].

The rest of the present paper is organized as follows. Section II introduces the background on Boolean satisfiability and and-inverter graphs used to represent circuits in circuit-based SAT solvers. Section III presents the main contributions of the paper, which is the design of a circuit-based solver for logic synthesis. Section IV presents experimental results while Section V concludes the paper.

II. Preliminaries

A. Boolean Satisfiability

A decision problem, which seeks an assignment such that a given Boolean formula $\mathcal{F}$ is evaluated to 1, is a *Boolean satisfiability* (SAT) problem. $\mathcal{F}$ is *satisfiable* if such assignment exists. Otherwise, the problem is *unsatisfiable* and equivalent to constant 0. In practice, the instance of $\mathcal{F}$ is often represented in *conjunctive normal form (CNF)* [15]. A CNF expression is a conjunction of clauses where each clause is a disjunction of *literals*. Literals are polarities (positive or negative) of Boolean *variables* used to formulate $\mathcal{F}$.

B. SAT Solving Framework

A program that solves SAT problems is called a *SAT solver*. Modern SAT solvers often utilize *conflict-driven clause learning* (CDCL) [5][16]. A SAT solver assigns 0 or 1 to variables by making *decisions*, as a mean of satisfiability *reasoning*. Activity-based decision heuristic is a robust strategy widely used in modern SAT solvers [6][2][3]. A necessary assignment deduced by reasoning is an *implication*. Consecutive implications result in constraint *propagation*. A decision-propagation cycle is a *decision level*. During propagation, a *conflict* means a variable is implied to be 0 and 1 simultaneously. When a conflict occurs, *conflict analysis* examines implication dependencies on *implication graph* [5], and characterizes the root cause of conflict as a *learnt clause*. In order to avoid the same conflict in later reasoning, the learnt clause is added to CNF without altering the satisfiability of original formula. After a conflict clause is learnt, solver cancels some decisions, *backtracks* to a previous decision level, and resumes the search.

This research was supported in part by MOST 109-2224-E-002-008 of the Ministry of Science and Technology of Taiwan and by SRC Contract 2867.001.

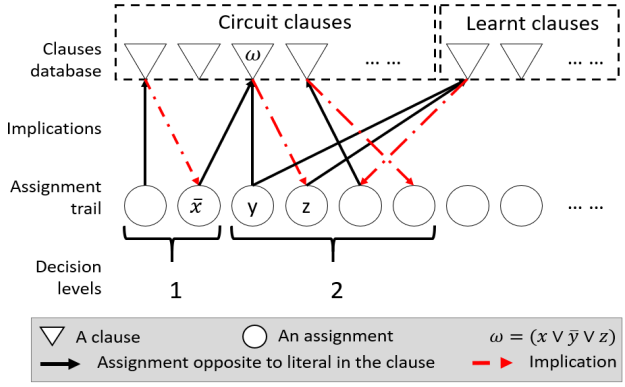

Fig. 1. The state of a SAT solver, including clause sets, assignments, decision level and implication graph, used in Example 1

Example 1. *We use directed bipartite graph for a compact representation of solver state. As shown in Figure 1, the variable z is implied to be 1 by the clause ω under the assignment $(x, y) = (0, 1)$. Note that, an assignment made at the beginning of each level is a decision not pointed by any red arrow, while all other followup assignments in the same level are implied by propagation. The red-black arrows in the graph form an implication graph. When a conflict occurs, the conflict analysis traces implications on the red-black arrows and adding to a new learnt clause the assignments characterizing the conflict. Next, backtracking cancels the assignments made later than the highest decision level of the literals (except for the conflicting literal) involved in the learnt clause.*

Incremental solving mode is available in SAT solvers [2][3]. It allows multiple calls to a solver with assumptions on variables, without resetting the solver. Between the calls, the CNF loaded into the solver can be reused by supplementing it with new clauses. Most importantly, the learning, which happened, helps improve performance of the solver.

C. And-Inverter Graph

Without losing generality, circuits are in the form of *and-inverter graphs* (AIGs) [17]. An AIG is a *directed acyclic graph* (DAG) consisting of *primary input/output* (PI/PO) and two-input and-nodes with optional inverter marks on the fanin edges. *Transitive fanin* (TFI) of a node is a set of nodes reachable by recursively traversing the node's fanins.

D. SAT Solving with Structural Guidance

Structural information, such as circuit connectivity, allows for a legitimate proof of satisfiability with partial assignment. In *circuit-based SAT solving*, input formula $\mathcal{F}$ is represented as a logic circuit without generating CNF. A logic gate is a *J-node* and requires *justification*, if its fanin values do not imply the output value of the gate. The set of J-nodes at any time during solving is called *J-frontier* [4][10]. Decisions made using a J-frontier aim at justifying all J-nodes. When J-frontier becomes empty, i.e., all J-nodes are justified, the solver concludes that the formula $\mathcal{F}$ is satisfiable. In contrast, pure CNF-based solving without circuit information requires a complete assignment on all variables as a witness of the satisfiability of formula $\mathcal{F}$. Figure 2 shows an example demonstrating output 0 is justified recursively.

III. Contributions

The main contribution of this paper is a hybrid SAT solver in the tradition of [7] based on the award-winning CNF solver Glucose [3], which is in turn based on MiniSAT [2]. The reason

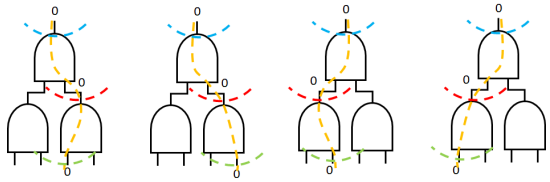

Fig. 2. Four possible configurations justifying the output 0 of the circuit, where colored dash-lines are justification frontiers. The branches without annotating a number can be left as unassigned.

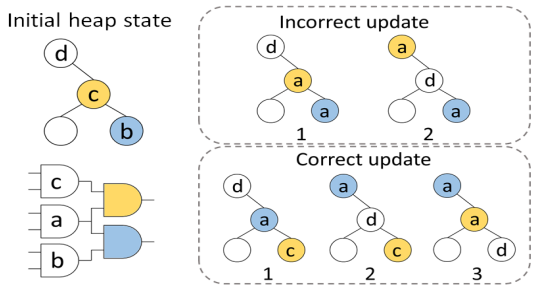

Fig. 3. An example of activity management in Section III-B

for selecting Glucose/MiniSAT as the base, is because they offer a strong implementation of CNF-based features needed to maintain learned clauses in a hybrid solver. In particular, CNF propagation using two-literal watching, conflict-driven clause learning, and conflict clause minimization, are reused in our solver without the need to reimplement them.

On the other hand, the circuit-based features proposed in this paper include the following novel building blocks: (1) efficient justification with activity values, (2) considerations for using precise activity values, (3) efficient non-chronological restoration of J-frontier, (4) seamless adoption of clause minimization techniques to circuit structure, and (5) refining the solving scope for better performance.

A. Activity-Based Justification

In Figure 2, when branching occurs on J-frontiers, the strategy selecting the fanin of the largest activity value is *activity-based justification*. In order to leverage activity-based decision and justification mechanism, a data structure is required to store gates on J-frontier in an efficient way, such that 1) gates with the highest activity can be easily found and 2) newly propagated gates requiring justification can be readily added.

A novel data structure, J-heap, makes it possible to perform efficient activity-based justification. Similar to the heap used in CNF solvers, J-heap stores variables according to the activity values. However, due to the lack of circuit information, the traditional heap requires initialization by adding all unassigned variables to the heap. In contrast, J-heap is more efficient because variables are added to it only when J-frontier explores the circuit by propagation or decision. This property grants J-heap smaller size, compared to the traditional heap, and thus J-heap has better performance when performing push or pop operations.

B. Management of Activity Values

Variable activity values are frequently updated during conflict analysis to ensure the quality of forthcoming decisions. When activity-based justification is used, the management of activity values is different from a CNF solver in two ways: 1) the activity value of a J-node equals the max activity among its fanins, 2) when conflict analysis bumps a variable, the position of its fanout J-nodes

TABLE I
SYMBOLS FOR DETAILED ANALYSIS OF J-WATCH

g, g'	SAT variables or logic gates.
$\texttt{level}(g)$	Decision level of variable g.
$\mathcal{J}(g)$	Fanins forming an irredundant justification of g.
$\mathcal{J}_w(g)$	J-watch of variable g.
l_c	The level conflict and backtracking occur.
l_b	The target backtracking level.

in J-heap may need to be updated more than once. The *max-heap* property of J-heap could be violated if an affected J-node updates its position incorrectly. To decouple the J-heap update problem from the increasing (bumping) or the decreasing (decaying) of variable activity, we collect affected J-nodes during conflict analysis and update the activity of variables in J-heap just before updating the heap.

Example 2. *`Max-heapify` is a typical operation for heap updating. In Figure 3, on the left are a circuit and sub-tree of the associated J-heap, where initial activities hold $a < b, c, d$. On the right are possible outcomes for the updated activities: $b, c, d < a$. In the incorrect case, the validity of max-heapify becomes order-dependent, for all activities J-heap referred to were updated at once. The blue node is blocked by the yellow and stays below node d. In contrast, the correct case updates the referred activity right before adjustment of a node.*

C. Non-Chronological Restoration of J-frontier

Non-chronological backtracking allows SAT solving to resume at a previous decision level where conflict originates. However, SAT solver still restores its state chronologically by pushing all cancelled variables back to the heap. In contrast, with the circuit information used in the solver, we introduce J-watch mechanism to realize non-chronological restoration of solver state during backtracking. It associates each variable with a watch-list recording justification dependency with two operations defined: (Symbols in Table I are used in the following context.)

Definition 1. ***J-watch insertion*** *if $g \in \mathcal{J}(g')$ holds, then g' is added into $\mathcal{J}_w(g)$.*

Definition 2. ***J-watch deletion*** *if g is cancelled during backtracking, then check and clear members of $\mathcal{J}_w(g)$. For any $g' \in \mathcal{J}_w(g)$, push g' back to J-heap if $\texttt{level}(g') \leq l_b$ holds.*

Backtracking in a typical SAT solver pushes N cancelled variables back to heap of size K in time

$$O(N \log K). \tag{1}$$

This could be quite costly, for N equals the area of a sub-circuit consisting of cancelled decisions and propagated variables, as denoted in the blue region in Figure 4. In contrast, J-watch enables non-chronological restoration of J-heap such that N roughly equals the difference of two cuts corresponding to respective J-frontier at level l_b and l_c. The reduction in heap operations from the scale of sub-circuit area to J-frontier size delivers better performance in our hybrid solver.

D. Engineering J-watch into CDCL framework

The overhead of using J-watches may be substantial during backtracking. For example, monitoring redundant justification degrades performance and simplicity. The following paragraph addresses the J-watch efficiency.

Definition 3. *g' is **J-watch-free** if $\texttt{level}(g) \leq \texttt{level}(g')$ holds for all $g \in \mathcal{J}(g')$.*

Theorem 1. *J-watch subsumes J-frontier membership of g' since $\texttt{level}(g')$.*

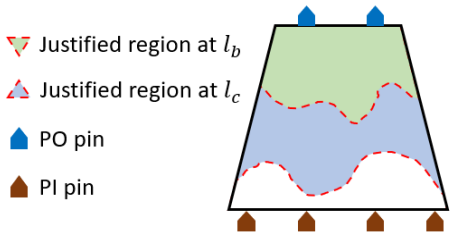

Fig. 4. Green and blue shapes are justified regions at level l_b and l_c respectively. The red dashed lines are J-frontiers.

Fig. 5. Venn diagram for justification related concepts.

Proof. If g' is J-watch-free and pending for justifying, then fanin $g \in \mathcal{J}(g')$ exists where $\texttt{level}(g') < \texttt{level}(g)$ holds. A contradiction. □

Note that, when g' has been justified and left J-frontier, g' is monitored by J-watch of its justification. Therefore, the inverse of Theorem 1 does not hold. Figure 5 depicts this relation.

Corollary 1. *1-valued and-gate g' is J-watch-free.*

Proof. Assigning 1 to an and-gate conceptually corresponds to the implications of 2-literal clauses where any unassigned fanin g has propagated at $\texttt{level}(g')$. A direct result from Theorem 1. □

Corollary 1 avoids part of redundant justification in performance-critical code. In other words, an and-gate assigned to 1 is either automatically justified, or conflicts during propagation. Furthermore, no need to monitor their justification by J-watch.

Corollary 2. *$|\mathcal{J}(g')| < 2$ holds for any and-gate.*

Proof. Corollary 1 indicates that output 0 is the only cared condition, where 0 is the controlling value of and-gate. □

Corollary 2 keeps engineering simplicity. To be specific, a gate g' can directly represent itself in unique $\mathcal{J}_w(g)$ without the need of extra placeholder.

Corollary 3. *$\mathcal{J}_w(g)$ and $\mathcal{J}_w(g')$, share the same semantic if $\texttt{level}(g) = \texttt{level}(g')$.*

Proof. Let $m \in \mathcal{J}_w(g)$ and $m' \in \mathcal{J}_w(g')$, then m and m' leave corresponding $\mathcal{J}_w$ simultaneously whenever backtracking occurs with $l_b < \texttt{level}(g)$. A direct result of Definition 2. □

Corollary 3 allows for a transition from the gate-based to level-based J-watch scheme, by showing the identical behavior between the two when backtracking occurs. From engineering perspective, Corollary 3 improves memory footprint, since it indicates that each decision level requires only one J-watch list, where the number of unique decision levels is often much fewer than the number of variables during constraint propagation.

Another improvement comes from the definition of J-watch itself. Theorem 1 avoids unnecessary J-heap operations. In other words, a gate g once requiring justification could become J-watch-free after propagation. In practice, we stack propagated variables until propagation has fully completed without conflict. Variables becoming J-watch-free are exempted from being pushed to J-heap.

TABLE II
SYMBOLS FOR DETAILED ANALYSIS OF INTERPRETATION

$\perp$	Unassigned value.
t	Traced variable.
r	Reason which implies value of t.
$\mathtt{I}(t,r)$	The clause deduced from reason r, implying t.
$\mathtt{D}(t)$	Direction of the first controlling fanin.

TABLE III
LOOKUP TABLE FOR INTERPRETING $s_0 = s_1 \wedge s_2$.

t	(s_0, s_1, s_2)	$\mathtt{I}(t, s_0)$	comments
s_0	$(0,0,\perp)$	$(\neg s_0 \vee s_1)$	covered by $(\neg s_0 \vee s_{\mathtt{D}(s_0)})$
	$(0,\perp,0)$	$(\neg s_0 \vee s_2)$	
	$(0,0,1)$	$(\neg s_0 \vee s_1)$	
	$(0,1,0)$	$(\neg s_0 \vee s_2)$	
	$(0,0,0)$	$(\neg s_0 \vee s_{\mathtt{D}(g)})$	
	$(1,1,1)$	$(s_0 \vee \neg s_1 \vee \neg s_2)$	
s_1	$(0,0,\perp)$	-	justification
	$(0,0,1)$	$(s_0 \vee \neg s_1 \vee \neg s_2)$	
	$(1,1,1)$	$(\neg s_0 \vee s_1)$	
s_2	$(0,\perp,0)$	-	justification
	$(0,1,0)$	$(s_0 \vee \neg s_1 \vee \neg s_2)$	
	$(1,1,1)$	$(\neg s_0 \vee s_2)$	

E. Interpreting Implication Graph On-the-fly

The edges in a hybrid implication graph can be clauses or circuit gates. Interfacing between the two is important for efficient *conflict analysis* and *learnt clause minimization* [18]. (Symbols in Table II are used in the following context.)

A typical CNF-based implication graph is a DAG where each propagated variable t is associated with a *reason clause* r implying the value of t. These properties allows us to define an interpretation function $\mathtt{I}(t,r)$ depicted in Figure 6, which takes a logic gate as the reason r of t and view the gate as a clause. Table III shows that such $\mathtt{I}(t,r)$ exists and gives the reason clause for the implication propagated by the gate s_0 uniquely, for each combination of (s_0, s_1, s_2) assignment and tracing target t. Based on the uniqueness of $\mathtt{I}(t,r)$ interpretation, it is trivial to show that $\mathtt{I}(t,r)$ generates an implication graph isomorphic

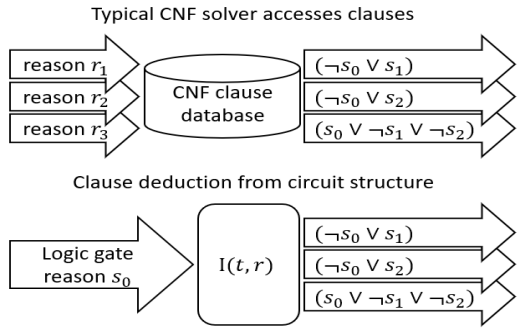

Fig. 6. Comparing CNF database and circuit clause deduction.

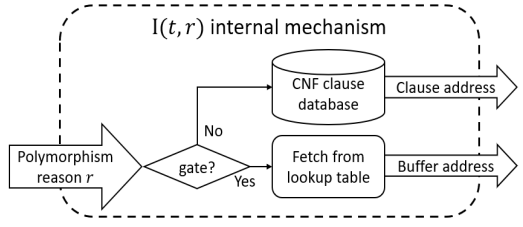

Fig. 7. $\mathtt{I}(t,r)$ casts clause queries in hybrid environment.

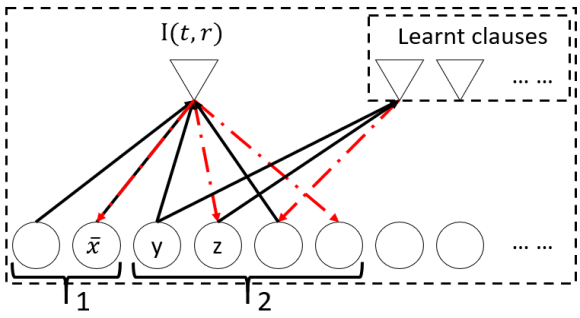

Fig. 8. The state of a circuit-based SAT solver using $\mathtt{I}(t,r)$ in Example 3.

to its pure CNF counterpart if the circuit gates are represented using CNF.

In practice, $\mathtt{I}(t,r)$ gives the result of querying the reason clause as shown in Figure 7, where r is a polymorphic datatype [19] capable of storing both a pure clause and a logic gate. During traversal of a hybrid implication graph, $\mathtt{I}(t,r)$ visits reason r one at a time and interprets clauses on-the-fly. Therefore, the temporary clause used to represent reasons for each implication generated by the circuit composed of two-input nodes can be stored in a three-literal buffer. In summary, the capability of traversing hybrid implication graph enables seamless adoption of existing clause minimization techniques [18] in the proposed solver.

Example 3. *Figure 8 depicts a circuit-based SAT solver using one-the-fly interpretation. Different from conventional CNF solvers shown in the Figure 1, the interpretation function $I(t,r)$ grants a single buffer the capability to represent arbitrary members in the set of circuit clauses during conflict analysis or learnt clause minimization. Given and-gate $z = \bar{x} \vee y$, according to Table III, while tracing z under the reason z, the buffer clause is interpreted to $I(z,z) = (z \vee x \vee \bar{y})$.*

F. Topological Abstraction for Solving Scope

Justifying circuits in large scale often requires incremental solving for better performance offered by learnt clauses. However, the accumulated sub-circuits in a solver are not necessarily relevant to every rounds of solving and can degrade performance. In order to resolve the trade-off, we apply topological abstraction, which recursively marks TFI from root nodes, to further refine the solving scope of current window. Afterwards, propagation and justification occur only in the relevant region, thus the window size of our solver can grow more aggressively for better reuse of the learnt information.

IV. EXPERIMENTAL RESULTS

The proposed solver is integrated into state-of-the-art SAT sweeper [14] in ABC [20] and can be called using command "*&fraig -x*". The experiments were run on AMD Ryzen 7 4800HS CPU with 40GB RAM. A single core and less than 1GB were used for any test case considered in this section. Due to the limited access to other circuit-based solvers, we compare our solver with original Glucose in a logic synthesis application, and also show the effect of our techniques.

A. Evaluating the Solver Performance in SAT Sweeping

Table IV shows experimental results on a subset of randomly selected large benchmarks from ITC'99, ISCAS'85/'89, IWLS'05 [21], and HWMCC'15 [22] benchmarks suites. Section "Statistics" lists the benchmark name (Name) (the number in parentheses next to the name, if present, shows the timeframe count used to unfold sequential AIGs), the number of primary inputs (PI), primary outputs (PO), logic levels (Lev), and internal and-nodes in the original AIG

(And). Section "Solver calls" lists the number of satisfiable calls (SAT) and total calls made by the SAT solver while running the SAT sweeper.

The section "SAT sweeping time" shows (1) the total time of SAT sweeping (column "Total"), (2) the time spent by the proposed solver, including data initialization and solving (column "Solver"), (3) the pure solving time (column "Solving"). The columns "NoCir" and "New" denote the runtime of using original Glucose and the proposed solver, respectively. The SAT solver conflict limit was disabled in all runs. However, the testcases running more than 900 seconds were aborted and the corresponding entries in the table contain a dash. The aborted runtime is assumed to be 900 seconds.

The last section shows improvements in runtime. The last row lists geometric averages for each runtime metric. The proposed solver runs 3.7x faster on average, resulting in a 2x speedup in SAT sweeping. Furthermore, if the data initialization is considered, including loading CNF into original Glucose and computing the cone of influence, the SAT solver speedup is about 4.2x. Besides, the huge improvement ratio shown in the case "s35932(40)" demonstrates the advantage of activity-based justification. The "NoCir" configuration encounters, on average, 1000 times more conflicts compared to the "New" in the satisfiable calls.

B. The Effect of J-Heap and J-Watch

The cost of using the heap can be evaluated based on the operation complexity and the heap size. Formula 1 shows the cost of applying N operations when the heap size is K. To consider the sum of individual costs of N cases with the respective heap size, the cost of J-heap operation is formulated as

$$\mathtt{S}_J = \sum_{i=1}^{N} log(K_i), \tag{2}$$

where N is the number of heap operations, including push and pop, indexed with i, performed when the heap has size K_i, in the actual order during the SAT solving process. In order to compare the effectiveness of using J-heap, the cost of operating a conventional heap is formulated in a similar way

$$\mathtt{S}_C = \sum_{i=1}^{N} log(U_i), \tag{3}$$

where U_i is the number of unassigned variables at the i-th heap operation. Due to the assignment made by propagation, the actual heap size is often larger than U_i. Thus, it is fair to use U_i as a conservative estimate of the conventional heap size.

Formula 2 and 3 allow for the comparison of computational cost between a J-heap and a conventional heap for the common push and pop operations. In addition, the effect of not using J-watch can be formulated as the penalty cost

$$\mathtt{P}_C = \sum_{j=1}^{A} log(U_j), \tag{4}$$

where there are A indexed heap operations, representing the push operations ruled out when $\mathtt{level(g')} > l_b$ occurs in Definition 2. On the other hand, the U_j shares the definition with Formula 3. The subscript j is used to emphasize that N and A refer to two different sets of heap operations occurring at different times.

Table V compares the computational cost of the experiments in Table IV. The section "Cost" shows the computational cost $\mathtt{S}_J$, $\mathtt{S}_C$, and the penalty cost $\mathtt{P}_C$ defined in Formula 2, 3 and 4 respectively. In the case of benchmark "s35932(20)", the three cost values are zeros since the equivalence is proved at top decision level while the non-equivalence is solved by simulation. The section "Op. #" shows the number of heap operations "N", including push and pop, described by Formula 2 and 3, while "A" is the skipped push described by Formula 4. The next section shows (1) the ratio of the cost using J-heap to that of using conventional heap without J-watch (column "$\mathtt{S}_J/(\mathtt{S}_C + \mathtt{P}_C)$"), (2) the ratio of the number of the skipped push operations A to the total heap operations (column "$A/(N + A)$"), (3) the ratio of the penalty cost to the J-heap cost (column "$\mathtt{P}_C/\mathtt{S}_J$").

The last row shows geometric averages for each cost metric. The computational cost of using J-heap is, on average, 25% compared to not using J-heap. The penalty cost $\mathtt{P}_C$ of not using J-watch in the conventional heap, i.e. non-chronological restoration, can be as high as 71% of the $\mathtt{S}_J$ cost, even if the skipped push operations amount to only 19% to the total heap operations.

V. Conclusions

The paper introduces an efficient circuit-based SAT solver for logic synthesis applications. In such applications, the solver typically processes a sequence of incremental SAT problems, which are numerous (typically more than a thousand, possibly a few million), topologically related (have overlapping logic cones), and relatively simple (the majority of them is solved after a few conflicts).

To address such problems in an application-specific solver, several novel data-structures (such as J-heap and J-watch) are used in combination with J-frontier, which is a well-known solution for tracking relevant nodes in circuit-based solvers. Additionally, several novel implementation tricks for developing and integrating circuit-based solvers are presented.

The proposed solver has been tried in several applications, in particular, in the context of SAT sweeping, that is, proving and merging of equivalence nodes in a combinational logic circuit. The experimental results show that the proposed solver achieves a 4x speedup of SAT solving, resulting in a 2x speedup of SAT sweeping, compared to a well-tuned integration of the original CNF-based Glucose.

Future work may include extending the solver to work on larger gates, such as muxes and multi-fanin and-nodes, and integration into applications using observability don't-cares, since currently it supports only satisfiability don't-cares.

References

[1] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in boolean networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 5, pp. 743–755, 2006.

[2] N. Eén and N. Sorensson, "An extensible SAT-solver," in *The International Conferences on Theory and Applications of Satisfiability Testing (SAT)*, pp. 502–518, 2003.

[3] G. Audemard and L. Simon, "SAT solver Glucose 3.0 (2013)." http://www.labri.fr/perso/lsimon/glucose/.

[4] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," in *IBM Journal of Research and Development*, vol. 10, pp. 278–291, 1966.

[5] J. P. M. Silva and K. A. Sakallah, "GRASP-a new search algorithm for satisfiability," in *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pp. 220–227, 1996.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proceedings of Design Automation Conference (DAC)*, 2001.

[7] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik, "Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver," in *Proceedings of Design Automation Conference (DAC)*, pp. 747–750, 2002.

[8] F. Lu, L.-C. Wang, K.-T. Cheng, and C.-Y. R. Huang, "A circuit SAT solver with signal correlation guided learning," in *Proceedings of Design, Automation and Test in Europe*, pp. 892–897, 2003.

[9] F. Lu, M. K. Iyer, G. Parthasarathy, L.-C. Wang, K.-T. Cheng, and K. Chen, "An efficient sequential SAT solver with improved search strategies," in *Proceedings of Design, Automation and Test in Europe*, 2005.

TABLE IV
COMPARING THE SAT SOLVER RUNTIME WITH AND WITHOUT THE PROPOSED NOVEL FEATURES. (TIMEOUT=900 SEC)

Statistics					Solver calls		SAT sweeping time [14]						Runtime ratio NoCir/New		
							Total		Solver		Solving				
Name	PI	PO	Lev	And	SAT	Total	NoCir	New	NoCir	New	NoCir	New	Total	Solver	Solving
b07(100)	100	800	1457	36600	1203	5974	41.06	21.01	39.98	20.10	38.35	19.77	1.95	1.99	1.94
b07(50)	50	400	732	18300	448	2669	4.49	3.48	4.11	3.11	3.76	3.01	1.29	1.32	1.25
b18(10)	370	230	486	817100	11867	31754	671.75	111.18	612.14	75.97	510.66	58.35	6.04	8.06	8.75
b18(15)	555	345	664	1225650	16786	46708	-	519.68	-	416.80	-	351.22	1.73	2.16	2.56
b19(5)	120	150	474	817600	16404	37166	231.69	50.78	202.71	27.82	158.21	17.89	4.56	7.29	8.84
b19(7)	168	210	628	1144640	21239	50455	835.74	159.59	749.41	93.91	583.40	59.64	5.24	7.98	9.78
s35932(20)	700	6400	380	238960	0	5760	3.15	0.90	2.73	0.49	0.08	0.03	3.50	5.57	2.67
s35932(40)	1400	12800	760	477920	8	11528	265.22	5.00	264.35	4.21	244.82	0.19	53.04	62.79	1288.53
s38584(10)	120	2780	343	120553	2106	5509	11.95	4.11	10.33	2.55	8.05	2.05	2.91	4.05	3.93
s38584(15)	180	4170	513	180538	3445	8558	47.26	17.18	43.10	13.34	37.00	11.95	2.75	3.23	3.10
leon2	298888	291880	58	789647	2140	3061	24.69	11.30	15.05	1.94	14.08	1.76	2.18	7.76	8.00
netcard	195730	97805	40	803848	6	599	3.82	3.68	0.19	0.04	0.02	0.00	1.04	4.75	1.00
RISC	15678	8111	40	75613	72	941	0.34	0.30	0.04	0.01	0.02	0.01	1.13	4.00	2.00
vga_lcd	34247	21412	24	126708	0	159	0.22	0.22	0.00	0.00	0.00	0.00	1.00	1.00	1.00
6s100	127138	97599	79	636637	2504	10189	4.71	2.92	2.25	0.48	1.59	0.35	1.61	4.69	4.54
6s203b41	80192	68958	65	474322	203	5525	2.40	2.29	0.12	0.04	0.05	0.03	1.05	3.00	1.67
6s281b35	268334	177236	121	2076248	10695	17095	55.12	13.54	42.79	2.54	37.13	1.82	4.07	16.85	20.40
6s299b685	719410	467370	75	4111296	902	59972	20.91	19.81	1.76	0.61	0.74	0.42	1.06	2.89	1.76
6s322rb646	82513	80928	108	641468	32	22365	2.88	2.17	0.96	0.26	0.26	0.14	1.33	3.69	1.86
6s342rb122	59253	56839	52	330130	191	3221	0.58	0.52	0.11	0.06	0.02	0.04	1.12	1.83	0.50
6s350rb46	245680	243400	194	1550412	112	3428	8.37	7.04	1.50	0.21	0.64	0.07	1.19	7.14	9.14
6s382r	106395	104831	2752	1756654	1493	6246	36.58	32.74	26.03	22.27	24.63	22.00	1.12	1.17	1.12
6s387rb291	30615	29495	30	330186	251	14760	0.92	0.78	0.26	0.09	0.12	0.07	1.18	2.89	1.71
6s392r	80920	80151	538	1599275	582	2877	1.61	1.32	0.37	0.09	0.25	0.08	1.22	4.11	3.13
												geomean	2.09	4.18	3.70

PI and PO reported include both primary inputs/outputs and flop outputs/inputs.

TABLE V
COMPARING COMPUTATIONAL COST OF USING J-HEAP VS USING CONVENTIONAL HEAP

Name	Cost			Op. #		Ratio		
	${}^{\mathtt{S}}J$	${}^{\mathtt{S}}C$	${}^{\mathtt{P}}C$	N	A	$\frac{{}^{\mathtt{S}}J}{{}^{\mathtt{S}}C+{}^{\mathtt{P}}C}$	$\frac{A}{N+A}$	$\frac{{}^{\mathtt{P}}C}{{}^{\mathtt{S}}J}$
b07(100)	3.28E+07	5.43E+07	6.21E+06	4.10E+06	4.94E+05	0.54	0.11	0.19
b07(50)	4.85E+06	8.64E+06	9.08E+05	6.82E+05	7.62E+04	0.51	0.10	0.19
b18(10)	9.99E+07	2.32E+08	7.03E+07	1.36E+07	4.22E+06	0.33	0.24	0.70
b18(15)	4.25E+08	9.02E+08	2.15E+08	4.97E+07	1.20E+07	0.38	0.20	0.50
b19(5)	6.23E+07	1.60E+08	6.63E+07	1.03E+07	4.31E+06	0.28	0.30	1.06
b19(7)	1.33E+08	3.41E+08	1.32E+08	2.03E+07	8.01E+06	0.28	0.28	1.00
s35932(20)	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	-	-	-
s35932(40)	7.06E+03	3.35E+04	1.46E+04	1.81E+03	7.94E+02	0.15	0.30	2.07
s38584(10)	6.42E+06	1.51E+07	4.29E+06	9.85E+05	2.85E+05	0.33	0.22	0.67
s38584(15)	2.91E+07	5.87E+07	1.29E+07	3.64E+06	8.13E+05	0.41	0.18	0.44
leon2	2.23E+07	4.17E+07	2.07E+07	2.47E+06	1.23E+06	0.36	0.33	0.93
netcard	8.58E+02	7.97E+03	1.72E+03	5.00E+02	1.03E+02	0.09	0.17	2.01
RISC	5.14E+04	1.49E+05	4.74E+04	1.33E+04	4.27E+03	0.26	0.24	0.92
vga_lcd	1.50E+01	3.89E+02	0.00E+00	4.10E+01	0.00E+00	0.04	-	-
6s100	7.33E+06	1.43E+07	6.99E+06	1.16E+06	5.66E+05	0.34	0.33	0.95
6s203b41	1.18E+05	4.61E+05	9.61E+04	4.18E+04	8.46E+03	0.21	0.17	0.82
6s281b35	3.48E+06	1.66E+07	7.55E+06	1.15E+06	5.24E+05	0.14	0.31	2.17
6s299b685	9.30E+05	2.42E+06	3.71E+05	2.10E+05	3.29E+04	0.33	0.14	0.40
6s322rb646	2.85E+03	1.90E+04	9.25E+02	1.66E+03	8.30E+01	0.14	0.05	0.33
6s342rb122	2.45E+05	5.82E+05	2.28E+05	5.25E+04	2.07E+04	0.30	0.28	0.93
6s350rb46	7.60E+04	2.74E+05	1.12E+05	1.89E+04	7.55E+03	0.20	0.29	1.47
6s382r	4.23E+07	7.87E+07	2.10E+07	6.01E+06	1.85E+06	0.42	0.24	0.50
6s387rb291	8.62E+04	6.19E+05	3.10E+04	6.08E+04	3.30E+03	0.13	0.05	0.36
6s392r	1.70E+06	3.37E+06	1.13E+06	2.92E+05	9.92E+04	0.38	0.25	0.66
					geomean	0.25	0.19	0.71

[10] C.-A. Wu, T.-H. Lin, C.-C. Lee, and C.-Y. R. Huang, "Qutesat: A robust circuit-based SAT solver for complex circuit structure," in *Proceedings of Design, Automation and Test in Europe*, 2007.

[11] J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet, "SBSAT: A state-based, bdd-based satisfiability solver," in *Theory and Applications of Satisfiability Testing (SAT)*, pp. 398–410, 2003.

[12] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing (SAT)*, pp. 244–257, 2009.

[13] H.-T. Zhang, J.-H. R. Jiang, and A. Mishchenko, "A circuit-based SAT solver for logic synthesis," in *Proceedings of International Workshop on Logic and Synthesis (IWLS)*, 2000.

[14] H.-T. Zhang, J.-H. R. Jiang, L. Amarú, A. Mishchenko, and R. Brayton, "Deep integration of circuit simulator and SAT solver," in *Proceedings of Des. Aut. Conf. (DAC)*, 2021.

[15] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Leningrad Seminars on Mathematical Logic*, 1966.

[16] J. P. M. Silva, I. Lynce, and K. A. Sakallah, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, IOS press, 2009.

[17] M. Ganay and A. Kuehlmann, "On-the-fly compression of logical circuits," in *Proceedings of International Workshop on Logic and Synthesis (IWLS)*, 2000.

[18] N. Soörensson and A. Biere, "Minimizing learned clauses," in *The International Conferences on Theory and Applications of Satisfiability Testing (SAT)*, pp. 237–243, 2009.

[19] B. Jacobs, "Categorical logic and type theory," in *Studies in Logic and the Foundations of Mathematics*, vol. 141, Elsevier, 1999.

[20] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proceedings of International Conference on Computer Aided Verification (CAV)*, pp. 24–40, 2010.

[21] "IWLS 2005 benchmarks." https://iwls.org/iwls2005/benchmarks.html.

[22] "HWMCC 2015 benchmarks." http://fmv.jku.at/hwmcc15/.