VeriSAT: the Hardware Design of Modern SAT Solver

Yue Tao[1], Shaowei Cai[1,2,*]
[1]*Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences*
[2]*School of Computer Science and Technology, University of Chinese Academy of Sciences*
Beijing, China
taoyue@iscas.ac.cn, caisw@ios.ac.cn

***Abstract*—VeriSAT is the first modern SAT solver implemented entirely in synthesizable SystemVerilog, leveraging FPGA architecture for hardware acceleration. This paper introduces the design of VeriSAT, focusing on hardware-specific optimizations that significantly improve performance over traditional software-based solvers. By rethinking key SAT components for hardware, VeriSAT introduces custom data structures and parallelized processes to accelerate solving efficiency.**

Central to VeriSAT's architecture are hardware-optimized data structures. A linked-list based literal-watching mechanism, enhanced with cached watching literals, reduces latency in unit propagation. Additionally, a concurrent propagation tree enables simultaneous traversal for faster conflict detection. The solver's pipelined clause learning further boosts throughput, allowing rapid conflict analysis without compromising performance.

Extensive benchmarking demonstrates that VeriSAT outperforms two other FPGA-based solvers, SAT-Hard and SAT-Accel, by a factor of 1044x and is 18x faster, respectively, on curated benchmark instances. Additionally, VeriSAT shows a 30x speedup over the popular CPU-based MiniSat solver for specific datasets, validating the effectiveness of our design optimizations.

VeriSAT represents a significant leap forward in FPGA-based SAT solving, offering unprecedented efficiency through its hardware-tailored design. This work lays the foundation for future advancements in hardware-accelerated SAT solvers and paves the way for more scalable, high-performance solutions in both academic and industrial applications.

***Index Terms*—FPGA, CDCL, SAT, solver, hardware, verification**

I. Introduction

The Boolean satisfiability (SAT) problem is a classical problem of proving whether an assignment of truth values to variables exists to make the Boolean formula true. It is a hard problem in computer science and is the first problem identified as NP-Complete[2]. Meanwhile, due to the simplicity of its form, a wide range of problems from diverse domains can be naturally expressed as SAT problems, including traditional tasks of artificial intelligence, such as the automatic theorem proving [10, 25, 18] and automated planning [35]. In the realm of electronic design automation (EDA), SAT solvers are instrumental in tasks such as formal verification [6], combinational equivalence checking [7, 27, 33], and automatic test pattern generation [3, 30].

A SAT solver is a software tool that determines the satisfiability of a given Boolean formula, typically expressed in Conjunctive Normal Form (CNF). After decades of development, modern SAT solvers are now capable of efficiently handling very large and complex problems from various domains. Recent advancements in SAT solver technology focus on three main areas:

1) **Heuristic improvements:** More intelligent variable selection strategies have been developed to prioritize variables that are likely to lead to conflicts, increase the speed of solving and reduce the search space [13, 24].
2) **Preprocessing and in-processing:** Simplification of the original or intermediate problem before or during solving[28, 17].
3) **Parallelism:** The development of parallel SAT solvers utilizing multicore and distributed computing allows the division of the solving process across multiple instances, optimizing performance through intelligent job scheduling and reduced communication overhead [34, 22].
4) **Optimization towards hardware:** Improvements in the use of modern CPU architectures, such as adopting of lazy evaluation data structures [8], utilizing custom memory allocators [13], and compiler directives for enhanced cache prefetching [32].

As the complexity of modern computing devices and compilers grows, optimizing algorithms to fully leverage hardware becomes more difficult. This paper takes a different approach, starting with the hardware itself, designing and optimizing a dedicated hardware solution for SAT solving to achieve superior performance and energy efficiency. However, this endeavor presents several challenges:

1) **Design Paradigm Shift:** Software algorithms assume abundant, flexible storage and high-level operation abstractions, whereas hardware requires adapting to fixed resources and constraints, necessitating a rethinking of the algorithm's structure.
2) **Ensuring Correctness and Completeness:** Unlike software, hardware implementations must explicitly manage all aspects of the algorithm to guarantee correctness and completeness.
3) **Optimizing for Hardware:** Identifying components of the algorithm that can be parallelized or pipelined to fully exploit the advantage of hardware's flexibility and concurrence.

To address these challenges, we present **VeriSAT**, a SAT solver designed with synthesizable SystemVerilog and implemented in FPGA for precise tracking of hardware structure and behavior at fine granularity, enabling full visibility into the solver's internals for targeted optimizations and efficient hardware mapping of each solving stage. In particular, following contributions are made in this paper:

- **Optimized Memory Scheme and Data Structure:** We designed a hardware-friendly memory scheme and custom data structures tailored for FPGA, minimizing memory latency and maximizing throughput.
- **Concurrent Propagation Engine:** To enhance the efficiency of the unit propagation stage, we have developed a concurrent propagation engine capable of simultaneously processing multiple propagation operations. This parallelization significantly reduces the time required to propagate literals, accelerating the overall performance of the solver during this critical phase.
- **Pipelined Conflict Analysis Engine:** In the stage of conflict analysis and clause learning, we have implemented **pipeline optimizations** to handle the complex tasks of identifying conflicts and generating learned clauses in a more efficient manner. This pipeline design improves throughput and reduces the cycle time

This work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDA0320000 and XDA0320300), and by the CCF-Huawei Populus Grove Fund.

*Corresponding author: Shaowei Cai (caisw@ios.ac.cn).

for resolving conflicts and makes the discovery of conflict faster, which is crucial to maintaining high performance in solving SAT problems.

- **Extensive Benchmarking and Comparison:** We conducted extensive experiments to evaluate the performance of **VeriSAT** against two other FPGA-based CDCL SAT solvers: SystemVerilog-based **SAT-Hard** and HLS-based **SAT-Accel**. The results show that **VeriSAT** outperforms **SAT-Hard** by a factor of **1044x** and is **18x faster than SAT-Accel** on curated benchmark instances, showcasing the effectiveness of our design choices and optimizations. We also conducted comparison between MiniSat on CPU, which shows that VeriSAT outperforms MiniSat by **30x** for specific datasets.

II. Preliminaries

A. Boolean Satisfiability Problem

The Boolean Satisfiability (SAT) problem is about proving the satisfiability of a Boolean formula. A Boolean formula consists of a finite set of Boolean variables, assuming $V = \{x_1, x_2, ..., x_n\}$. A literal l of the variable x is either x or its negation $\neg x$. A clause ω is a disjunction of literals, given by $\bigvee_j l_j$. A Conjunctive Normal Form (CNF) formula ϕ is given by:

$$\phi = \bigwedge_i \omega_i = \bigwedge_i \bigvee_j l_{ij}$$

Variables assume one of three states: True, False, or Undet (unassigned). A literal l is *satisfied* if it evaluates to True under the current assignment, and *unsatisfied* if it evaluates to False. A clause (a disjunction of literals) is *satisfied* if at least one literal is satisfied, *unsatisfied* if all literals are unsatisfied and remain Undet otherwise. A clause with exactly one Undet literal and all others unsatisfied is called a *unit clause*.

We distinguish between two notions of unit clause. A *structural unit clause* refers to a clause that contains exactly one literal in the original or learned formula. In contrast, during solving, we refer to a clause as a *unit clause under assignment* when it contains exactly one unassigned literal and all others are currently falsified. Unless otherwise specified, we use the ’unit clause’ to refer to this assignment-dependent definition.

B. Modern SAT Solving Algorithm

The first practical complete algorithm is *DPLL*[1] (Davis-Putnam-Logemann-Loveland), which explores all possible assignments to variables using a depth-first search approach. *Conflict-Driven Clause Learning (CDCL)*, the successor of DPLL, incorporates *clause learning* and *non-chronological backtracking*, which significantly reduce search space and allow solvers to solve larger and more complex instances more efficiently. The CDCL algorithm is widely considered the core of modern SAT solvers, such as *MiniSat* [13] and *Kissat* [29], which are optimized with advanced heuristics and data structures. Modern SAT solvers have introduced several optimizations to further improve performance, such as dynamic *heuristics* for variable selection, *restarts* to escape local minima, *clause elimination* to reduce the size of the clause database, and advanced memory management techniques to optimize resource usage.

The Conflict-Driven Clause Learning (CDCL) algorithm consists of several fundamental stages that guide its operation through the search space. In the following, we describe these key components:

- **Unit Propagation** (`prop`): This stage repeatedly applies unit propagation, deducing the values of variables based on the current partial assignment. The process continues until either a conflict is encountered or no further unit clauses can be propagated.

Algorithm 1: Modern CDCL Algorithm Framework

```
1  while $res$ = Undef do
2      if prop$(F, V)$ = CONF then
3          $bakLev \leftarrow$ conf$(F, V)$;
4          if $bakLev < 0$ then
5              $res \leftarrow$ UNSAT;
6          else
7              back$(F, V, bakLev)$;
8      else
9          if all variables are assigned then
10             $res \leftarrow$ SAT;
11         else
12             decide$(V)$;
13             updateHeuristics$(V)$;
14             if a restart condition is met then
15                 restart$(F, V)$;
16     if rephase condition is met then
17         rephase$(V)$;
18     if clause deletion condition is met then
19         reduceClauseDB$(F)$;
20 return $res$;
```

- **Conflict Analysis** (`conf`): When a conflict is detected, the solver analyzes the conflicting clauses to identify a new learned clause. This process identifies the *First Unique Implication Point* (First UIP), which represents a key decision point in the implication graph that causally leads to the conflict.
- **Non-chronological Backtracking** (`back`): The algorithm then backtracks non-chronologically to the First UIP. By backtracking to the appropriate level and enforcing the learned clause (which typically negates the UIP), CDCL guarantees that the solver does not revisit the same conflict based on the current partial assignment.
- **Variable Decision** (`decide`): If no conflict occurs, a new decision is made by selecting a variable and assigning a value to it. Typically, heuristics are used to prioritize variables, improving search efficiency by guiding the solver toward likely solutions.

CDCL can be viewed as a process of *consecutive refutation*: each time unit propagation uncovers a conflict, the solver locates the First UIP in the implication graph and derives a learned clause that blocks the conflicting partial assignment. By backjumping non-chronologically to just before the First UIP and adding this clause to the database, CDCL incrementally prunes the conflicting region of the search space, avoiding redundant exploration. Repeating this cycle progressively tightens the formula, allowing CDCL to converge rapidly when refuting unsatisfiable formulas.

While CDCL excels at proving unsatisfiability through successive refutations, efficiently finding a satisfying assignment requires careful guidance of the search process. To this end, modern SAT solvers rely heavily on heuristics to prioritize decisions and clause management. Variable selection heuristics and clause quality metrics play crucial roles in steering the solver toward promising regions of the search space, accelerating the discovery of a satisfying assignment when one exists.

These techniques include:

- **Restart**: Periodic restarts help the solver escape local minima by abandoning the current search state and beginning a new search. This prevents the solver from becoming stuck in unfruitful areas of the search space.
- **Phase Saving**: Phase saving retains the most recent variable assignments from previous search attempts, which can be reused in future decisions. This minimizes the impact of unsuccessful searches and guides the solver toward a solution.
- **Variable Decision Heuristics**: These heuristics guide the decision-making process in choosing which variables to assign. By dynamically adjusting the decision strategy based on the search history, heuristics help prioritize variables that are more likely to lead to a solution.

Besides, what is not shown in the algorithm list is the data structure of storing information.

III. Progress of FPGA-based SAT Solvers and Accelerators

A. Early Works

The idea of accelerating SAT problem solving with FPGAs dates back to the late 1990s and early 2000s. Skliarova et al. [11] summarized the early efforts in this domain, noting that, with the gradual adoption of commercial FPGA devices, pioneering research began to explore their potential for SAT solving. Notable early works include Suyama et al. [9], Plazner [5], and Zhong [4], among others. These works introduced the first category of FPGA-based SAT solvers, referred to as *instance-specific* implementation. The limited hardware resources and the need for frequent reconfiguration led to the decline of this approach.

B. Recent Works

Several notable advancements in FPGA-based SAT solving emerged after 2017, as summarized by Sohanghpurwala. [23]. During this period, research focused on both incomplete and complete SAT solving algorithms, exploiting FPGA's parallelism capabilities. Early FPGA implementations, including those based on local search algorithms like WSAT, demonstrated the potential for hardware acceleration in SAT solving.

For incomplete SAT algorithms, FPGA implementations of *WSAT* were explored in [12] and [15], while [20] describes an FPGA implementation of *ProbSAT*, another local search-based approach. These works highlighted FPGA's ability to efficiently explore the solution space; however, the limitations of local search algorithms meant that these solutions could not guarantee satisfiability or unsatisfiability.

As FPGA hardware matured, many FPGA-based complete SAT solvers adopted heterogeneous architectures, using FPGAs as accelerators for specific parts of the CDCL algorithm. Operations such as unit propagation and clause learning were recognized for their potential to be parallelized or pipelined, which led to several efforts in FPGA-based acceleration. For example, [21] and [14] attempted to accelerate the unit propagation stage by creating multiple hardware threads, while [16] used FPGAs to speed up the conflict analysis procedure.

For complete SAT solvers implemented directly in hardware, a batch of DPLL-based designs are proposed , such as in [31] and [19]. These implementations focus on embedding the DPLL algorithm into FPGA hardware for more efficient SAT solving. A recent contribution in this direction is SAT-Hard [26], which proposes a monolithic state-machine-based implementation of the full CDCL framework.

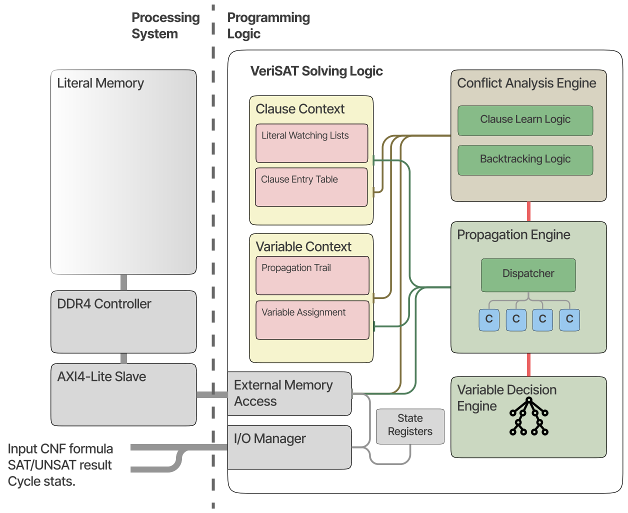

Fig. 1. The Overall architecture of VeriSAT

Additionally, SAT-Accel [36], a high-level synthesis (HLS)-based SAT solver, further pushes the boundaries of hardware-accelerated SAT solving.

IV. VeriSAT

A. Overview of VeriSAT design

As depicted in Figure 1, The solving logic of VeriSAT solver can be organized as three principal modules, each corresponding to a core step of the CDCL framework:

- **Propagation Search Engine (PSE)**: Implements Boolean Constraint Propagation (BCP). PSE scans watched clauses to detect conflicts or unit clauses. Upon encountering a conflict, it invokes the Conflict Analysis Engine; if no conflict and no further unit clauses remain, it hands control to the Variable Decision Engine.
- **Conflict Analysis Engine (CAE)**: Performs conflict analysis when PSE reports a conflict. CAE analyzes the implication graph to derive a learned clause and compute the backtracking level. If the backtracking level is negative, the solver concludes UNSAT. Otherwise, CAE appends the learned clause to the clause database, backjumps to the computed decision level, and returns the control to PSE.
- **Variable Decision Engine (VDE)**: Responsible for branching decisions. VDE applies a dynamic variable selection heuristic to pick an unassigned variable and assigns it a truth value. If all variables are assigned without conflict, the solver concludes SAT. Otherwise, VDE updates the heuristic data structures and invokes PSE to continue propagation.

It should be noted that, based on the foundation of the current CDCL algorithm, all three main modules must run alternately to maintain the correctness and completeness of the algorithm. Unlike other hardware pipelining schemes that require multiple modules to run in the same time window, making any modules mentioned above work concurrently will violate the correctness and completeness of the algorithm.

B. Memory Scheme in VeriSAT

The memory scheme of **VeriSAT** is designed to decouple context and memory storage as much as possible, with each functional module owning its respective memory. Unlike other FPGA applications,

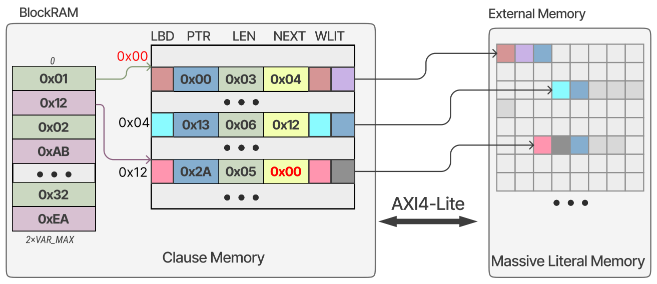

Fig. 2. The Memory scheme of Clause Entry Table and Literal Watching Lists

such as stream processors, where data flow and is processed on the fly, SAT solving algorithms like CDCL are stateful and involve massive search spaces. Therefore, the design of the memory scheme and storage devices must come first, before translating the procedural algorithm into hardware logic. To optimize for both capacity and extensibility, the memory is organized into a multi-level memory hierarchy, with distinct types of memory allocated based on their usage and access patterns. This approach ensures that the layout and behavior of the memory system directly influence the algorithmic logic, allowing for efficient and scalable hardware implementations.

VeriSAT makes extensive use of two types of memory: on-chip memory (OCM), typically arranged in the FPGA fabric and directly accessible by the logic, and external memory, such as DDR-SDRAM, which can be accessed through peripherals or custom communication devices. For the SAT solving logic, the memory is categorized into two primary contexts that are critical for the CDCL algorithm: the *clause context* and the variable context. The *clause context* deals with the CNF formula itself, including literals and clauses. Its memory management is structured as follows:

- **Literal Memory**: All literals are stored in external memory because of their large volume and infrequent updates. This storage choice ensures that the memory footprint remains manageable while maintaining quick access to clauses. Two registers are set to record the current number of literals and the address pointer of the newly written literal, for helping the clause addition. Note that when a true unit clause (the clause that contains only one literal) is being added, either from original clauses or learned clauses, it will trigger backtracking and assign the variable as assumption.
- **Clause Entry Table and Literal Watching Lists**: The clause entry table is stored linearly in OCM. In most cases, the table is not moved or shifted; new clauses, whether from initialization or clause learning, are simply appended at the end of the table. Each clause entry also caches frequently checked literals and stores the LBD (literal block distance) information for restarting. This design optimizes the memory access time by minimizing the need to repeatedly fetch literals from external memory.
  The watching list is implemented as a linked list built upon the clause entry table, as shown in Fig, 2. In addition to a list of list headers, the clause entries themselves are linked together, allowing for flexible clause management. This structure enables efficient reordering of clause entries within the same watching list without the need to move or copy the actual data.
  This approach provides two key benefits. First, the operations are significantly simplified: the need for frequent data movement and memory overwriting is eliminated, avoiding the complexities of designing a dynamic memory allocator on FPGA. Second, the atomic operations allowed by this design enable concurrent read/write access to the watching lists, improving overall performance and scalability.

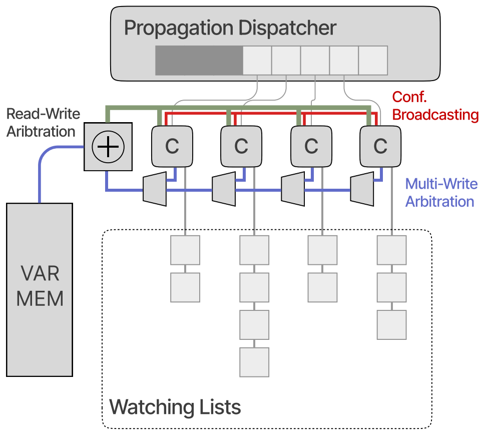

Fig. 3. The anatomy of Propagation Engine

The **variable context** is responsible for storing the state of the current variables and their associated information:

- **Trail**: The trail is a core memory structure that underpins the unit propagation, clause learning, and backtracking processes. It represents the current state of the solving process and is crucial for all solving logic. The trail is stored in OCM for high-speed access, and additional contextual information is associated with it, such as the decision level and reason clause storage. Controlled by the propagation engine, the trail is designed to support safe concurrent access, incorporating mechanisms for read-write arbitration and prioritized writes to ensure efficient operation during these critical tasks.
- **Variable Assignment**: The variable assignments need to be updated and accessed concurrently during the search process. To ensure fast access, this information is stored in LUTRAM, which guarantees that the results are available with 1 clock cycle.
- **Historical Phases and Variable Activity Scores**: These data structures are used to inform the heuristics when selecting new variables. The **activity scores** reflect the importance of each variable, helping the solver prioritize the variables that are more likely to lead to a solution.

In summary, the memory scheme in VeriSAT is carefully designed to support the core operations of the CDCL algorithm, while leveraging the FPGA's memory hierarchy for optimal performance. By efficiently managing memory for both the **clause context** and the **variable context**, the memory scheme well supports the algorithm built upon it.

C. Unit Propagation with the Concurrent Propagation Engine

1) Design of the Concurrent Propagation Engine: Traditional software SAT solvers process the trail of implied literals strictly sequentially, scanning each watching list one literal at a time. In contrast, **VeriSAT**'s **Concurrent Propagation Engine (CPE)** employs a single *dispatcher* and multiple *concurrent* cursors to exploit FPGA concurrency (shown in Figure 3). The dispatcher and cursors operate as follows:

1) **Dispatching.** When a new literal is assigned (either by decision or backjump), the dispatcher immediately assigns it to an idle cursor, which begins scanning the watch list of that literal.
2) **Concurrent Propagation.** Whenever a cursor discovers a propagatable literal, it returns it to the dispatcher, which enqueues it on the trail and dispatches another cursor to scan its corresponding watch list.
3) **Conflict Broadcast.** If any cursor detects a conflict, it reports this to the dispatcher, which then issues an instantaneous halt signal to all other cursors and forwards the conflicting clause to the Conflict Analysis Engine.
4) **Completion.** If no conflict arises, the dispatcher continues to assign the remaining unscanned literals to idle cursors. Once all newly enqueued literals have been processed and no further propagation is possible, control returns to the Variable Decision Engine.

2) Maintaining Read-Write Consistency: The concurrent execution of cursors may cause data hazard, we implement a lightweight arbitration mechanism to maintain the read-write consistency:

- **Read-Write Arbitration:** Before each read or write, every cursor broadcasts its target address to a comparison network. If a cursor attempts to read an address currently being written by another cursor , the network detects an address match and stalls the reading cursor until the writing one finishes. This enforces true Read-After-Write semantics both for checking variable assignments and for moving clause entries between watching lists.
- **Multi Write Arbitration:** When multiple cursors contend to write simultaneously (to the assignment table or clause database), the fixed priority scheme ensures that only the highest priority cursor proceeds, preventing livelock and guaranteeing forward progress.

Because unit propagation is confluent (the final conflict or assignment set does not depend on operation order), these concurrency controls preserve the logical behavior of the CDCL algorithm while delivering significant performance gains on FPGA. The number of cursors used in the system is determined by the available hardware resources, including the number of ports for communicating with external memory and the logical resources required for implementing the comparison network.

D. Conflict Analysis Engine

The Conflict Analysis Engine (CAE) module implements the essence of the First-UIP method entirely on the FPGA fabric. The First-UIP is identified as the most recently assigned decision at the current highest decision level. By negating the value of this assignment, the same conflict can be avoid for subsequent assignment. The learned clause, as the result of consecutive resolution, can avoid entering the same conflict regardless to the current assignment. The procedure is listed in Algorithm 2.

Unlike the propagation check, the procedure of clause learning requires to check each literal of the clauses that participated the unit propagation.

The process of checking if a literal is the First UIP consists of three steps, 1) retrieve the literal, 2) check the decision level, and 3) the visited mark. In our implementation, retrieving literal from external requires 4 cycles according to the clock of FPGA, and retrieving the decision level and visited mark require another 3 cycles. (the first cycle is for setting the read address.) And a comparison of the literal's decision level with the highest level of conflict clause is

Algorithm 2: Conflict Analysis with Inlined Clause Minimization

```
Input : Conflict clause $C$, decision levels $level[\cdot]$,
        reason clauses $reason[\cdot]$, assignment trail $trail$
Output: Learned clause $learnt$, backtrack level
$learnt \leftarrow [\,]$;
$resLit \leftarrow 0$;
$topLevel \leftarrow level(|C[0]|)$;
repeat
    $paths \leftarrow 0$;
    for $i \leftarrow (resLit = 0)\ 0\ :\ 1$ to $|C| - 1$ do
        $lit \leftarrow C[i]$;
        $var \leftarrow |lit|$;
        if $\neg visited[var] \wedge level[var] > 0$ then
            $visited[var] \leftarrow true$;
            if $level[var] \geq topLevel$ then
                $paths \leftarrow paths + 1$;
            else
                $redundant \leftarrow true$;
                foreach $u \in reason[var]$ do
                    if $|u| \neq var \wedge \neg visited[|u|]$ then
                        $redundant \leftarrow false$;
                if $\neg redundant$ then
                    append $lit$ to $learnt$;
    find last visited literal $lit$ in $trail$ with $level \geq topLevel$,
     assign it to $resLit$;
    $visited[|resLit|] \leftarrow false$;
    $paths \leftarrow paths - 1$;
    $C \leftarrow reason[|resLit|]$;
until $paths = 0$;
$learnt[0] \leftarrow -resLit$;
```

done in combinational logic. Since we stored all the literals in the external memory, this can be a performance impact to the whole solving process. To mitigate this situation, we introduced the fine-grained pipeline based on delayed shift registers. The registers will be reset to zero, and unconditionally shift after reset.

The delayed registers works like the history version of data. For example, we note the delayed registers of retrieved literal as lit^{-1}, lit^{-2}, lit^{-3}, indicating that they store the data of lit 1, 2, and 3 cycles before. When issuing the read request, the registers are reset within same cycle. When lit^{-3} is not zero, it means it is the data retrieved 3 cycles ago, and henceforth a valid literal data. Similar procedure is applied to retrieving the decision level and visited mark data.

With this simple pipeline design, the efficiency of conflict analysis in VeriSAT is dramatically improved.

E. Variable Decision Engine

The Variable Decision Engine (VDE) is invoked after unit propagation when no conflicts are detected. It selects the next branching variable using the Variable State Independent Decay Sum (VSIDS) heuristic. The VDE maintains an activity score table for each variable, which is initialized to 0 and updated during conflict analysis. The decay factor of 0.9275 is calculated as $x - (x \gg 16)$, avoiding floating-point operations to minimize the hardware complexity.

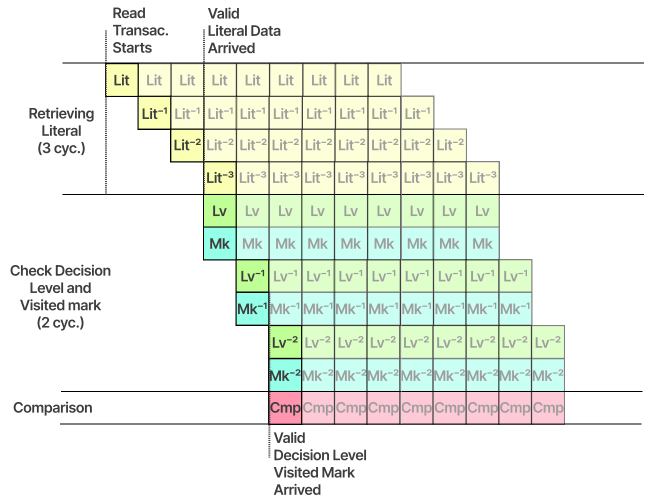
Read Transac. Starts
Valid Literal Data Arrived
Retrieving Literal (3 cyc.)
Lit
Lit⁻¹
Lit⁻²
Lit⁻³
Check Decision Level and Visited mark (2 cyc.)
Lv
Mk
Lv⁻¹
Mk⁻¹
Lv⁻²
Mk⁻²
Comparison
Cmp
Valid Decision Level Visited Mark Arrived

Fig. 4. The Pipeline optimization. The cells on horizontal direction indicates the cycles used

The VDE uses a min-heap to efficiently select the variable with the least activity score in O(log n) time. The maintenance of the min-heap is non-blocking; updates to the variable scores and decision-making logic will check if the heap is busy and wait until maintenance is completed.

Additionally, phase saving ensures the variable phase is stored before a restart and restored afterward, reducing redundant searches and improving solver efficiency.

V. Implementation of VeriSAT

We deployed VeriSAT on a Xilinx ZU9EG FPGA platform, which features a Xilinx XCZU9EG core and a Processing System (PS) that includes four Cortex-A53 processors operating at 1 GHz. The PS system also provides a DDR4 controller and four high-performance AXI4 slave ports connected to the Programmable Logic (PL), or FPGA fabric. The clock frequency for this configuration was set at 150 MHz, ensuring high throughput and efficient execution. The underlying Alinx AXU9EG platform provides four Micron MT40A512M16LY-062E DDR4 memory modules, with 2GB dedicated to the SAT solver. To communicate with DDR4 memory, we implemented an AXI4-Lite master IP that interfaces between the PL and the PS. We used Xilinx Vivado 2023.4 for synthesizing and implementing the design on the FPGA, and Verilator 5.020 for simulation and verification. The final design of VeriSAT can hold up to 16,384 variables and 1,048,576 literals in total. Table I shows the comparison of resource utilization between VeriSAT, SAT-Accel, and SAT-Hard.

In the input stage, the text-based DIMACS CNF file is first parsed in a driver program residing on the PS side. The clauses and literals are transmitted sequentially through the data port by the parsing process, while the address port serves as the control signal. Each literal is sequentially stored in the Literal Memory, and an additional control signal is used to indicate whether the current literal is the last one in a clause.

Finally, the registers containing the solve result, status and statistical information such as the cycle counter are exposed to the driver program for monitoring and control.

VI. Evaluation and Benchmark

A. General Benchmark

We evaluated the performance of VeriSAT on a set of diverse instances of SAT problems. The data sets used in this evaluation include instances from both the SATCOMP benchmark and SATLIB, which are widely recognized for benchmarking SAT solvers.

UF and **UUF** instances: These are generated from the SATLIB benchmark, where the UF instances are random 3-SAT problems and the UUF instances are similar but with a higher percentage of unsatisfiable instances.

QG, **BMC**, **HOLE**, **II**, and **LOGISTICS**: These instances also come from SATLIB, each presenting different problem structures and variable-to-clause ratios, which provide valuable insight into solver performance across a range of configurations.

PLANNING and **BATTLESHIP**: These are families of instances from the SATCOMP benchmark, obtained from the benchmark-database.de. They are designed to test the solver's ability to handle real-world problem structures, typically involving complex optimization problems.

The results of our experiments, shown in Table II, present the average number of cycles and the average time (in ms) to solve SAT instances using VeriSAT@150MHz on an FPGA, alongside MiniSat running on a high performance AMD EPYC 7542 CPU. The comparison highlights the performance of VeriSAT in terms of its cycle count and time per instance, as well as the speedup over MiniSat.

Notably, the instances from PLANNING and BATTLESHIP (from SATCOMP) demonstrate extreme speedups of 2.34x and 11.71x, respectively, highlighting VeriSAT's efficiency in tackling structured problem instances that might involve more complex heuristics and optimizations. For QG, BMC, and HOLE, which are larger and more challenging problem instances, VeriSAT achieves speedups of 1.12x, 9.41x, and 251.83x, respectively. These results suggest that the FPGA-based implementation scales well with larger problems, delivering a significant reduction in execution time compared to the CPU-based solver.

However, for the UF/UUF dataset as the number of variable grows, the average solving time of VeriSAT grows fast. This demonstrate some incapability of handling random instances, which can be further studied.

B. Solver Comparison

We also performed a comparison between the state-of-the-art (SOTA) stand-alone FPGA-based CDCL SAT solver, including SAT-Accel and SAT-Hard.

Table III presents a comparison of resource utilization and implementation level of the algorithm between VeriSAT, SAT-Hard, SAT-Accel, and MiniSat. and Table IV shows the runtime comparison based on the public benchmark instance and record. VeriSAT demonstrates significant speedups, achieving up to 1043.97x over SAT-Hard and 17.94x over SAT-Accel. Its FPGA-based design excels in structured problems, while SAT-Accel and SAT-Hard struggle, particularly with larger or more complex instances. But it should be noted that for the publicly accessible instances that are used by SAT-Accel, all from SATCOMP, are not able to be solved by VeriSAT, due to the limitation of hardware resource and randomness.

SAT-Hard is the first known stand-alone FPGA-based SAT solver that implements CDCL. However, due to its monolithic state machine design and the limited logic resources on the FPGA device, its solving ability is not ideal. SAT-Accel, on the other hand, is

TABLE I
COMPARISON OF FPGA RESOURCE UTILIZATION FOR SAT-ACCEL, VERISAT, AND SAT-HARD

2*Resource	SAT-Accel (XCU55C)			VeriSAT (XCZU9EG)			SAT-Hard (XC7Z020)		
	Used	Available	%	Used	Available	%	Used	Available	%
LUT	251,283	1,303,680	19.00%	15,770	274,080	5.75%	1,894	53,200	3.56%
FF	324,891	2,607,360	12.00%	9,175	548,160	1.67%	765	106,400	0.72%
DSP	48	9,024	0.50%	0	2,520	0.00%	0	0	0.00%
BRAM	419	2,016	21.00%	707	912	77.52%	109	140	77.86%
URAM	778	960	81.00%	0	0	0.00%	0	0	0.00%

TABLE II
COMPARISON OF SAT SOLVER PERFORMANCE ON DIFFERENT DATASETS

Dataset				VeriSAT@150MHz			MiniSat		Speedup
Dataset	#Var. Range	#Cl. Range	#Inst	Cycles	Time (ms)	TO	Mean Time (ms)	TO	
UF20	20 - 20	91 - 91	1000	2,803.15	0.02	0	0.6	28	**30.0**
UF50	50 - 50	218 - 218	1000	39,240.15	0.26	0	0.7	11	**2.69**
UF75	75 - 75	325 - 325	100	298,172.04	1.99	0	1.1	0	0.55
UF100	100 - 100	430 - 430	1000	1,712,141.66	11.41	0	1.9	0	0.17
UF125	125 - 125	538 - 538	100	16,593,148.63	110.62	0	4.3	0	0.04
UF150	150 - 150	645 - 645	100	107,043,051.36	713.62	18	9.1	0	0.01
UUF50	50 - 50	218 - 218	1000	91,667.29	0.61	0	0.8	0	**1.31**
UUF75	75 - 75	325 - 325	100	633,991.85	4.23	0	1.5	0	0.35
UUF100	100 - 100	430 - 430	999	5,110,190.56	34.07	0	3.2	0	0.09
UUF125	125 - 125	538 - 538	100	39,052,766.50	260.35	0	8.2	0	0.03
UUF150	150 - 150	645 - 645	100	143,538,806.50	956.93	0	21.8	0	0.02
QG	343 - 2197	9685 - 148957	22	27,193,919.44	181.29	1	202.2	0	**1.12**
BMC	2810 - 13215	11683 - 65025	13	2,709,938.33	18.07	0	170.7	0	**9.41**
HOLE	42 - 110	133 - 561	5	10,474,512.25	69.83	0	1010.5	0	**14.5**
II	66 - 1728	186 - 24792	41	1,594,420.02	10.63	0	18.0	0	**1.69**
LOGISTICS	828 - 4713	6718 - 21991	4	2,408,013.05	16.05	0	16.5	0	**1.03**
PLANNING	48 - 2958	153 - 28371	9	983,125.57	6.55	0	15.3	0	**2.34**
BATTLESHIP	28 - 1368	58 - 16308	7	348,537.25	2.32	0	27.16	0	**11.71**

the latest stand-alone FPGA-based SAT solver with Tier 1 FPGA hardware, demonstrating tremendous solving ability. However, since it uses HLS (High-Level Synthesis), the mapping of the algorithm to hardware is left to the EDA tool, making it challenging to track the correspondence between the hardware and the procedural code.

In summary, VeriSAT utilizes a custom FPGA architecture that directly maps the CDCL procedural components with a simple and reliable hardware design, which achieves the balance between performance and cost. It shows the potential of solving SAT problems on specific domain.

VII. CONCLUSION AND FUTURE WORK

In this work, we introduced VeriSAT, an FPGA-based SAT solver that takes advantage of a custom hardware-friendly memory scheme, concurrent propagation engine, and pipelined conflict analysis engine to significantly enhance performance. Through extensive benchmarking, we demonstrated that VeriSAT outperforms other FPGA-based CDCL SAT solvers, such as SAT-Hard and SAT-Accel, by factors of 1043.97x and 17.94x respectively. The results highlight the effectiveness of our design choices and optimizations, making VeriSAT a promising solution for high-performance SAT solving, particularly for well-structured, real-world problems with less randomness.

Looking ahead, there are several key areas for future improvement. First, due to the nature of the watching list, implementing efficient clause deletion in VeriSAT remains challenging. We aim to design a better method for handling clause deletion to overcome the efficiency drawbacks as the number of learned clauses grows. Second, the current design has long critical paths that limit the operating frequency. We will further modularize the design and optimize the data path to reduce these critical paths, thus increasing the clock frequency. Finally, while CDCL is an effective SAT solving algorithm, it is not fully optimized for FPGA architectures. We will continue to explore variants of the CDCL algorithm that can better exploit the fine-

TABLE III
COMPARISON OF SAT SOLVERS: VERISAT, SAT-HARD, SAT-ACCEL, MINISAT

Solver	VeriSAT	SAT-Hard	SAT-Accel	MiniSat
Model	XCZU9EG	XC7Z020	XCU55C	EPYC 7542
Freq.	150 MHz	Unknown	230 MHz	2.0 GHz
Impl.	System Verilog	Verilog	C++	C++
CDCL	✓	✓	✓	✓
2-lit Watching	✓		✓	✓
Phase Saving	✓		✓	✓
VSIDS	✓		✓	✓
Restart	Glucose		Luby	Luby
Clause Del.			✓	✓
Clause Min.			✓	✓

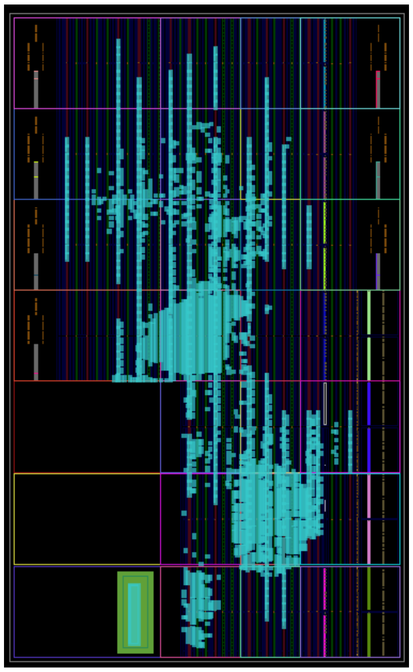

Fig. 5. The layout of implemented VeriSAT on Xilinx Zynq UltraScale+ ZU9EG. The Blue parts indicates the utilized hardware resource. Observed that most of the BRAM (the vertical bars) are utilized.

grained parallelism and pipelining capabilities of FPGA platforms to further enhance VeriSAT's performance.

TABLE IV
COMPARISON OF PROBLEM INSTANCES WITH VERISAT, SA, AND SH TIMES

Problem Name	Var	Cls	VS(ms)	SA(ms)	SH(ms)
hole7	56	204	**10.68**	125	330
hole8	72	297	**62.05**	691	2,270
hole9	90	415	**344.11**	N/A	15,290
uf100-010	100	430	0.15	1.00	580
uf125-01	125	538	181.06	4.00	1,160
uf150-08	150	645	19.85	1.00	3,920
uuf100-02	100	430	18.43	4.00	4,940
uuf125-05	125	538	717.54	7.00	4,900
CBS-k3-n100-m403-k3-10	100	403	0.57	2.00	2,340
aim-200-3-4-yes1-4	200	320	0.27	4.00	1,200
aim-200-3-4-no-yes1-4	200	320	3.89	0.30	10
ii16e2	222	1,186	10.61	4.00	5,760
ii32e1	222	1,186	1.05	0.10	20
battleship-6-9	54	171	**1.44**	514.00	-
bmc-ibm-1	9,685	55,870	134.41	54.00	-
bmc-ibm-2	2,810	11,683	1.94	1.00	-
bmc-ibm-5	9,396	41,207	21.08	8.00	-
bmc-ibm-7	8,710	39,774	86.37	8.00	-
qg3-08	512	10,469	2.47	1	-
qg6-10	1,000	43,956	**16.01**	31	-
qg6-12	1,728	69,311	183.22	135	-
qg7-10	1,000	43,756	**19.12**	29	-
qg7-12	1,728	70,327	**264.23**	292	-
			Speedup:	**17.94x**	**1043.97x**

REFERENCES

[1] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[2] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[3] P. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. "Combinational test generation using satisfiability". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.9 (1996), pp. 1167–1176. DOI: 10.1109/43.536723.

[4] Peixin Zhong et al. "Accelerating Boolean satisfiability with configurable hardware". In: Apr. 1998, pp. 186–195. DOI: 10.1109/FPGA.1998.707896. URL: https://ieeexplore.ieee.org/document/707896.

[5] O. Mencer and M. Plazner. "Dynamic circuit generation for Boolean satisfiability in an object-oriented design environment". en. In: Maui, HI, USA: IEEE Comput. Soc, 1999, p. 8. ISBN: 978-0-7695-0001-0. DOI: 10.1109/HICSS.1999.772883. URL: http://ieeexplore.ieee.org/document/772883/.

[6] Edmund Clarke et al. "Bounded Model Checking Using Satisfiability Solving". en. In: *Formal Methods in System Design* 19.1 (July 2001), pp. 7–34. ISSN: 1572-8102. DOI: 10.1023/A:1011276507260. URL: https://doi.org/10.1023/A:1011276507260 (visited on 09/22/2024).

[7] E.I. Goldberg, M.R. Prasad, and R.K. Brayton. "Using SAT for combinational equivalence checking". In: *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. 2001, pp. 114–121. DOI: 10.1109/DATE.2001.915010.

[8] Matthew W. Moskewicz et al. "Chaff: engineering an efficient SAT solver". In: DAC '01. New York, NY, USA: Association for Computing Machinery, June 22, 2001, pp. 530–535. ISBN: 978-1-58113-297-7. DOI: 10.1145/378239.379017. URL: https://doi.org/10.1145/378239.379017.

[9] T. Suyama et al. "Solving satisfiability problems using reconfigurable computing". en. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.1 (Feb. 2001), pp. 109–116. ISSN: 1063-8210, 1557-9999. DOI: 10.1109/92.920826.

[10] S. Chaki et al. "Modular verification of software components in C". In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 388–402. DOI: 10.1109/TSE.2004.22.

[11] I. Skliarova and A. De Brito Ferrari. "Reconfigurable hardware SAT solvers: a survey of systems". en. In: *IEEE Transactions on Computers* 53.11 (Nov. 2004), pp. 1449–1461. ISSN: 0018-9340. DOI: 10.1109/TC.2004.102.

[12] K. Kanazawa and T. Maruyama. "An FPGA solver for WSAT algorithms". In: *International Conference on Field Programmable Logic and Applications, 2005*. 2005, pp. 83–88. DOI: 10.1109/FPL.2005.1515703.

[13] Niklas Sörensson and Niklas Een. "Minisat v1.13-a SAT solver with conflict-clause minimization". In: *International*

Conference on Theory and Applications of Satisfiability Testing (Jan. 1, 2005).

[14] Mandar Waghmode et al. “An Efficient, Scalable Hardware Engine for Boolean SATisfiability”. In: ISSN: 1063-6404. Oct. 2006, pp. 326–331. DOI: 10.1109/ICCD.2006.4380836. URL: https://ieeexplore.ieee.org/document/4380836.

[15] Kenji Kanazawa and Tsutomu Maruyama. “An FPGA Solver for Very Large SAT Problems”. In: *2007 International Conference on Field Programmable Logic and Applications*. 2007, pp. 493–496. DOI: 10.1109/FPL.2007.4380697.

[16] Kanupriya Gulati et al. “FPGA-based hardware acceleration for Boolean satisfiability”. In: *ACM Trans. Des. Autom. Electron. Syst.* 14.2 (Apr. 2009). ISSN: 1084-4309. DOI: 10.1145/1497561.1497576. URL: https://doi.org/10.1145/1497561.1497576.

[17] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. “Inprocessing Rules”. In: *Automated Reasoning*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 355–370. ISBN: 978-3-642-31365-3.

[18] Jun Sakoh, Noriaki Yoshimasa, and Yoshinobu Kawabe. “Automated proof for equivalence of telephone systems”. In: *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*. 2013, pp. 497–502. DOI: 10.1109/ICIS.2013.6607888.

[19] Khadija Bousmar et al. “A new FPGA-based DPLL algorithm to improve SAT solvers”. In: *2015 27th International Conference on Microelectronics (ICM)*. 2015, pp. 287–290. DOI: 10.1109/ICM.2015.7438045.

[20] Ali Asgar Sohanghpurwala and Peter Athanas. “An effective probability distribution SAT solver on reconfigurable hardware”. In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016, pp. 1–6. DOI: 10.1109/ReConFig.2016.7857150.

[21] Ma Kefan et al. “An FPGA SAT solver based on enhanced constraint”. In: May 2017, pp. 25–30. DOI: 10.1109/FPGA4GPC.2017.8008962. URL: https://ieeexplore.ieee.org/document/8008962.

[22] Ludovic Le Frioux et al. “PaInleSS: A Framework for Parallel SAT Solving”. In: *Theory and Applications of Satisfiability Testing – SAT 2017*. Ed. by Serge Gaspers and Toby Walsh. Cham: Springer International Publishing, 2017, pp. 233–250. ISBN: 978-3-319-66263-3.

[23] Ali Asgar Sohanghpurwala, Mohamed W. Hassan, and Peter Athanas. “Hardware accelerated SAT solvers—A survey”. In: *Journal of Parallel and Distributed Computing* 106 (Aug. 1, 2017), pp. 170–184. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2016.12.014.

[24] Gilles Audemard and Laurent Simon. “On the Glucose SAT Solver”. In: *International Journal on Artificial Intelligence Tools* 27.01 (2018), p. 1840001.

[25] Aye Myint Myat, Khine Khine Htwe, and Nobuo Funabiki. “Fill-a-Pix Puzzle as a SAT Problem”. In: *2019 International Conference on Advanced Information Technologies (ICAIT)*. 2019, pp. 244–249. DOI: 10.1109/AITC.2019.8920898.

[26] Buse Ustaoglu et al. “SAT-Hard: A Learning-Based Hardware SAT-Solver”. In: Aug. 2019, pp. 74–81. DOI: 10.1109/DSD.2019.00021.

[27] Vinicius N. Possani et al. “Parallel Combinational Equivalence Checking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2020), pp. 3081–3092. DOI: 10.1109/TCAD.2019.2946254.

[28] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. “Preprocessing in SAT Solving”. In: *Handbook of Satisfiability*. 2021.

[29] Nils Froleyks et al. “SAT Competition 2020”. In: *Artificial Intelligence* 301 (2021), p. 103572. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2021.103572.

[30] Junhua Huang et al. “Accelerate SAT-Based ATPG via Preprocessing and New Conflict Management Heuristics”. In: *Proceedings of the 27th Asia and South Pacific Design Automation Conference*. ASPDAC ’22. Taipei, Taiwan: IEEE Press, 2022, pp. 365–370. ISBN: 9781665421355.

[31] Jinghui Jiang et al. “P4-DPLL: accelerating SAT solving using switching ASICs”. In: *Proceedings of the ACM SIGCOMM Workshop on Formal Foundations and Security of Programmable Network Infrastructures*. FFSPIN ’22. Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 24–30. ISBN: 9781450393294.

[32] Alexander Nadel. “Introducing Intel(R) SAT Solver”. In: *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Ed. by Kuldeep S. Meel and Ofer Strichman. Vol. 236. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 8:1–8:23. ISBN: 978-3-95977-242-6. DOI: 10.4230/LIPIcs.SAT.2022.8.

[33] Zhihan Chen et al. “Integrating Exact Simulation into Sweeping for Datapath Combinational Equivalence Checking”. In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, Oct. 2023, pp. 1–9. DOI: 10.1109/iccad57390.2023.10323876. URL: http://dx.doi.org/10.1109/ICCAD57390.2023.10323876.

[34] Dominik Schreiber and Peter Sanders. “MallobSat: Scalable SAT Solving by Clause Sharing”. In: *J. Artif. Int. Res.* 80 (Sept. 2024). ISSN: 1076-9757. DOI: 10.1613/jair.1.15827. URL: https://doi.org/10.1613/jair.1.15827.

[35] Cai Li-Sha and Lin Er-Min. “Design of Lychee Picking Robot Based on SAT Path Planning Algorithm and Fuzzy Obstacle Avoidance Strategy”. In: *2024 IEEE 4th International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA)*. Vol. 4. 2024, pp. 347–351. DOI: 10.1109/ICIBA62489.2024.10868311.

[36] Michael Lo, Mau-Chung Frank Chang, and Jason Cong. “SAT-Accel: A Modern SAT Solver on a FPGA”. In: *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’25. Monterey, CA, USA: Association for Computing Machinery, 2025, pp. 234–246. ISBN: 9798400713965. DOI: 10.1145/3706628.3708869. URL: https://doi.org/10.1145/3706628.3708869.