CSEE 4840 BCP Accelerator Proposal

Christian Scaff (cts2148)
Columbia University
New York, New York, USA

1 Introduction

The boolean satisfiability problem (SAT) is a fundamental problem within computer science with applications in domains such as artificial intelligence, circuit design, and theorem proving. SAT was the first problem proven to be NP-complete [2], establishing it as a canonical benchmark for computational hardness. Despite this, modern SAT solvers have made significant progress in solving large instances efficiently. In particular, the Conflict-Driven Clause Learning (CDCL) algorithm has been a cornerstone of SAT solver design, enabling solvers to learn from conflicts and prune the search space effectively [5].

CDCL algorithms tend to spend a majority of their time performing Boolean Constraint Propagation (BCP), which is the process of deducing variable assignments based on the current partial assignment and the clauses in the formula. This process is computationally intensive and often is a source of bottlenecks in SAT solvers. As a result, numerous papers have proposed hardware accelerators for BCP, leveraging hardware parallelism to speed up this critical component of SAT solving [6]. For the CSEE 4840 capstone project, we propose to design and implement a toy hardware accelerator for BCP using the DE1-SoC FPGA platform.

2 Background

2.1 Boolean Satisfiability

Boolean satisfiability involves finding a satisfying assignment for a given formula in conjunctive normal form (CNF). Take for example the formula $(a \vee \neg b) \wedge (\neg a \vee c)$, which is satisfiable with the assignment $a = \text{true}, b = \text{false}, c = \text{true}$.

2.2 CDCL Algorithm

Modern SAT solvers approach this through the CDCL algorithm which is an extension of the Davis-Putnam- Logemann-Loveland (DPLL) algorithm with conflict-driven clause learning. It functions through iteratively making decisions on variable assignments and performing BCP where unit clauses are identified and propagated. When a conflict is detected, the solver analyzes the conflict to learn a new clause that prevents similar conflicts from occurring again. The solver then backtracks to an earlier decision level and continues the search, utilizing the learned clause to prune the search space. This process continues until either a satisfying assignment is found or the solver determines that the formula is unsatisfiable. Algorithm 1 gives a high-level pseudocode description.

Algorithm 2 denotes a variation of unit propagation. Within Algorithm 2, we denote a trail of literals that have been assigned but not yet processed for propagation. The algorithm iterates through the trail, checking clauses that are watching the negation of the current literal. Assuming a two-watched-literal scheme, if the other watched literal in the clause is true, we can skip this clause. Otherwise, we attempt to find a non-false replacement literal in the clause to watch. If we are unable to discover a replacement, we are either in a conflict (the clause is falsified) or we have a unit implication (the clause has only one unassigned literal left, which must be true to satisfy the clause).

Algorithm 1: CDCL(ϕ) [3]

```
Input  : CNF formula φ
Output : sat with assignment τ, or unsat
τ ← ∅
while true do
    τ ← UnitPropagate(φ, τ)
    if τ falsifies a clause in φ then
        if decision level = 0 then return unsat
        C ← AnalyzeConflict(φ, τ)
        φ ← φ ∧ C
        backjump to earlier decision level implied by C
    else
        if all variables assigned then return sat
        start a new decision level
        choose undefined literal l
        τ ← τ ∪ {l}
```

Algorithm 2: UnitPropagate(ϕ, τ)

```
Input  : CNF formula φ, partial assignment τ
Output : Conflict clause index, or −1
while trail has unprocessed literals do
    l̂ ← ¬ (next literal from trail)
    foreach clause c watching l̂ do
        if other watched literal of c is true then keep watch, skip
        if non-false replacement literal found in c then
            update watches to new literal
        else
            if c is conflicting then return index of c
            enqueue unit implication from c
return −1
```

3 Proposed Design

We aim to accelerate the BCP component of the CDCL algorithm by designing a hardware accelerator that can perform unit propagation in parallel across multiple clauses. As of now, we only have a preliminary design for a pipelined version of the BCP component. The accelerator will be implemented on the DE1-SoC FPGA platform

with a simple hardware interface that takes in a negated literal from the trail and outputs either a conflict or an implied variable with its assigned value as well as the index of the clause that caused the implication.

3.1 System Architecture

At a high level, the system will consist of four core submodules with three memory systems.

3.1.1 Memory Modules. The system will contain a clause database, implemented in on-chip memory, that stores all the clauses in the formula with a bit denoting clause satisfaction. Additionally, a watch list memory located on-chip will store the list of clauses watching each literal. To avoid complications related to dynamic memory patterns in CDCL, we will implement a fixed-size occurrence list for each literal. The system will also contain memory containing the current variable assignments that contain variable indices and their current assignments.

3.1.2 Core Modules. The system will function as a four stage pipeline.

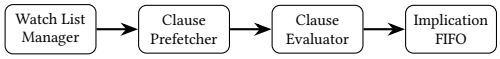

Figure 1: Four-stage BCP accelerator pipeline.

The first stage fetches and streams clause IDs for the given negated literal, reading from the watch list memory. The second state pipelines clause database reads, allowing us to fetch a clause while the previous clause is evaluated. This particular optimization is inspired by the unsatisfied clause prefetching design proposed by Choi and Kim [1] (§III-C). The core of the BCP evaluation is performed in the clause evaluator submodule which scans a clause, determining if the clause is satisfied, conflicting, or unit. If the clause is satisfied, we can skip it. If the clause is conflicting, we can output a conflict. If the clause is unit, we can output the implied variable and its assignment to the implication FIFO. This particular step can be parallelized. For the sake of feasibility in a semester-long project, we only make a promise of the singular evaluator design, but hope to explore parallelization if time permits. This likely requires a more complex design with a submodule to manage conflicts and implications from multiple parallel evaluators. Our final submodule is the implication FIFO which serves as a buffer for the output implications, adding some slack to the system to prevent stalling. The buffer takes inspiration from the pipelined implication queue described in Lo et al. [4] (§IV-B).

3.2 Interface and Integration

We will implement a very basic bare metal implementation of the CDCL algorithm to be ran on the DE1-SoC's ARM processor. The BCP accelerator will be implemented on the FPGA fabric and will be interfaced via the AXI bus. The processor will send negated literals to the accelerator and will receive implications and conflicts in return. The DE1-SOC itself will also require a parsing system to read in CNF formulas and initialize the clause database and watch list memories.

3.2.1 Likely Issues. The main bottleneck of the design will likely by the memory bound nature of BCP. Because this is a toy design, we will support smaller solving formulas that can fit on-chip. We likely will also run into issues with memory synchronization. Because the accelerator requires strict read-write times to pipeline, we necessitate the proposed PLMs (Watch List, Clause Database, Variable Assignment) rather than shared memory that both the CPU and accelerator can read-write to. This requires careful synchronization as the CPU and accelerator both modify variable assignments and clause memory.

4 Implementation Plan

We will first develop the hardware design in SystemVerilog. We will implement the three memory systems and follow them by implementing the four core submodules. For each submodule, including the memory systems, we plan to create testbenches to verify correctness. After the completion of the hardware design, we will implement a very basic bare metal CDCL implementation in C to run on the ARM processor. We intend to create an end-end test suite to verify the correctness of the solver itself as well as the software-hardware integration, which we expect to be the most difficult part of the project. Then we will implement a simple parser for CNF formulas to initialize the system.

5 Expected Results

Because of the nature of this project as a toy design, we do not expect great speed-ups. Realistically, depending on the clock speed of the ARM processor and the memory limitations, we may not even be able to outperform a software implementation on a standard CPU. However, we hope to illustrate the performance of a BCP accelerator design against simple SAT benchmarks that can fit on-chip, and showcase an interesting exploration of hardware design for SAT solving.

6 Notes to TAs

I wrote this proposal as an individual. I am still attempting to find a project group, but have been unsuccessful in finding one. Prof. Edwards told me to write the proposal as an individual for now, while I wait to find a group. Although, I am open to attempting this myself.

References

[1] Young-Kyu Choi and Changsoo Kim. 2024. FYalSAT: High-Throughput Stochastic Local Search K-SAT Solver on FPGA. *IEEE Access* 12 (2024), 65503–65519. doi:10.1109/ACCESS.2024.3397130

[2] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71)*. ACM, 151–158. doi:10.1145/800157.805047

[3] Tommi Junttila. 2020. Conflict-Driven Clause Learning (CDCL) SAT Solvers. Lecture notes, CS-E3220: Propositional Satisfiability and SAT Solvers, Aalto University. https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cdcl.html

[4] 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)*. ACM, Monterey, CA, USA. doi:10.1145/3706628.3708869

[5] João P. Marques-Silva and Karem A. Sakallah. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Comput.* 48, 5 (1999), 506–521. doi:10.1109/12.769433

[6] Soowang Park, Jae-Won Nam, and Sandeep K. Gupta. 2021. HW-BCP: A Custom Hardware Accelerator for SAT Suitable for Single Chip Implementation for Large

Benchmarks. In *Proc. 26th Asia and South Pacific Design Automation Conf. (ASP-DAC '21)*. ACM, 29–34. doi:10.1145/3394885.3431538