Accelerating Boolean Constraint Propagation for Efficient SAT-Solving on FPGAs

Hari Govindasamy
Carleton University
Ottawa, Canada
hari@sce.carleton.ca

Babak Esfandiari
Carleton University
Ottawa, Canada
babak@sce.carleton.ca

Paulo Garcia
Chulalongkorn University
Bangkok, Thailand
paulo.g@chula.ac.th

ABSTRACT

We present a hardware-accelerated SAT solver targeting processor/Field Programmable Gate Arrays (FPGA) SoCs. Our solution accelerates the most expensive subroutine of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, Boolean Constraint Propagation (BCP) through fine-grained FPGA parallelism. Unlike prior state-of-the-art solutions, our solver eliminates costly clause look-up operations by assigning clauses directly to clause processors on the FPGA and dividing large formulas into smaller partitions manageable by FPGA. Partitions are hot-swapped during runtime as required and the supported formula size is limited only by available external memory, not on-chip FPGA memory.

We evaluate our solver on a Xilinx Zynq platform with results showing quicker execution time across various formula sizes, subject to formula partitioning strategy. Compared to prior state-of-the-art, we achieve 1.7x and 1.1x speed up on BCP for 2 representative benchmarks and up to 6x total speedup over software-only implementation.

CCS CONCEPTS

• **Computer systems organization** → **Robotic autonomy**; • **Networks** → **Cyber-physical networks**; • **Applied computing** → **Industry and manufacturing**; • **Hardware** → **Hardware accelerators**; **Application specific processors**; • **Theory of computation** → **Equational logic and rewriting**.

KEYWORDS

FPGA, SAT, acceleration, embedded, boolean, satisfiability

1 INTRODUCTION

The Boolean Satisfiability problem (SAT) is a fundamental problem in computer science, the first NP-Complete problem [7]. SAT solvers have become the backbone of several engineering domains, as any NP-Complete problem can be encoded as instance of SAT [2, 7]. SAT solvers determine whether a given boolean formula is satisfiable by identifying an assignment to the formulas' free variables that evaluate the formula to true. The formula is unsatisfiable otherwise. Most SAT solvers target CNF-SAT, a subset of SAT that determines the satisfiability of formulas encoded in *Conjunctive Normal Form* (CNF) . Formulas in CNF are conjunctions of clauses, where each clause is a disjunction of one or more literals (a variable or its negation).

With the advent of modern Systems-on-Chip (SoC) comprised of both hard embedded processors and configurable FPGA fabric offering myriad implementation opportunities [17], deployed from the embedded to the high performance computing domain [1], accelerating SAT-solving through hardware is an attractive approach. We present a novel architecture for hardware-accelerated SAT-solving that outperforms state of the art solutions, released in open-source form for the Xilinx Zynq platform. Specifically, this article offers the following contributions:

- We describe a methodology to map and runtime-manage clauses across a processor and connected FPGA, making efficient use of FPGA resources and avoiding recurring performance pitfalls.
- We describe the implementation of an open-source prototype system, deployed on a Xilinx Zynq chip, identifying how the hardware architecture effects the aforementioned strategy.
- We evaluate our design against the state of the art using two representative benchmarks, showing speed-ups of 1.7x and 1.1x, respective, and overall a 6x improvement over vanilla software execution.

Section 2 describes necessary background knowledge on a particular SAT-solving algorithm required to understand the remainder of this paper. Section 3 presents an overview of historical solutions and state of the art, directly compared against in this paper. Section 4 presents our contribution, evaluated in Section 5, with concluding remarks and suggestions for future work described in Section 6.

2 BACKGROUND: DPLL AND BCP

SAT solvers are categorized into complete and incomplete solvers[15]. Complete solvers evaluate every possible variable assignment, ending on the first satisfying assignment or after exhausting the search space. A formula is unsatisfiable if the complete solver concludes without finding a satisfying assignment. Most incomplete solvers use Stochastic Local Search (SLS) to greedily search for a satisfying assignment in the formula's variable assignment search space[16]. While typically quicker than complete solvers, incomplete solvers do not guarantee results as they tend to get stuck in local maxima or skip satisfying assignments. Since they don't explore the solution
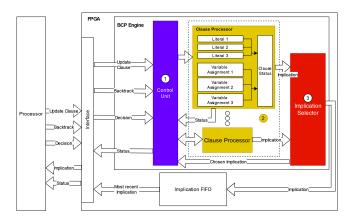
Figure 1: Interface between processor and FPGA-based BCP Coprocessor. DPLL's BCP is accelerated through fine-grained parallelization across Clause Processors.

space exhaustively they can never conclude that a formula is unsatis-fiable. Davis-Putnam-Logemann-Loveland (DPLL) and DPLL-based algorithms are the most predominant complete solvers [16]. DPLL performs two primary operations: 1) *decision* and 2) *Boolean Constraint Propagation* (BCP). DPLL-based algorithms follow DPLL's core structure, and propose improved decision heuristics, learning and BCP mechanisms. During decision, DPLL heuristically picks and assigns truth values to free variables. BCP subsequently propagates the effect of the decision using the *unit implication rule*[5]. The unit implication rule identifies unit clauses where all but one of its literals are false. Unit clauses can only be satisfied by assigning the variable to true if the literal is positive or to false on negative literals. The resulting assignment is known as an implication. BCP repeatedly applies this rule until all clauses are satisfied (formula is therefore satisfiable) or at least one clause evaluates false (conflict). On conflicts, DPLL backtracks by retracting and/or inverting assignments from earlier decisions. BCP is expensive, accounting for 80-90% of DPLL's CPU time, rendering it a prime candidate for hardware acceleration [5, 18]. BCP coprocessors accelerate DPLL by implementing specialized BCP processing engines on FPGA. These run alongside a General Purpose Processor (GPP) that performs the remaining DPLL operations: decision heuristics and backtracking. Using this architecture, the BCP coprocessor is first configured with the clauses, and then waits to evaluate decisions from the GPP. Any DPLL-based software solver can integrate with a BCP-coprocessor by replacing software BCP with the hardware accelerated BCP-coprocessor [18, 19]. FPGA-based BCP coprocessors are either instance-specific or application-specific. Instance-specific solver are built to solve a single SAT instance and designed by translating an input formula into its equivalent logical circuit. However, to solve new instances, the FPGA requires a complete rebuild (synthesis and FPGA programming may take up to several hours). Although these solvers can be significantly quicker than their software counterparts, their performance becomes notably slower when build times are included. For instance, Ivan et al's best result against the *hole7* benchmark achieves a 6.66x speedup against MiniSAT[11, 12]; however, when build times are included,

compilation alone takes 50 seconds, whereas MiniSAT finishes in under 0.064 seconds [10]. Application-specific solvers eliminate the need to rebuild the FPGA by instantiating general-purpose processing units capable of tackling any SAT instance (given that it fits in hardware). The BCP coprocessor is configured with the target problem by simply overwriting FPGA memory.

3 STATE OF THE ART

Algorithmic techniques for efficient SAT solving have been extensively researched, and the literature contains several surveys that describe the history and state of the art of the problem ([8], [13]). Techniques aimed at accelerating the execution of a particular SAT solving algorithm include software parallelization [9], deployment on specialized GPUs [14], and even acceleration through machine-learning approaches [20].

Our approach sits within FPGA-based acceleration, which began roughly 3 decades ago [6], with a few prominent results at the turn of the century ([21], [3]). However, it was not until significant advances in FPGA performance occurred in the last decade, and the rise of SoC platforms combining FPGA fabric with hard processors, that FPGA-based SAT acceleration matured. The most notable architectures were proposed by Davis et al [5] and Thong et al [18, 19]: both exploring the use of FPGA to implement BCP coprocessors, keeping the remainder of DPLL in software.

Davis et al calculate implications in parallel by using several inference engines (IE), each assigned a list of clauses (partitions) [5]. For every decision/implication, the clause containing the assignment variable is first retrieved before calculating implications. Implications are forwarded to a conflict detector that ensures that two or more IEs have not implied opposing values for the same variable. Implications are then sent to the processor and queued up for propagation.

To keep clause retrieval time low, a variable only occurs once in each IEs partition (i.e clauses within the same IE share no common variables). This limits the number of clauses affected by a decision to one, thereby also limiting implications per IE to one, constraining the effected performance. While some strategies to increase this limit have been proposed [4], they remain unexplored.

Thong et al. propose a concurrent BCP coprocessor comprising multiple sequential processing engines (PE) [19]. Identifying that Davis et al.'s clause lookup is slower than direct access [18], they develop a clause storage and encoding scheme that efficiently links clauses with shared variables. The processor sends decisions to the FPGA and starts BCP execution at a single PE. Using the linked list, the PE traverses every clause containing the decision variable and calculates implications, which are then added to a local queue and propagated. The running PE triggers BCP execution in another PE when it arrives at a link to a clause that is located elsewhere. The coprocessor supports multithreaded software execution, hiding communication and software latency by keeping the coprocessor busy while software threads make decisions when possible.

Davis et al. and Thong et al. have laid a strong foundation in developing application-specific FPGA-based BCP coprocessors; we extend their work and propose a solution that processes clauses in parallel without the need for clause lookup.

4 THE SAT SOLVER ARCHITECTURE

We present a BCP coprocessor that works alongside vanilla DPLL (and should, in theory, work seamlessly with any DPLL-based solver). Like Thong et al., we forgo clause lookup and allow clauses to share variables within the same partition. However, we still achieve Davis et al.'s high degree of parallelism by placing clauses directly in clause processors (explained in Section 4.1).

SAT instances larger than the available number of Clause Processors (CPs) are partitioned, stored in external memory (i.e., software) and hot-swapped into the BCP coprocessors as required during runtime. Solvable instance size is limited only by the GPP's RAM, not on-chip FPGA memory. We deploy our solution on the Zynq chip, and available here[1] for use. To our knowledge, this is the first open-source hardware-accelerated SAT solver.

4.1 The BCP accelerator architecture

Figure 1 illustrates our approach, comprising a GPP and an FPGA accelerated BCP coprocessor. The GPP executes DPLL's remaining elements (decisions, backtrack, etc.), partitions large SAT instances (explained in Section 4.2) and swaps partitions into hardware as required. Its default state is idle, awaiting instructions to execute. Once a decision is received, the systems loops until all unit clauses are exhausted. The BCP coprocessor, depicted in Figure 1, comprises a control unit (1), an array of clause processors (2) and an implication selector (3). The central control unit communicates directly with the GPP and each CP. Based on the received GPP command, it loads clauses into CPs, broadcasts decisions, or clears assignments during backtrack. At its core, the BCP coprocessor consists of an array of CPs that calculate decision and implication results in parallel. CPs store clauses as an array of literals maintain a local copy of each literal's respective variable assignment. Partitions are hot-swapped into FPGA by overwriting a CPs array of literals with the literals of the new clause. Variable assignments are updated during decisions and BCP, and cleared during backtrack. Finally, the implication selector chooses a single implication to propagate when multiple implications arise as a result of BCP. Rather than using an explicit implication conflict detector, as done by Davis et al [5], we propagate the chosen implication, and identify conflicts during evaluation.

4.2 Formulae partitioning

SAT instances contain an arbitrary number of variables and clauses. The problem size solvable on FPGA is limited by its available Configurable Logic Block (CLB) and memory, and requires large problems be partitioned into smaller manageable sizes. Partitions are stored in the GPP, and swapped into FPGA during run time by overwriting CPs clauses. BCP is performed individually on each partition, and implications are relayed back to the GPP. Implications are subsequently propagated to other partitions. We aim to make partitions as large as possible, limited by the coprocessor's clause and variable threshold. Consider Equation 1, composed of four clauses, and an instance of our coprocessor that supports two clauses and three variables. Equation 2 and 3 outline the two possible ways to partition Equation 1. Equation 2 describes a scenario where the partitions

[1]https://github.com/harigovind1998/FPGA_BCP_acceleration
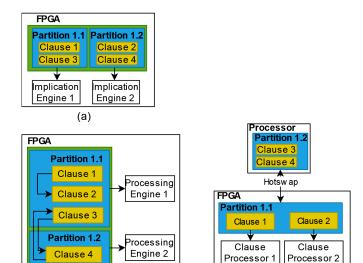

Figure 2: (a) Davis et al. store the formula directly on FPGA. Clauses within partitions contain no shared variables, and partitions are mapped directly to Implication Engines. (b) Thong et al. store formula directly on FPGA. Clauses are linked to other clauses with shared variables and are processed sequentially. (c) Formula stored in external memory ("software" view). Clauses in partitions mapped directly to Clause Processors, and hot-swapped as required.

reach the clause limit, while the Equation 3 reaches the variable limit.

$$f = (\neg a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg d \vee e \vee f) \wedge (d \vee e \vee f) \quad (1)$$

$$variation\_1 = \{\{(\neg a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)\},$$
$$\{(\neg d \vee e \vee f) \wedge (d \vee e \vee f)\}\} \quad (2)$$

$$variation\_2 = \{\{(\neg a \vee b \vee \neg c)\}, \{(a \vee \neg b \vee \neg c)\},$$
$$\{(\neg d \vee e \vee f)\}, \{(d \vee e \vee f)\}\} \quad (3)$$

Results (refer to Section 5) indicate that partitioning is a bottleneck in our approach. Performance improvement is dictated by the amount of required partition swapping and the number of unused CPs (occurs when the number of clauses in a partition is less than the available number of CPs). Thus, performance improvement is observed with certain partition assignments, while others lead to performance degradation. System performance can be improved by developing a more effective partitioning algorithm, but beyond the scope of this paper and reserved for future work.

4.3 Execution

Each clause processor is only associated with a single clause; thus, no clause look-up or traversal is required to retrieve the affected clause for processing. All clauses on the FPGA are processed in parallel as soon as a decision is received. Consider Equation 1's

Step		0	1	2	3	4
Our Approach	CP 1	Rx Decision	Process Clause 1	Done		
	CP 2	Rx Decision	Process Clause 2	Done		
Davis et al.	IE 1	Rx Decision	Retrieve Clause 1	Process Clause 1	Done	
	IE 2	Rx Decision	Retrieve Clause 2	Process Clause 2	Done	
Thong et al.	PE 1	Rx Decision	Process Clause 1	Traverse to Clause 2	Process Clause 2	Done
	PE 2	Idle	Idle	Idle	Idle	Idle

Figure 3: Execution steps of each described approach.

mapping of partitions to hardware as presented in Figure 2. Figure 3 summarizes the execution stages for Davis et al.'s, Thong et al.'s and our approach for the theoretical execution for a decision of variable a. In our approach, clauses 1 and 2 are processed by Clause Processor 1 and 2 in parallel once the decision is received. Since clauses 3 and 4 do not contain variable a, Partition 2 remains in external memory and is not processed. Though Davis et al. also process clause 1 and 2 in parallel, each Implication Engine first performs a clause look-up to retrieve the affected clause. Results of the decision on the affected clause are then calculated. Thong et al.'s approach starts BCP on Processing Engine 1. After clause 1 is processed Processing Engine 1 traverses to clause 2. In the manner, clauses in a partition are processed sequentially. Execution concludes after computing Partitions 1's final element, clause 2. Processing Engine 2 remains idle for the entire duration as clauses in partition 2 do not contain variable a.

4.4 Processor-FPGA interface

The BCP coprocessor implements the Advanced eXtensible Interface 4-Lite (AXI-Lite) IP interface, acting as a subordinate to a processor (AXI master). Using AXI, the processor writes directly to the coprocessor's registers to send instructions and data, and continues polling for status updates and new implication until the coprocessor completes.

Status changes dictate DPLL's flow, either allowing the search to continue assigning additional variables, or triggers backtracking on conflicts. A copy of all the implications are saved on the processor to avoid re-assigning implied variables, and further propagated to the remaining partitions.

5 EXPERIMENTS AND RESULTS

On a Xilinx Zynq chip with total capacity of 14400 LUTs and 28800 FF, our solution supports 224 parallel Clause Processors and 63 variables. We achieve a clock frequency of 106.66 MHz, utilizing 647 LUTRAM of on-chip memory, 13151 LUTs, and 11059 FFs.

Related work calculates throughput (in BPCs performed per second), assuming full data availability: i.e., not taking into account software execution and communication/data transfer latency. Whilst this is a useful metric to assess hardware performance in isolation (and we report equivalent results in Table 1), it does not accurately depict system performance; to do so, we break down

	Millions of BCP/s		
SAT Instance	Davis et al [5]	Thong et al [19]	Our Design
bmc-galileo-8	40	102	175
bmc-ibm-12	33	150	169

Table 1: Comparison of BCP engine throughput (BCPs/s) with related work. Results reflect maximum theoretical throughput, achieved only data is fully available to BCP engines.
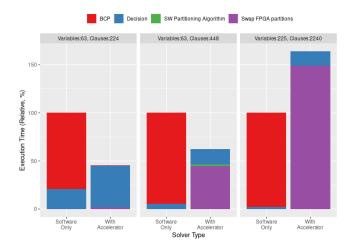

Figure 4: Breakdown of the total execution time across constituent components.

		Variables			
		63	126	252	630
Clauses	224	362M BCP/s 2.2x	17K BCP/s 0.17x	NA	NA
	448	702K BCP/s 1.6x	21K BCP/s 0.21x	13K BCP/s 0.08x	NA
	2240	441K BCP/s 1.91x	22K BCP/s 1.26x	16K BCP/s 0.61x	12K BCP/s 0.10x
	22400	313K BCP/s 6.32x	20K BCP/s 5.04x	16K BCP/s 4.86x	14K BCP/s 3.31x

Table 2: Varied clause/variable sizes and their impact on the relative speedup of hardware/software and the effective throughput of BCP engines.

the full execution in Figure 4 and evaluate speedup over vanilla software implementation, evaluating combinations of clause and variable sizes, with speedup depicted in Table 2 for meaningful combinations. For each combination, we also depict real throughput, in the form of BCPs/s averaged over total execution time (63 variables and 224 clauses is the theoretical upper bound, without the need for hot swapping). To evaluate the different effects of clause/variable sizes on execution, we fix one and vary the other, measuring total execution time: results are depicted in Figures 5 and 6.
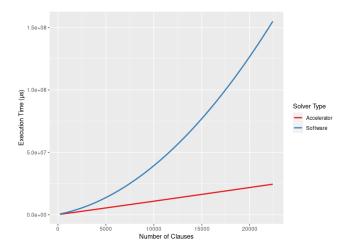
Figure 5: Effect of increasing clauses size on total execution time, for 63 variables.
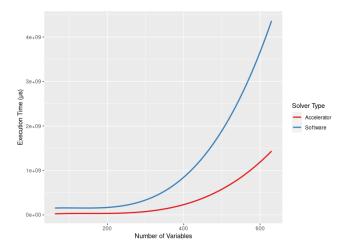

Figure 6: Effect of increasing variables size on total execution time, for 22400 clauses.

6 CONCLUSIONS

We described a SAT-solver hardware-accelerated architecture that outperforms state of the art by hot-swapping clause assignment at runtime, making efficient use of FPGA resources. Our solution prototype, on a Xilinx Zynq chip, is available in open-source. Practitioners may use the presented solution in their designs, whenever a problem is encoded in SAT form and performance is critical.

An important open question remains: our performance is constrained by how clauses are partitioned. A partitioning scheme that minimizes the distribution of variables among clauses will minimize runtime swapping, resulting in improved execution. However, how to best partition a formula to achieve this is not yet known. Future work must formulate this challenge as an optimization problem, and methods for its efficient solution must be devised. Once that is achieved, they can be applied (offline) prior to deployment on our architecture.

REFERENCES

[1] Rabie Ben Atitallah and Karim MA Ali. 2017. FPGA-Centric High Performance Embedded Computing: Challenges and Trends. In *2017 Euromicro Conference on Digital System Design (DSD)*. IEEE, 390–395.

[2] Stephen A Cook. 2023. The complexity of theorem-proving procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*. 143–152.

[3] Andreas Dandalis and Viktor K Prasanna. 2002. Run-time performance optimization of an FPGA-based deduction engine for SAT solvers. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 7, 4 (2002), 547–562.

[4] John D Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. 2008. Designing an efficient hardware implication accelerator for SAT solving. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 48–62.

[5] John D. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. 2008. A practical reconfigurable hardware accelerator for boolean satisfiability solvers. In *2008 45th ACM/IEEE Design Automation Conference*. 780–785. https://doi.org/10.1145/1391469.1391669

[6] Amir H Farrahi and Majid Sarrafzadeh. 1994. FPGA technology mapping for power minimization. In *International Workshop on Field Programmable Logic and Applications*. Springer, 66–77.

[7] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

[8] Weiwei Gong and Xu Zhou. 2017. A survey of SAT solver. In *AIP Conference Proceedings*, Vol. 1836. AIP Publishing.

[9] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. 2010. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation* 6, 4 (2010), 245–262.

[10] Anping He, Lvying Yu, Haitao Zhang, Lian Li, and Jinzhao Wu. 2018. A FPGA Based SAT Solver with High Random and Concurrent Strategies. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 221–228. https://doi.org/10.1109/QRS-C.2018.00049

[11] Teodor Ivan and El Mostapha Aboulhamid. 2013. An Efficient Hardware Implementation of a SAT Problem Solver on FPGA. In *2013 Euromicro Conference on Digital System Design*. 209–216. https://doi.org/10.1109/DSD.2013.31

[12] Teodor Ivan and El Mostapha Aboulhamid. 2013. Exploring limits of parallelism in FPGA-based Boolean satisfiability. In *2013 2nd Mediterranean Conference on Embedded Computing (MECO)*. 62–65. https://doi.org/10.1109/MECO.2013.6601319

[13] Ruben Martins, Vasco Manquinho, and Inês Lynce. 2012. An overview of parallel SAT solving. *Constraints* 17 (2012), 304–347.

[14] Muhammad Osama, Anton Wijs, and Armin Biere. 2021. SAT solving with GPU accelerated inprocessing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 133–151.

[15] I. Skliarova and A.B. Ferrari. 2004. A software/reconfigurable hardware SAT solver. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12, 4 (2004), 408–419. https://doi.org/10.1109/TVLSI.2004.825859

[16] Ali Asgar Sohanghpurwala, Mohamed W. Hassan, and Peter Athanas. 2017. Hardware accelerated SAT solvers: A survey. *J. Parallel and Distrib. Comput.* 106 (2017), 170–184. https://doi.org/10.1016/j.jpdc.2016.12.014

[17] Robert Stewart, Bernard Berthomieu, Paulo Garcia, Idris Ibrahim, Greg Michaelson, and Andrew Wallace. 2019. Verifying parallel dataflow transformations with model checking and its application to FPGAs. *Journal of Systems Architecture* 101 (2019), 101657.

[18] Jason Thong and Nicola Nicolici. 2013. FPGA acceleration of enhanced boolean constraint propagation for SAT solvers. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 234–241. https://doi.org/10.1109/ICCAD.2013.6691124

[19] Jason Thong and Nicola Nicolici. 2015. SAT solving using FPGA-based heterogeneous computing. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 232–239. https://doi.org/10.1109/ICCAD.2015.7372575

[20] Haoze Wu. 2017. Improving SAT-solving with machine learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 787–788.

[21] Peixin Zhong, Margaret Martonosi, and Pranav Ashar. 2000. FPGA-based SAT solver architecture with near-zero synthesis and layout overhead. *IEE Proceedings-Computers and Digital Techniques* 147, 3 (2000), 135–141.