SAT Swarm: a massively parallel SAT solver

Tate Staples (jts98) and Shaan Yadav (ay140)

I. Abstract

THE Boolean Satisfiability (SAT) problem is foundational to numerous computational applications but notoriously computationally intensive due to its NP-complete nature. While software-based SAT solvers like MiniSAT have seen significant algorithmic advancements, they encounter performance bottlenecks as problem sizes scale. We introduce SAT Swarm, a massively parallel hardware accelerator specifically designed to exploit the inherent parallelism within SAT solving. By implementing a tiled grid architecture composed of independently operating nodes interconnected in various configurable topologies, SAT Swarm leverages both parallel branch exploration and SIMD-based clause evaluation to dramatically accelerate SAT solving. Our design features specialized hardware implementations of the Davis–Putnam–Logemann–Loveland (DPLL) algorithm, adapted to parallelize recursive searches and clause evaluation effectively. Utilizing a custom, cycle-accurate simulator and synthesis estimates targeting the Skywater 65nm process, we demonstrate significant performance improvements over state-of-the-art software solvers. Additionally, hardware estimations indicate scalability potential well-suited to modern integrated circuit technologies. SAT Swarm thus represents a promising advancement towards addressing large-scale industrial and computational SAT instances efficiently and effectively.

II. Overview and Motivation

A. Paper Outline

This paper was written as a project report for Duke's ECE 652: Advanced Computer Architecture II. It was designed to explore how hardware accelerators can exploit parallelism in problems not available to general purpose computers. In addition, it provided an exploration of the networking, design, simulation, and benchmarking challenges surveyed in the papers read in the class.

In the rest of Section II, we provide the motivation for the project and why we believe that accelerating SAT solving is a useful and interesting topic. In Section III, we fully describe what SAT solving is and discuss the current software solving solutions. In Section IV, we will describe our proposed solution to the problem. This starts with our observed parallelism opportunities followed by the networking and architectural tradeoffs of different designs to exploit that parallelism. Section V evaluates the effects of different parameters in our design. We use our own Rust-based cycle-accurate simulator to evaluate and compare performance to modern software solutions. Additionally in Section VI, we evaluate the size and power usage of our design by synthesizing a naive Verilog implementation. Section VII elaborates more on our methodology for developing the simulator and functionally validating it. We evaluate our results in Section VIII. We review related work in Section IX, conclude our paper in Section X and provide our acknowledgments in Section XI.

B. Goals as a Class Project

When deciding on a project for the class, we wanted to design an accelerator for some general computation task. Many areas, such as computer vision and machine learning, seemed to be areas that were already heavily saturated. We looked for inspiration in other projects we were working on, namely SAT solvers. SAT was chosen because the problem is both very general and very computationally hard (it is NP-complete). Furthermore, the core computation of SAT solvers exhibit few, regular data-dependencies, making it suitable for acceleration.

C. Industrial Use

SAT problems have found a wide array of applications in industrial use. First, they are frequently used for efficient resource allocation and scheduling problems. One example is from Tate's prior work; Duke hospital used SAT solvers to minimize operating room downtime and thereby allowing more surgeries to take place per week. This problem involved constraints on patient, surgeon, operating room, and recovery room availability and is impossible to efficiently allocate by hand. Electronic Design Automation (EDA) also leverages SAT solvers for optimizations involving Equivalence Checking, Test Pattern Synthesis, and FPGA compilation. Additionally, computer science leverage SAT solvers for functional programming (pattern matching), import dependency resolution, formal verification, and low-level register allocation.

Due to the exponential complexity growth of SAT problems, problems quickly become infeasible for conventional solvers. Industrial problems start to become infeasible above 10,000 clauses depending on the structure of the problem. [14]

III. Background

Satisfiability (SAT) problems are a class of problems about finding an assignment to function input that returns true. It has been shown to NP-complete, meaning it is functionally equivalent to a wide range of computationally hard to solve, but easy to check problems. SAT problems can come in many forms, but we specifically chose to solve 3SAT instances.

3SAT, as shown in Figure 1, is a logical statement in Conjunctive Normal Form (CNF); featuring AND'ed clauses with OR'ed terms in each clause. 3SAT also has the constraint that each clause has at most 3 terms. The solution to 3SAT is either a set of boolean values for each variable or a claim that no such assignment exists.

The simplest algorithm for this is to use a backtracking algorithm to try all of the different assignments and then

$$(x_1 \vee \neg x_2 \vee x_3) \quad (1)$$
$$\wedge (\neg x_1 \vee x_2 \vee \neg x_4) \quad (2)$$
$$\wedge (x_4 \vee \neg x_3 \vee x_5) \quad (3)$$

Fig. 1. Example 3SAT formula. With ∧/∨/¬ representing AND/OR/NOT

check if any of the clauses are all false. Since the clauses are all AND'ed together, any completely false clause makes that assignment not satisfiable. The algorithms recursively tries all assignments and either something works or you have exhaustively checked that any assignment will contain a contradiction.

Additionally, most solvers also include an optimization called unit propagation which says if you ever have a term with all false and one unassigned value (ie $(F \vee F \vee \neg x_3)$), then assign the final value must make the clause evaluate to **true** (ie $x_3 \leftarrow F$). This accelerates assignments by reducing the number of guessed assignments that you have to make.

Algorithm 1 Davis–Putnam–Logemann–Loveland (DPLL) Algorithm

Input F: CNF Formula, A: Variable Assignments
Output Satisfiable: bool

```
1: if contradiction(F, A) then
2:    return false
3: end if
4: if isSAT(F, A) then
5:    return true
6: end if
7: A ← UnitPropagations(F, A)
8: v ← NewVariable(F, A)
9: A' ← Assign(A, v, true)
10: Ā' ← Assign(A, v, false)
11: if DPLL(F, A') or DPLL(F, Ā') then
12:    return true
13: end if
14: return false =0
```

In Algorithm1, we show the full DPLL algorithm that forms the core parts of software based SAT solvers. It implements the algorithm and optimizations we have discussed so far with searching implemented through recursive calls.

IV. Proposed Solution

A. Adapting DPLL for Hardware

The DPLL algorithm, as outlined in Algorithm 1 requires some modification before being suitable for hardware acceleration. Modifications are necessary to reflect the static structure of circuits and exploit the available parallelism in custom circuits

Actual recursion is not possible in hardware due to a circuit no physically being able to contain itself. Instead recursion is implementable through a stack to track and revert changes. In Algorithm 2 this is stack S for *speculative* assignments. For any assignments we might want to undo, an update is pushed onto the stack corresponding to the recursive assignment from the software implementation. Returning up the recursive stack is done through some rollback procedure on the assignments popped off the stack. We capture the OR relation from line 8 in Algorithm 1 by both branches of the search being able to return satisfiable independently.

Algorithm 2 Hardware Compatible DPLL

Output satisfied: boolean

```
1: S := Stack()
2: v ← NewVariable(F, A)
3: A ← Assign(A, v, false)
4: S.push(v, false)
5: while S not empty and not satisfied do
6:    if contradiction(F, A) then
7:       v, a ← S.pop()
7:       A ← rollback(A, v)
8:       A ← Assign(A, v, ¬a)
9:    else
10:      A ← UnitPropagations(F, A)
11:      v ← NewVariable(F, A)
12:      A ← Assign(A, v, false)
13:      S.push(v, false)
14:      satisfied := isSAT(F, A)
15:   end if
16: end while=0
```

Additionally, representing the DPLL algorithm in hardware requires encoding data into binary. The CNF Formula (denoted F) was represented by a buffer holding an array of Clauses. Each clause has 3 Terms with two pieces of information each: the Term's current assignment state and a unique variable id (ie. x_{12}). The Term's assignment state is one of **True/False/Symbolic**; naively we save this with two bits. The unique variable id (meaning i in x_i) is some n-bit word reflecting a goal of 2^n max supported variables.

B. Parallelization Opportunities

Now that we have an implementation of DPLL suitable for hardware, we can examine what acceleration benefits we can gain from its parallelism.

1) Parallel Branch Searching: The DPLL algorithm works by searching the full assignment space, checking of any assignment works. Due to branching structure of the DPLL algorithm, we know that each branch will contain no redundant or missed work. This means that each branch can be processed independently from other branches without any synchronization requirements.

We use this feature of the problem to reduce the number of speculative assignments that a given core works. Since each branch is self-similar, we emulate recursion by giving one branch to another identical copy of the processor. By networking together a grid of identical cores, we implement a work-stealing protocol to parallelize branch searching to an arbitrary number of cores.

2) Parallel Clause Evaluation: Many of operations in the DPLL algorithms implement the same simple instruction on all

of the clauses in the CNF Formula. This pattern is amenable to the Single Instruction, Multiple Data (SIMD) structure of parallel processing.

Specifically contradiction(F, A) was implemented with De-Morgan's Law to get:

$$\mathit{UNSAT} := \bigvee_{\text{clause} \in F} \left(\bigwedge_{\text{term} \in \text{clause}} \text{term} = \textbf{false} \right)$$

This a small AND gate on each clause with a large OR gate between clauses that should be very easy to implement in hardware. Likewise *isSAT* can be implemented in the complementary manner.

Additionally, Unit-propagation is done by checking each clause for a isolated literal and adding it a saved register. This can be done through a priority encoder returning the first available unit-propagation.

Finally, the Term's assignment states can all be updated in parallel. The implementation of this will be discussed under the Look Up Table in the next section.

C. Components of a Node

The first component of the node is the Assignment History Buffer. This tracks the order that we assigned variables and which assignments were made speculatively. Like in Algorithm 2, it is a stack that is used to rollback assignments and undo speculative state. We also keep a small bit-mask to track which variables have yet to be assigned so that we can implement the *NewVariable* Function.

The *Assign* function is implemented by means of a Look Up Table (LUT) that maps a chosen assigned variable to an updated Term for each corresponding Term. For example, if we update x_7 to **true** then the LUT will transmit **True** to all x_7 terms, **False** to all $\neg x_7$ literals. Additionally, it coordinates with the Assignment History Buffer to **Reset** assignments at a deeper speculative depth as part of the rollback implementation. The rest of the terms receive **Ignore** and don't update.

Clauses are stored in some memory system and streamed into the SIMD processing pipelines discussed in the previous section. Any UNSAT detection interrupts the processing and starts rollback. All Unit-Propagation variables are added to a small buffer that is preferred for assignments.

Finally the node supports slow *Model Resolution* which is essentially probing the Term States to find the final assignments that make the formula SAT. It is an important usability feature to be able to use the satisfying variables.

D. Networking Requirements

Thus far we've only discussed single node operation. SAT Swarm's core feature is the tiled nature of its design. We reduce the number of speculative assignments by doing both branches in parallel nodes where possible. We arrange the nodes in an arbitrary topology and then whenever a neighboring node doesn't have its own work and we are about to make a speculative assignment, we send a fork message with the current state of our clause assignments. As long as our node-to-node bandwidth is higher than the rate at which we which we process our clauses, we can start the neighboring node's processing in constant time.

When a node receives a fork, it starts with an empty Assignment History Buffer and the bit-mask of what was previously assigned. It can derive the *NewVariable* from this data and take the complementary assignment as the previous core

In our project presentation we initially wanted to reduce the number of LUTs because they take up a lot of space ($\log_2$ Max Variables $\times$ Number of Terms). We implemented this by having a single LUT that networked to each core. This optimization added a lot of stalling for competing clause line access between cores and made several other parallelism optimizations problematic. We have since removed this networking overhead by giving a copy of the LUT to each node.

V. Microarchitectural Design

This section demonstrates the microarchitectural feasibility of the accelerator, building upon the logical feasibility established previously.

The accelerator is conceptualized as a tiled grid (Figure 2), in which each tile corresponds to a node. Each node is a self-contained unit capable of independently solving a SAT problem. Figure 3 illustrates the interface of an individual node, which enables tiling for various topologies. It is worth mentioning while only four connection ports are shown, the architecture supports scaling to an arbitrary number of ports to accommodate a variety of topologies.

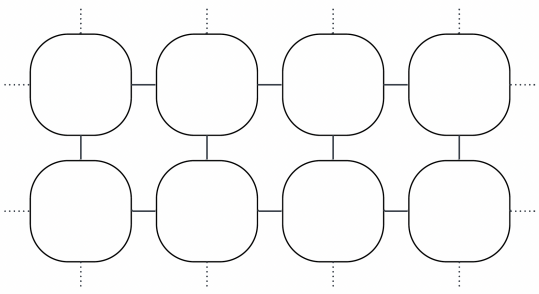

Fig. 2. The SAT Swarm accelerator comprises multiple nodes arranged in a user-defined topology. A segment of a grid-based layout is shown above for illustration.

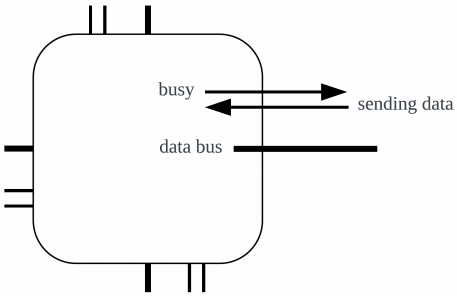

Fig. 3. Tile structure of a node in a torus topology. Each port includes: a busy signal (indicating if the node has work), a data-in signal (identifying the source of incoming data), and a data bus (carrying the SAT state forwarded from a previous node).

From a microarchitectural perspective, the design has to address three main problems: (i) Formatting and transmission of outgoing data during a fork (ii) Reception, decoding, and state reconstruction of incoming data (iii) Efficient updates to the data lookup table upon branching, without incurring significant performance penalties.

A. Sending Data

Various schemes for inter-node communication were considered, each involving trade-offs between transmission bandwidth and the preprocessing complexity required for data utilization. Based on mathematical estimations, we chose a method that balances performance and implementation simplicity (to avoid excessive hardware for decoding compressed messages). Alternative designs are discussed in the extensions and future work section.

In the implemented microarchitecture, a complete SAT state is transmitted from one node to another upon a fork. For instance, the clause:

$$(A_{\text{true}} \vee B_{\text{false}} \vee C_{\text{false}}) \wedge (A_{\text{true}} \vee D_{\text{symbolic}} \vee (\neg B)_{\text{true}})$$

may be encoded as the vector $[T, F, F, T, S, T]$, where T, F, and S (true, false, symbolic) are represented using 2 bits each.

In simulation, the system transmits SAT states at a rate of 100 clauses per cycle, corresponding to 600 bits per cycle. This is a naïve implementation, since each state is encoded in 2 bits, but only uses 3 states (this can be compressed in the future to 1.58 bit implementations). Each node is pipelined to concurrently process incoming data and forward new states at this throughput. Details of this pipeline implementation are discussed in the following subsection.

Although this level of data transfer may appear excessive, it significantly simplifies decoding logic at the receiving end. For example, assigning $A = 1$ typically requires searching for all occurrences of A in the formula—a costly operation in hardware. Transmitting the entire state eliminates the need for such lookups, enabling fast and predictable data path behavior. The pipelined nature of the nodes works well with this design choice, ensuring high throughput with minimal control complexity.

B. Receiving Data

An important requirement for the feasibility of this architecture is the ability to process 100 clauses per cycle. This means that the microarchitecture needs to be capable of evaluating all 100 clauses in parallel to determine whether each clause is SAT, UNSAT, or requires symbolic variable assignment. This architecture is depicted in Figure 4. Additionally, a persistent data structure needs to maintain the state of all variables, including their positions (i.e. where the variables A and B exist within the SAT problem) and current assignments (i.e. is A assigned as true, false or symbolic). This structure is referred to as the Lookup Table (LUT).

As shown in Figure 3, data from neighboring nodes arrive through multiple input wires, each passing through a multiplexer. The multiplexer selects the appropriate data input line based on the active sender port signal. This mechanism is topology-agnostic; arbitrary interconnect configurations are supported simply by adjusting the number of input ports.

Following the multiplexer, the data path is divided into two parallel components: one responsible for clause computation, and the other for memory management.

Clause evaluation is performed by a parallel array of logic units that assess whether each incoming clause is satisfied. The output of each clause evaluation unit is logically ANDed to determine the overall SAT/UNSAT result. The input to this evaluation logic depends on the node's mode: if the node is backtracking, it reads from local memory; otherwise, it directly streams input data received from an adjacent node.

The LUT maintains two essential pieces of information for each variable:

1. Location – the clause indices in which a variable appears
2. Assignment – whether the variable is currently assigned true, false or remains symbolic

Conceptually, the LUT is analogous to a cache: variable identifiers (represented as 8-bit identifiers) serve similarly to cache tags, and the associated logical values are similar to the stored data. This is particularly helpful for evaluating the LUT's physical characteristics.

Upon receiving data from another node, the architecture must perform 600 bits worth of updates per cycle—corresponding to 100 clauses. To support such throughput, the LUT is partitioned into four independent banks, allowing simultaneous writes to different segments. Each bank is responsible for updating 25 clauses per cycle, and each LUT line stores 25 clause entries. This enables 150 bits of data to be written per bank per cycle.

An address counter sequentially selects LUT lines for update, incrementing once per clock cycle to distribute the incoming data across the entire table. This banking scheme allows operation of the node at 600 bits per cycle, sustaining the clause processing throughput required by the system.

C. Variable Assignment

Variable assignment is initiated within the clause evaluation component of the microarchitecture. This process occurs either through retrieving the trivial unit propagation if available to selecting the first available unassigned variable as tracked by a bit-mask. By default, the symbolic variables is guessed to be false on first assignment. If possible, the true case is forked to a neighboring node. If all neighboring nodes are busy, the node transitions into backtracking mode and pushed the speculative assignment into a Assignment History Buffer (not shown in our figures).

The chosen variable's unique identifier is stored in the assign reg, and its new assignment is stored in the assign val latch. These registers store these values through an entire single pass of the problem, enabling the state update in memory.

Figure 5 shows the structure of each memory bank that supports variable assignment. Each bank is logically divided into two segments: a static region and a dynamic region.

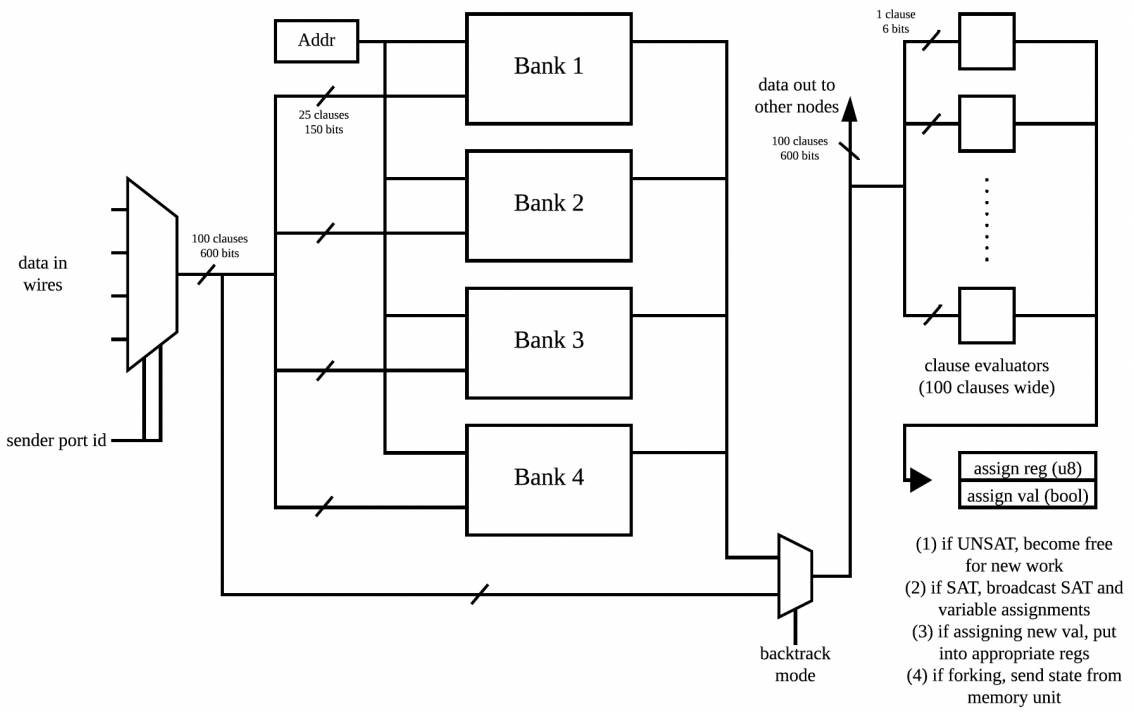

Fig. 4. Node microarchitecture. The data in wires represent the data buses through which other nodes can send data in.

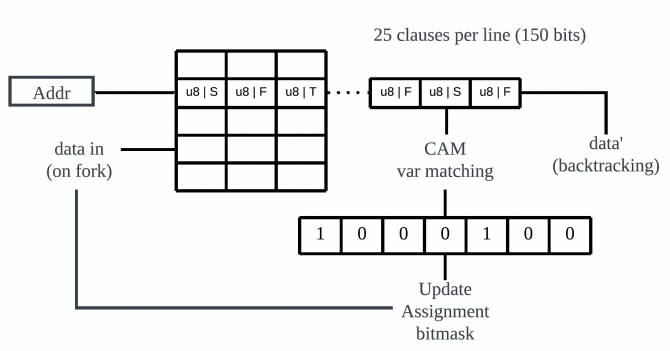

Fig. 5. Structure of each memory bank in a node. A variable's 2-bit assignment can be written to or read from the bank, while the 8-bit variable ID is matched using a content-addressable memory (CAM) structure.

The dynamic region stores variable assignment values, which are updated as the algorithm explores different assignments. The static region holds the 8-bit variable identifiers, which are fixed during problem initialization. The variable identifiers are needed in the architecture to map variable IDs to their location in the SAT problem.

To support efficient variable lookups by ID, the static region is implemented using a content-addressable memory (CAM) like structure. This structure looks up the variable IDs in a single 25 clause line and generates a bitmask indicating whether the variable being updated exists in that line and where.

As CAM matching incurs a one-cycle latency, a trailing write port is used to commit updated variable assignments to the address current addr - 1. For a problem with n clauses, the process of scanning memory, selecting the first symbolic variable, and committing the assignment requires $\frac{n}{100} + 1$ cycles.

This latency is deterministic and pipelined, allowing sustained throughput across cycles.

VI. Hardware Estimation

To ensure the simulator reflects realistic hardware constraints, we performed a preliminary estimation of both latency and area. These estimates helped guide key architectural parameters and avoiding optimistic or impractical performance assumptions.

A. Node Estimation

A baseline implementation of a single compute node was written in Verilog, using deliberately conservative logic and no architectural optimizations to reflect pessimistic, worst-case bounds.

The design was synthesized using Yosys, targeting the Skywater 65nm PDK [2], [5]. Post-synthesis, timing analysis was conducted with OpenSTA [3] to identify the maximum clock frequency where all critical paths met timing. An additional slack margin was included to account for process variation and environmental noise. Table I summarizes the resulting hardware characteristics.

Since we estimated a lower bound for a significantly outdated technology, we can estimate (using technology scaling trends and the fact that we had very pessimistic assumptions in our design) that our nodes would be able to support around a 1 GHz clock speed when implemented in modern technology.

TABLE I
HARDWARE ESTIMATION FOR ONE NODE (SKYWATER 65NM)

Metric	Value
Max Clock Frequency	634 MHz
Technology Node	65 nm (Skywater PDK)
Synthesis Tool	Yosys
Timing Tool	OpenSTA
Implementation Style	Unoptimized (pessimistic)

B. Lookup Table Estimation

The architecture's lookup tables (LUTs)—which store clauses and variable mappings—represent a significant portion of the design area but were not included in the node latency estimation. To characterize their area and power, we modeled the LUTs as caches and used the CACTI simulator [4] for these estimations.

Each LUT behaves like a cache with line widths of 150 bits (25 clauses), and a total of $\frac{n}{25}$ lines for n clauses. For representative 3-SAT instances with up to 100,000–200,000 clauses, this structure was scaled accordingly. The smallest available process in CACTI (22nm) was selected, using process parameters from the early 2000s. As a result, these estimates are pessimistic, and real-world implementations in modern nodes would likely show lower power and area overheads. Table II summarizes the resulting hardware characteristics.

TABLE II
LOOKUP TABLE ACCESS AND ENERGY ESTIMATES (CACTI, 22NM)

Parameter	Value
Access Time (ns)	0.464
Dynamic Read Energy (nJ/access)	0.0533
Dynamic Write Energy (nJ/access)	0.0560
Leakage Power (mW/bank)	8.57
Gate Leakage (mW/bank)	0.0248
Area (mm)	0.300 × 0.737

The results of the simulator show that a single node likely takes just over 0.2 mm^2 die area in a 22 nm process with tooling from around the 2000s. Intel's Raptor lake CPU has a die size of around 245 mm^2 [13] meaning that we can get up to 1225 cores on a die that size. We are very confident that even more nodes will be able to fit onto that size die with modern technologies, and furthermore, since this architecture requires no global cross-chip communication we can theoretically put this architecture on an arbitrarily large die and get arbitrarily more performance for larger and larger problems (more variables).

VII. METHODOLOGY

A. Cycle-Accurate Simulator

The cycle counts reported in the Evaluation Section were obtained using a custom cycle-accurate simulator written in Rust. This simulator implements the architecture described earlier, matching the microarchitectural latency and incorporating realistic network delays based on message size.

The hardware estimates previously calculated were used to parameterize the simulator such as the clock frequency and the number of nodes feasible within physical constraints.

A key limitation is scalability: simulating large problem instances is computationally expensive due to the cycle-level granularity and the fact that the simulator is single-threaded. As a result, the evaluation is limited to smaller SAT instances, even though the architecture is expected to scale favorably with larger inputs and more nodes.

B. Functional Validation

Simulator correctness was validated using benchmarks from SATLib [6], compared against outputs from MiniSAT [7], a well-established SAT solver.

Validation followed two stages: 1. Satisfiability Check: For each benchmark, we verified that our simulator produced the same classification (SAT or UNSAT) as MiniSAT. 2. Assignment Verification: For satisfiable cases, we extracted the variable assignments produced by the simulator and evaluated the original boolean formula under these assignments to confirm logical correctness.

This two-step process ensured both high-level and bit-level functional equivalence with a reference solver.

VIII. EVALUATION

Our evaluation primarily compares the performance of our architecture against state-of-the-art SAT solvers. We use MiniSAT [7] as a baseline due to its widespread adoption, extensive documentation, and high level of optimization.

Performance is defined as the real-world execution time (simulated as clock cycles) required to solve SAT instances. This metric is used to (i) compare our architecture against MiniSAT and (ii) evaluate how changes in architectural parameters—such as topology and node count—affect solver performance.

A. Effect of Topology

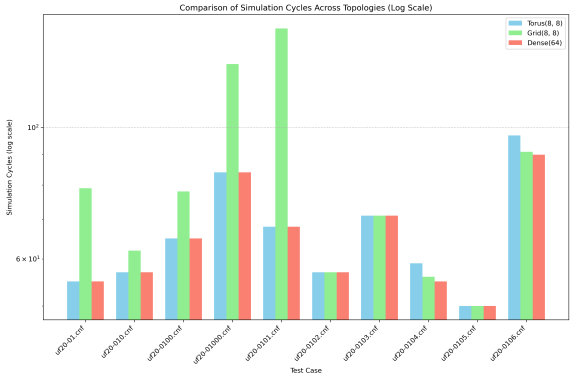

Fig. 6. Performance comparison across different topologies on the first 10 SATLib test cases. Y-axis: simulation cycles (log scale); X-axis: test cases with varying topologies.

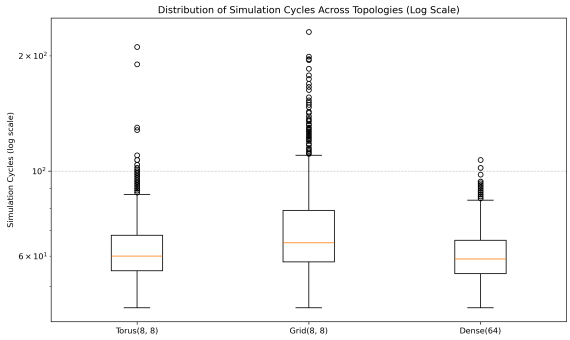

Fig. 7. Box plot of simulation cycles across all test cases for different topologies.

Figures 6 and 7 compare the performance of three network topologies: Dense (fully connected), Torus, and Grid (no wraparounds). The Dense topology outperforms the others, as expected, due to minimal communication latency and maximal parallel work distribution.

Interestingly, the Torus topology performs almost as well despite having significantly fewer interconnects. This is likely because the wraparound connections reduce communication bottlenecks at the edges—an issue that affects the Grid topology more severely by leaving edge nodes underutilized.

B. Effect of Node Count

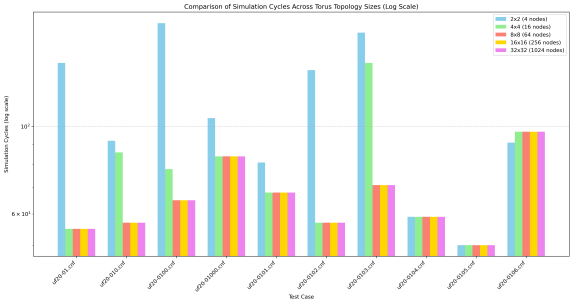

Fig. 8. Performance for varying numbers of nodes (4 to 1024) on a Torus topology for the first 10 SATLib test cases. Y-axis: simulation cycles (log scale).

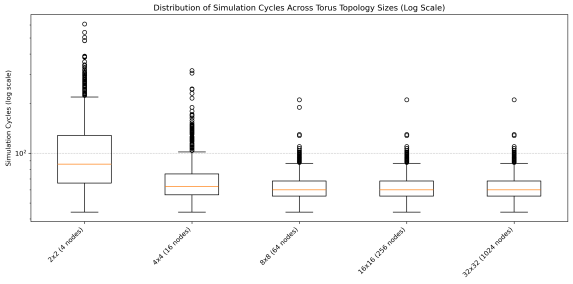

Fig. 9. Box plot of performance for varying node counts on a Torus topology across all test cases.

Figures 8 and 9 show the effect of scaling node count on performance within a Torus topology. We chose the Torus due to its near-optimal performance with lower overhead compared to Dense.

Initially, increasing the number of nodes leads to substantial performance improvements. However, beyond 64 nodes, we observe diminishing returns. This is likely because our SATLib benchmark only involves problems with 20 variables, limiting the amount of parallelism that can be effectively exploited. For larger, industrial-scale problems, we expect performance to continue scaling with the number of nodes.

C. Comparison with MiniSAT

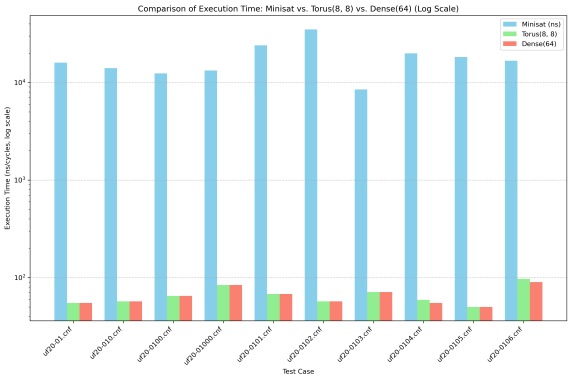

Fig. 10. Comparison between MiniSAT and our solver (Dense 64-node and Torus 64-node configurations) on the first 10 SATLib test cases. Y-axis: simulation cycles (log scale).

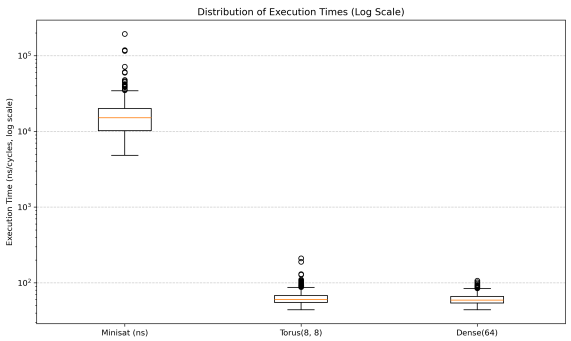

Fig. 11. Box plot comparing MiniSAT with the best configurations of our solver across all test cases.

Figures 10 and 11 compare MiniSAT with two high-performance configurations of our solver: a 64-node Dense topology and a 64-node Torus topology. Our architecture achieves up to 100x speedup over MiniSAT for 20-variable problems (50x with $\frac{1}{2}$ clock frequency).

It is worth noting that our simulation excludes the initialization overhead for populating lookup tables, which will likely take some constant number of cycles and can be highly parallelized.

IX. Related Work

A. Software-Based SAT Solvers

The evolution of SAT solvers has been marked by significant advancements in algorithmic strategies. The

Davis–Putnam–Logemann–Loveland (DPLL) algorithm laid the groundwork for systematic backtracking in SAT solving. Building upon this, Conflict-Driven Clause Learning (CDCL) [12] introduced mechanisms such as clause learning, non-chronological backtracking, and restarts, which have become standard in modern solvers like MiniSAT, Chaff, and Glucose. These enhancements have substantially improved solver efficiency, particularly in handling large and complex SAT instances.

Parallel SAT solving approaches have also been explored to leverage multi-core and distributed computing environments. Portfolio-based methods run multiple solver instances with varied heuristics concurrently, while divide-and-conquer strategies partition the problem space among processors. The Cube-and-Conquer paradigm exemplifies the latter, combining look-ahead techniques for problem partitioning with CDCL solvers for solving subproblems []

B. Hardware-Accelerated SAT Solvers

To overcome the limitations of software-based solvers, researchers have investigated hardware acceleration for SAT solving. SatIn is a notable example, presenting a hardware accelerator designed to enhance SAT solving performance by exploiting parallelism inherent in hardware architectures [8]. Similarly, SAT-Accel employs an algorithm-hardware co-design approach, integrating modern SAT solver techniques into FPGA implementations to achieve significant speedups. [9]

Other efforts have focused on optimizing specific components of SAT solvers. For instance, accelerating Boolean Constraint Propagation (BCP) on FPGAs has been shown to improve overall solver efficiency by eliminating costly clause look-up operations and leveraging fine-grained parallelism. These hardware-based approaches demonstrate the potential for substantial performance gains in SAT solving tasks. [10]

1) Comparison with SAT Swarm: While previous hardware-accelerated SAT solvers have achieved notable improvements, SAT Swarm distinguishes itself through its massively parallel architecture and emphasis on scalability. By implementing a grid of interconnected nodes, each capable of independent SAT solving and dynamic work distribution, SAT Swarm effectively balances the computational load and minimizes idle resources. This design facilitates efficient exploration of the solution space, particularly for large and complex SAT instances. [11]

Moreover, SAT Swarm's architecture supports seamless integration of advanced techniques such as clause learning and non-chronological backtracking, aligning with the capabilities of modern software-based solvers. This combination of hardware acceleration and sophisticated algorithmic strategies positions SAT Swarm as a promising solution for high-performance SAT solving applications.

C. Extensions and Future Work

There are a couple of further developments on this project we think would be interesting but either didn't have the time or knowledge of how to properly implement.

First, under the current model we are moving all of the clauses out of memory to the node core. For large problems with >100,000 clauses this is very network inefficient. There is a separate architecture where we split the clause table into stages and move our assignments along this pipeline. We started developing this in our simulator and found some interesting benefits. First, you can start to speculatively assign new variables before that previous assignment completes processing. Each assignment is assumed to work and new assignments pushed onto the checking pipeline. UNSATs then flush previous speculations and pop a variable number of speculative states off the stack. The second benefit is you have much lower networking as the size of your assignments tends to stay several orders of magnitude smaller than you number of clauses. Our two main holdups with this design was area overhead and the difficulty of efficient Assignment/Rollback.

The next project that could be interesting is augmenting our clauses. Right now our Clause Buffer is completely static, but CDCL allows for adding additional clauses as 'theorems' to catch common contradictions earlier. Additionally there is potentially interesting work on reordering the clauses based off how often they cause conflicts to reduce latency of contradiction detection. These weren't implemented due to time constraints and our focus on parallelism over single core performance.

Finally, there is the possibility that we could have reduced network overheads by allowing a sparse representation of our clauses. Often in software-based DPLL algorithms, they remove any Clauses with a **True** term because other updates won't affect it. Implementing this would require a more complicated memory system but could be interesting.

X. Conclusion

SAT Swarm presents a novel approach to accelerating Boolean Satisfiability (SAT) solving through a massively parallel hardware architecture. By reframing the Davis–Putnam–Logemann–Loveland (DPLL) algorithm for hardware implementation and leveraging configurable tiled-node topologies, the design achieves up to 100× speedup over state-of-the-art software solvers like MiniSAT for 20-variable problems. Key innovations include speculative state management through assignment rollback stacks, massively parallel clause evaluation pipelines, and decentralized work-stealing protocols that minimize communication overhead.

The architecture demonstrates scalability through cycle-accurate simulations and synthesis estimates, with node designs achieving around 1 GHz clock speed in 65nm technology and efficient area utilization. While current benchmarks focus on smaller SAT instances, the tiled structure shows promise for industrial-scale problems by enabling linear performance scaling with additional nodes. Future work could integrate conflict-driven clause learning (CDCL) techniques, optimize sparse clause representations, and explore different data movement patterns.

XI. Acknowledgments

We would like to thank Professor Brian Towles for his advice and for bearing with lots of our questions.

REFERENCES

[1] Austin, T. M. (1999). DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture* (pp. 196-207). Haifa, Israel. https://doi.org/10.1109/MICRO.1999.809458

[2] Clifford Wolf, Johann Glaser, and Johannes Kepler. "Yosys-A Free Verilog Synthesis Suite". In: 2013. URL: https://api.semanticscholar.org/CorpusID:202611483.

[3] Tutu Ajayi et al. "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain". In: 2019. URL: https://api.semanticscholar.org/CorpusID:210937106.

[4] Muralimanohar, Naveen & Balasubramonian, Rajeev & Jouppi, Norman. (2009). Cacti 6.0: A tool to model large caches. HP Laboratories.

[5] Google. (2020). SkyWater Open Source PDK. https://github.com/google/skywater-pdk

[6] Holger H. Hoos and Thomas Stützle: SATLIB: An Online Resource for Research on SAT. In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000, pp.283-292, IOS Press, 2000. SATLIB is available online at www.satlib.org.

[7] Sörensson, Niklas and Niklas Eén. "MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization." (2005).

[8] Chenzhuo Zhu and Alexander C. Rucker and Yawen Wang and William J. Dally. "SatIn: Hardware for Boolean Satisfiability Inference". 2023. https://arxiv.org/abs/2303.02588

[9] Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2025. 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). Association for Computing Machinery, New York, NY, USA, 234–246. https://doi.org/10.1145/3706628.3708869

[10] Accelerating Boolean Constraint Propagation for Efficient SAT-Solving on FPGAs, Hariprasadh Govindasamy and Babak Esfandiari and Paulo Garcia. 2024. https://arxiv.org/abs/2401.07429

[11] Ali Asgar Sohanghpurwala, Mohamed W. Hassan, Peter Athanas, Hardware accelerated SAT solvers—A survey, Journal of Parallel and Distributed Computing, Volume 106, 2017, Pages 170-184, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2016.12.014.

[12] Handbook of Satisfiability. Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsch. IOS Press. 2008.

[13] Aaron Klotz. 2022. Raptor Lake Die Size Confirmed: Larger vs. Alder Lake. Tom's Hardware. Retrieved from https://www.tomshardware.com/news/raptor-lake-die-size-confirmed-larger-vs-alder-lake

[14] Ansótegui C, Bonet ML, Levy J (2009) On the structure of industrial SAT instances. In: Proc. int'l conf.on principles and practice of constraint programming (CP), Springer, pp 127–141