IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 5, 2020, accepted March 3, 2020, date of publication March 11, 2020, date of current version March 19, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2980008

FPGA-Based Hardware/Software Co-Design of a Bio-Inspired SAT Solver

ANH HOANG NGOC NGUYEN[1], MASASHI AONO[2,3], AND YUKO HARA-AZUMI[1], (Member, IEEE)

[1]Department of Information and Communications, Tokyo Institute of Technology, Tokyo 152-8550, Japan
[2]Faculty of Environment and Information Studies, Keio University, Tokyo 108-8345, Japan
[3]Amoeba Energy Co., Ltd.

Corresponding author: Anh Hoang Ngoc Nguyen (anh@cad.ict.e.titech.ac.jp)

This work was supported in part by the JSPS KAKENHI from the Japan Science and Technology Agency (JST) under Grant 17H04677, and in part by the PRESTO from the Japan Science and Technology Agency (JST) under Grant JPMJPR1528.

ABSTRACT For various kinds of Internet of Things (IoT) systems whose control rules can be expressed in a Satisfiability (SAT) problem, this work aims at realizing an IoT-oriented FPGA-based SAT solver leveraging a bio-inspired algorithm, AmoebaSAT, using a hardware/software co-design approach. With regard to the software component, we extended the baseline algorithm to escape from local minima more quickly and achieve significant reduction in iteration count. With regard to hardware, we fully extracted the fine-grained parallelism of the algorithm to further accelerate the solution search. Through our evaluations using several benchmarks of varying variable count and complexity, we demonstrated the efficiency of our solver, especially for larger practical SAT instances. Compared with three state-of-the-art solvers (i.e., one software implementation of the original AmoebaSAT algorithm and two FPGA-based hardware solvers), we achieved an average of 15.9× and up to 48× reduction in iteration count. Furthermore, through in-depth analyses of the experimental results, we provided the essential findings on the relationship between the problem's complexity and the SAT algorithm that can be leveraged for extensions of both the hardware and software designs.

INDEX TERMS Bio-inspired algorithm, SAT solver, FPGA, high-level synthesis.

I. INTRODUCTION

Internet of Things (IoT) is an ever-growing technology in which a huge number of edge devices are connected in order to share information with each other and/or control a target object in their own applications. As more and more IoT applications seek for solutions to achieve their purposes, an increasing number of problems are dealt with as combinatorial optimization problems – for example, finding the best route or best combination from a known list [1], [2] can be handled as travelling salesman problems, minimum spanning tree problems, etc. Those solutions/decisions should be resolved on-site for self-sustainability (i.e., at the edge devices, not in the cloud) due to the network congestion and the need for real-time processing in IoT applications. Considering such environmental issues and application requirements, it is crucial to develop problem solvers that can run on the edge devices while exhibiting the following features:

1) *Scalability* in searching time: Most combinatorial problems are known to be NP-complete. Therefore, problem solvers need to efficiently handle a number of decision variables. Importantly, in many IoT applications, *near-optimal* solutions are often appreciated in lieu of a true optimal solution if searching time to obtain the optimal solution is expensive [3].
2) *Lightweight* operations: Since most edge devices have resource/power constraints [4], solution search via lightweight operations (such as binary, logic operations) are more appreciated than heavy arithmetic operations.
3) *Applicability* to various kinds of IoT applications: Since the diversity of IoT applications and their associated combinatorial problems is increasing, the ease of expressing various problems in a uniform way is crucial to enabling wide application and reuse of the solver.

For decades, a variety of heuristics for combinatorial problems have been studied, including two main types of

The associate editor coordinating the review of this manuscript and approving it for publication was Zhenliang Zhang.

approaches: deterministic and stochastic. Although deterministic algorithms (ILP [5], MIP [6], etc.) can guarantee whether or not there exists a solution to a given problem, their complexity in solution derivation makes it difficult to achieve feature (1). Stochastic algorithms, on the other hand, are reasonable and capable of finding a solution in less time. There are roughly three types of stochastic algorithms: conventional evolutionary algorithms such as simulated annealing (SA) [7] and genetic algorithms (GAs) [8], etc.), annealers based on the Ising model [9], and stochastic solvers for satisfiability (SAT) problems [10]–[12]. Since conventional evolutionary algorithms tend to employ many heavy arithmetic operations, they are not suitable for resource-constrained devices (i.e., they lack feature (2)). The Ising model (quantum annealing [13], digital annealing [14], and CMOS annealing [15]) is an emerging approach which can search for a solution faster than conventional algorithms (such as SA) and uses binary lightweight operations. However, as the transformation from various problems into the Ising model has not been well studied, this approach is still limited to few applications (i.e., lack of the feature (3)). In addition, some annealing machines must be operated under strict conditions and consume large amount of area. Hence, for now, they are more well-suited to cloud settings rather than for edge devices. Contrary, although most SAT solvers handle Boolean variables, generalization/encoding techniques to handle combinatorial problems have been well studied [16]. Moreover, stochastic SAT solvers are lightweight, yet they can achieve searching performance comparable to their heavier stochastic alternatives. Hence, out of the aforementioned stochastic techniques, this approach appears to be the most suitable one for edge devices. Aside from WalkSAT, a recently-developed SAT algorithm, AmoebaSAT [12], has received attention in the form of IoT-oriented hardware implementations on various devices such as FPGAs [17], [18] and electrical Brownian ratchets [19]. Such IoT applicaitons handle mainly hundreds of variables, enabling the algorithm to extract a high degree of parallelism on edge devices. For example, it was demonstrated in [12] that AmoebaSAT outperforms WalkSAT in terms of iterations# (i.e., sequential time step in which one or more variable assignments are flipped). Details of the AmoebaSAT algorithm are presented in Section II-A).

In this paper, we propose an effective FPGA-based SAT solver that is faster and more practical than a the current state-of-the-art AmoebaSAT hardware implementation [17]. Our solver has the following important features: (i) it employs algorithmic extensions to more efficient solution search, escaping from local minima in fewer iterations than in [17], (ii) it exploits a high-level design approach to extract the inherent parallelism of AmoebaSAT, and (iii) it efficiently solves various problems in terms of both the size and the community structure [20]). A preliminary version of our work appeared in [18]. In this extended journal, we elaborate the feature (i) with in-depth analyses on different behaviors of intermediate variables by comparing the original AmoebaSAT and its algorithmic extensions. We additionally demonstrate the effectiveness of our solver through analyzing flips# per iteration compared with state-of-the-art hardware solvers ([21] and [17]). We also apply architectural extensions to enhance the feature (ii) by removing the loop-carried dependencies to achieve fine-grained parallelism. Finally regarding feature (iii), we evaluated our solver on two sets of benchmark instances and discussed the scalability towards IoT-based applications such as 5G network scheduling. We achieved the speed up of up to 39,350× versus the software solver [12] and 23.7× versus the hardware solver [17].

Our contributions are summed up as follows:

- We develop an FPGA-based AmoebaSAT solver in a hardware/software co-design manner to extract fine-grained (i.e., variable-level) parallelism, achieving better performance than state-of-the-art implemetations.
- On the software side, we generate effective feedback mechanisms so that the solution search part (on the hardware side) can efficiently search for a solution, leading to achieve less iterations#.
- On the hardware side, we apply hardware-aware design optimizations through high-level synthesis, such as targeting irregular memory accesses and applying loop pipelining to reduce cycle counts.
- Through two sets of evaluations (random and graph coloring instances), we provide useful observations and findings for deploying the AmoebaSAT solver on IoT devices.

The remainder of this paper is organized as follows: Section II explains in details the original AmoebaSAT algorithm. Section III and Section IV present our algorithmic extensions and high-level design techniques, respectively. Section V describes our experiment setup and demonstrates the effectiveness of our work versus the state-of-the-art. Section VI concludes this paper.

II. AmoebaSAT: AMOEBA-INSPIRED SAT SOLVER ALGORITHM

This section explains the overview of the AmoebaSAT algorithm, followed by its unique feature of conflict resolution mechanisms called "bounceback rules."

A. COMPUTING MODEL

Satisfiability (SAT) is concerned with determining the existence of a solution to a given formula, where the solution is a variable assignment that is said to *satisfy* it. The *Boolean satisfiability problem*, in particular, determines whether or not a Boolean formula (i.e., a formula $f(V) \mid v \in \{\texttt{true}, \texttt{false}\}\ \forall\, v \in V$) could be satisfied. A Boolean SAT formula (hereafter referred to as a *SAT instance*) is often represented in *conjunctive normal form*, which is the conjunction ($\wedge$) of multiple *clauses*, each of which is a disjunction ($\vee$) of one or more logical *variables*. For example, the formula $f = (\overline{x_1} \vee x_2) \wedge (x_1 \vee x_2)$ is a SAT instance of two literals and is satisfied (i.e., $f = 1$) when $(x_1, x_2) \in \{(0, 1), (1, 1)\}$.

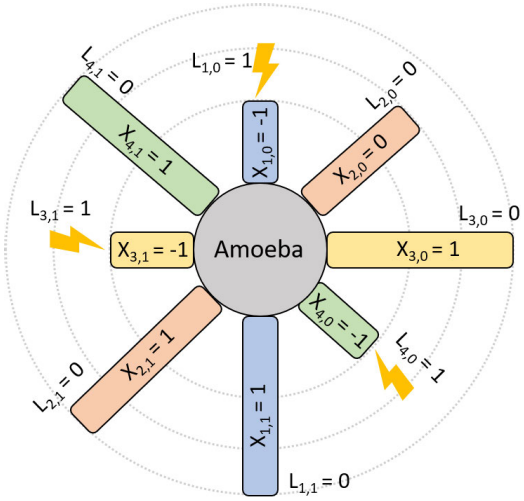

FIGURE 1. The AmoebaSAT model for a four-variable instance.

The computing model of AmoebaSAT is inspired from the spatiotemporal dynamics of an amoeba in its adaption with the surrounding environment. This Amoeba is a single-cell organism that uses multiple expandable pseudopods (herein referred to as *branches*) to effectively find food sources. It is photosensitive, so whenever light is projected on to a branch, the organism will typically retract that branch and attempt to expand in other directions remaining devoid of light. Inspired by such an expansion scheme, AmoebaSAT [12] constructs a unit updating model in order to solve SAT problems.[1] For an instance of N variables, the solver employs a model of $2N$ units, wherein each ternary pair of units determines the value of a single variable. Each unit is associated with a set of internal variables capturing its expansion state. By updating all variables, all units' expansion statuses are updated as well, effectively corresponding to how a biological amoeba expands its branches in nature. Also, inter-clause conflicts generate control signals that simulate light stimuli, guiding the amoeba to conduct iterative updating of variable assignments in an effective fashion.

For N variables, each unit (i, v) represents the variable x_i, $i \in \{1, 2, \ldots, N\}$ under the speculative variable assignment $v \in \{0, 1\}$. Unit $(i, 0)$ asserts $x_i = 0$, whereas unit $(i, 1)$ asserts $x_i = 1$. Fig. 1 shows a simplified AmoebaSAT model for a four-variable instance, where it uses four pairs of ternary units, each color-coded pair determine the value of a unique variable. Each unit has three expansion states (represented by an intermediate variable X), which are expanded ($X = 1$), neutral ($X = 0$) and retracted ($X = -1$). The central circle represents the body of the amoeba, which supplies or cuts off resource to each branch for extension and retraction, respectively. The resource supply decision on a branch depends on whether it is subjected to a stimulus of light, which is simulated by the assignment $L = 1$ and represented by a lightning strike in the figure (i.e., $L_{1,0}$, $L_{3,1}$, and $L_{4,0}$). The assignment $L_{i,v} = 1$ means that the light stimulus is turned on at branch (i, v), which cuts off the resource supply to it. Definitions and descriptions of variables used in the AmoebaSAT model are summarized in Table 1.

[1]The extensions of AmoebaSAT have been studied for combinatorial optimization problems such as traveling salesman problem (TSP), whose detailed explanations are omitted in this paper.

TABLE 1. Definitions and descriptions of variables.

Variable	Description
$x_i \in \{0, 1\}$	i^{th} Boolean variable ($i \in \{1, 2, \cdots, N\}$).
$X_{i,v} \in \{-1, 0, 1\}$	Intermediate variable representing the equilibrium state of the unit (i, v). A pair of $X_{i,0}$ and $X_{i,1}$ expresses the assignment of x_i (i.e., $x_i = v$ when $X_{i,v} = 1$ and $X_{i,1-v} \leq 0$).
$Y_{i,v} \in \{0, 1\}$	Intermediate variable representing a resource supply on the unit (i, v). $Y_{i,v} = 1$ means that the unit (i, v) is supplied to increase its volume ($X_{i,v}$) and thus considers to assign $x_i = v$.
$L_{i,v} \in \{0, 1\}$	Control variable representing a light stimulus for the resource-supply bounceback on the unit (i, v).
$Z_{i,v} \in [0.0, 1.0]$	Fluctuated variable representing random extension of the unit (i, v).
ϵ	Exploration parameter expressing an erroneous behavior. The optimal value depends on how the oscillation is generated.

Starting from an initial state in which all branches of the amoeba are neutral (i.e., $X_{i,v} = 0$) and no supply decisions or light stimuli are applied (i.e., $Y_{i,v} = 0$, $L_{i,v} = 0$), the system will iteratively update each units' intermediate variables until a satisfying solution is found. At iteration t, each variable x_i determines its value based on the expansion volume of its represented units $X_{i,0}$ and $X_{i,1}$.

The decision of x_i and unit volume $X_{i,v}$ are defined mathematically and provided in the formulae (1) and (2), respectively:

$$x_i(t) = \begin{cases} 0, & X_{i,0}(t) = 1 \;\&\; X_{i,1}(t) \leq 0 \\ 1, & X_{i,1}(t) = 1 \;\&\; X_{i,0}(t) \leq 0 \\ x_i(t-1), & \text{otherwise} \end{cases} \tag{1}$$

$$X_{i,v}(t) = \begin{cases} X_{i,v}(t-1) + 1, & Y_{i,v}(t) = 1 \;\&\; X_{i,v}(t-1) < 1 \\ X_{i,v}(t-1) - 1, & Y_{i,v}(t) = 0 \;\&\; X_{i,v}(t-1) > -1 \\ X_{i,v}(t-1), & \text{otherwise} \end{cases} \tag{2}$$

The resource supply $Y_{i,v}$ determines whether the unit (i, v) should attempt to expand or retract (i.e, whether $X_{i,v}$ can increase or decrease in accordance with formula 2). The state transition of $Y_{i,v}$ depends on the dynamics of the fluctuated variable $Z_{i,v}$ (for $2N$ units, $Z_{i,v}$'s are independent each other) and an exploration parameter ϵ. Note that $Y_{i,v}$ is determined by $L_{i,v}$ of *the previous iteration* $(t - 1)$.

$$Y_{i,v}(t) = \begin{cases} 0, & L_{i,v}(t-1) = 1 \\ sgn(1 - \epsilon - Z_{i,v}(t)), & \text{otherwise} \end{cases} \tag{3}$$

where *sgn* is a sign function:

$$sgn(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

Formula (3) expresses that a unit (i, v) can be supplied resources based on its control signal $L_{i,v}$ (the concept and definition of $L_{i,v}$ will be explained in Section II-B). When there is no light stimulus ($L_{i,v} = 0$), as long as the fluctuation $Z_{i,v}$ does not exceed a threshold of error occurrence (i.e., $1 - \epsilon$), the unit can freely increase its volume $X_{i,v}$. While the original AmoebaSAT uses a logistic map $Z_{i,v}(t) = 4Z_{i,v}(t-1)(1-Z_{i,v}(t-1))$ to let $Z_{i,v}$ exhibit chaotic oscillation through iterations, this work uses *a tent map* [22] to generate oscillatory dynamics considering the simplicity and suitability for implementing fixed-point operations on FPGAs (the details will be explained in Section IV-A).

$$Z_{i,v}(t) = \begin{cases} 2Z_{i,v}(t-1), & Z_{i,v}(t-1) < 0.5 \\ 2(1 - Z_{i,v}(t-1)), & Z_{i,v}(t-1) > 0.5 \end{cases} \quad (5)$$

B. BOUNCEBACK RULES

The light stimulus $L_{i,v}$ used in formula (3) is an essential concept in AmoebaSAT to detect conflicts induced by the variable assignments in the previous iteration $t-1$. In other words, AmoebaSAT *learns from failures* by way of L. If there is a conflict, the *bounceback control signal* is asserted (i.e., $L_{i,v} = 1$) on the corresponding units to cut off the resource supply, thus making them retract in the next iteration. The bounceback control is a set of rules constructed from the clauses of the target SAT instance as expressed in formula (6).

$$L_{i,v}(t) = \begin{cases} 1, & (P, Q) \in B_{ON} \text{ such that } (i, v) \in Q \ \& \\ & \forall (j, u) \in P, X_{j,u}(t-1) > 0 \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

where (P, Q) represents a pair of contradictory units that cannot be true at the same time, and B_{ON} represents the set of bounceback rules geared toward branch retraction. B_{ON} consists of three types of rules: *INTRA*, *INTER*, and *CONTRA*. Hence, $B_{ON} = INTRA \cup INTER \cup CONTRA$.

Since a variable x_i can only be either 0 or 1 in a given iteration, the two units representing $(i, 0)$ and $(i, 1)$ must not be supplied at the same time. *INTRA* is therefore defined to prohibit the inconsistent assignment of each variable x_i:

$$INTRA \ni ((i, v), (i, 1-v)), \quad v \in \{0, 1\} \quad (7)$$

INTER captures the contradiction between elements within each clause. For example, in the clause $C = (x_1 \vee x_2 \vee \overline{x_3})$, if x_1 and x_2 are assigned to 0, then x_3 should be set to 0 in order to keep the clause true. Therefore, while the unit $(3, 0)$ should be expanded, the unit $(3, 1)$ should be bounced-back to avoid the contradiction. The set *INTER* is defined as follows:

$$Inter = \begin{cases} (P, (i, 0)), & C \ni i \\ (P, (i, 1)), & C \ni -i \end{cases} \quad (8)$$

where P, for all $j \neq i$, includes the following units:

$$P \ni \begin{cases} (j, 0), & C \ni j \\ (j, 1), & C \ni -j \end{cases} \quad (9)$$

Algorithm 1 AmoebaSAT

Input: the target instance f & the maximum iterations t_{max}
Output: variable assignments x's

1: $t = 0$
2: Construct the bounceback rules B_{ON} by the formulae (7), (8), & (11)
3: **for all** (i,v) = (1,0) to (N,1) **do**
4: Initialize $X_{i,v}(0) = Y_{i,v}(0) = L_{i,v}(1) = 0$
5: Initialize $Z_{i,v}(0)$ with a random value in (0.0, 1.0) other than 0.5
6: **end for**
7: **while** $t < t_{max}$ **do**
8: Obtain $x_i(t)$ by the formula (1)
9: **if** $f = 1$ **then**
10: Return 'Found' & output the solution x's
11: **else**
12: Calculate $X_{i,v}(t)$, $Y_{i,v}(t)$, $Z_{i,v}(t)$, and $L_{i,v}(t+1)$ by the formulae (2), (3), (5), & (6)
13: $t = t + 1$
14: **end if**
15: **end while**
16: Return 'Found no solution'

Finally, *CONTRA* manages the contradiction between multiple units from different clauses. For example, given two clauses $C_1 = (\overline{x_3} \vee x_4 \vee \overline{x_1})$ and $C_2 = (x_3 \vee x_4 \vee \overline{x_2})$, if at a particular timestep the assignments $x_1 = 1$, $x_2 = 1$ and $x_4 = 0$ all hold, then $x_3 = 0$ must be held to satisfy C_1. However, $x_3 = 1$ also needs to be held to satisfy C_2. These requirements will lead to the bounceback of both $X_{3,0}$ and $X_{3,1}$, i.e., leading to a situation where x_3 unreasonably retreats from both possible assignments 0 and 1. *CONTRA* rules are enforced to preclude such undesired behavior. Firstly, two sets $\mathbf{P}_{i,0}$ and $\mathbf{P}_{i,1}$ are constructed based upon the *INTER* rule set:

$$\begin{matrix} \mathbf{P}_{i,0} \ni P, & \mathbf{P}_{i,0} \in INTER \\ \mathbf{P}_{i,1} \ni P, & \mathbf{P}_{i,1} \in INTER \end{matrix} \quad (10)$$

By combining each element $(P_{i,0}, P_{i,1})$ of the above two sets together, the set *CONTRA* is formed:

$$\forall (P_{i,0}, P_{i,1}) \in \mathbf{P}_{i,0} \times \mathbf{P}_{i,1} \quad (CONTRA \ni (P_{i,0} \cup P_{i,1}, P_{i,0} \cup P_{i,1})) \quad (11)$$

To summarize the aforementioned formulae, Algorithm 1 describes the simplified pseudo code of the AmoebaSAT algorithm. We also illustrate how the intermediate variables work for a variable x_i in Fig. 2, where we assume that x_i is initialized as 0. The upper part, middle part and bottom part of the figure describe the statuses of Z's, Y's, and X's, respectively, each of which is illustrated through three consecutive iterations (i.e., $t = 0$, $t = 1$ and $t = 2$) from left to right. At the iteration $t = 0$, Y's, X's and L's are initialized as 0, while Z's take a random value in range (0.0; 1.0) other than 0.5 (Lines 4-5). Then, at the iteration $t = 1$, because fluctuated variable $Z_{i,0}$ updates by the formula 5 and exceeds

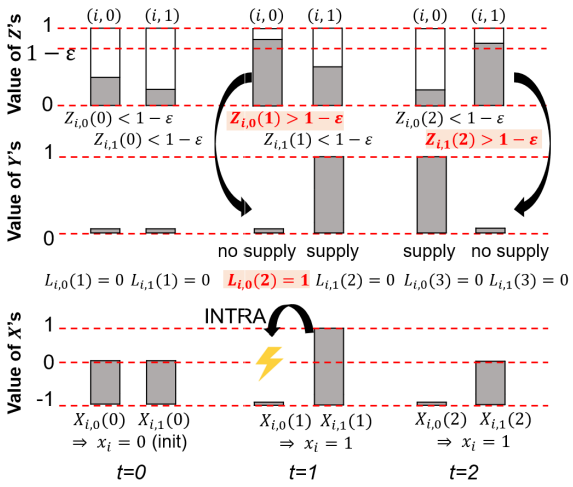

FIGURE 2. An example of the status changes in intermediate variables.

$1-\epsilon$, $Y_{i,0} = 0$ is set (i.e., the resource supply is cut off) by the formula (3), leading to decrement $X_{i,0}$ by 1 (i.e., $X_{i,0} = -1$). On the other hand, since $Z_{i,1}$ is within the range of normal occurrence, $Y_{i,1} = 1$ supports $X_{i,1}$ to increase by 1 (i.e., $X_{i,1} = 1$). Since the exploration parameter ϵ indicates the probability of error occurrence, even if a unit is not blocked ($L_{i,v} = 0$), the supply is still cut off at the unit ($Y_{i,v} = 0$). Then, $x_i = 1$ is set by the formula (1). Because of $X_{i,1} = 1$, *INTRA* prohibits to supply the resource to $X_{i,0}$, resulting in $L_{i,0} = 1$ (represented by a lightning bolt on the unit $(i, 0)$) and $L_{i,1} = 0$ (no lightning on the unit $(i, 1)$) (Line 7-12). Similarly, at the iteration $t = 2$, X's, Y's, Z's and L's are calculated and set accordingly. Since no resource supply is provided to the unit $(i, 1)$ (i.e., $Y_{i,1} = 0$), the volume of $X_{i,0}$ is kept the same (i.e., $X_{i,0} = -1$), leading to unchange the value of x_i. This loop (Lines 6-15) is iteratively performed until a solution is found (i.e., when all units satisfy $f = 1$(Line 9) or the maximum iteration t_{max} is reached. If the solver reaches t_{max}, it will return "Found no solution" (Line 16).

As demonstrated, AmoebaSAT computes updates of all variables in parallel. Hence, it has finer-grained parallelism than conventional SAT algorithms that update a single variable per iteration, such as WalkSAT. In this work, we aim at developing an FPGA-based SAT solver that is faster than the current state of the art by exploiting two important features of the AmoebaSAT algorithm: (1) *variable-level parallelism* and the (2) *ability to learn from failures* (i.e., using intermediate variables). These two features encourage an effective solution search by flipping as many variables as necessary per iteration when the current variable assignments are far from a solution, especially during earlier iterations. Also, through the algorithmic extension described in Section III, our solver succeeds in avoidance of useless flips as it approaches a solution (i.e., during later iterations).

III. HIGH-LEVEL DESIGN APPROACH

This section describes the overview of our AmoebaSAT solver, followed by two additional algorithmic extensions to the bounceback control mechanism.

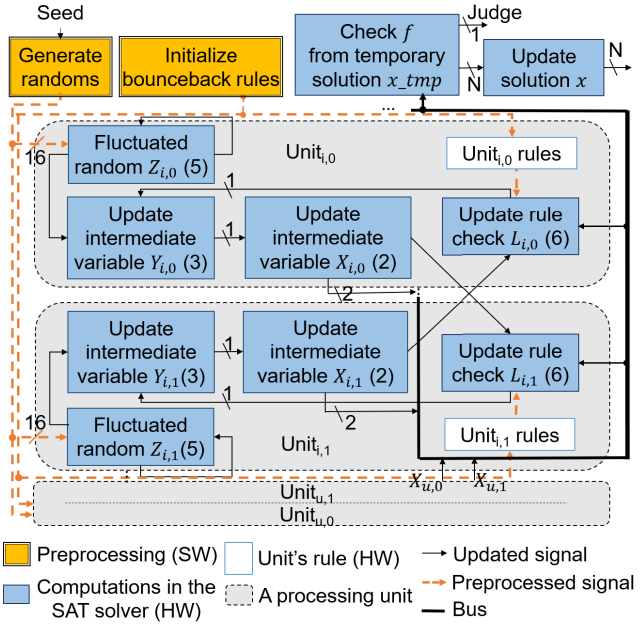

FIGURE 3. An overview of our AmoebaSAT solver (the numbers in the parentheses represent the formulae explained in Section II).

A. THE OVERVIEW

Our SAT solver is realized in a hardware/software co-design manner as described in Fig. 3: construction of the bounceback rules, as well as the random initial generation for Z's are realized as a software pre-processing component (the upper left of the diagram) and performed only once at the beginning of the computation. All other components comprise the iterative solution search itself, and are performed in hardware. Each AmoebaSAT unit is realized as a processing element updating its own intermediate variables in each iteration. A shared bus is used to deliver the equilibrium information X's of each unit to all other related ones so that their bounceback control signals (i.e., L's) can be updated accordingly. Since many of today's FPGA devices feature a hard microprocessor core, we target an FPGA device for the implementation so that although the pre-processing component runs on the processor core, the units are designed as a hardware accelerator utilizing the reconfigurable fabric through high-level design (i.e., HLS). Although in this paper we utilize a Xilinx FPGA device and Vivado HLS, this work is not limited to any specific FPGA device or HLS tool.

The remainder of this section elaborates the algorithmic extensions applied to bounceback rule generation in pre-processing and bounceback controls (i.e., update L's). Section IV follows this by providing details of hardware design optimizations applied to each unit.

B. EXTENSIONS OF BOUNCEBACK RULES

Although the bounceback controls described in section II-B effectively resolve conflicts by learning from the failures (i.e., with a help of the recent statuses of the intermediate variables), we found room for improvement in the bounceback

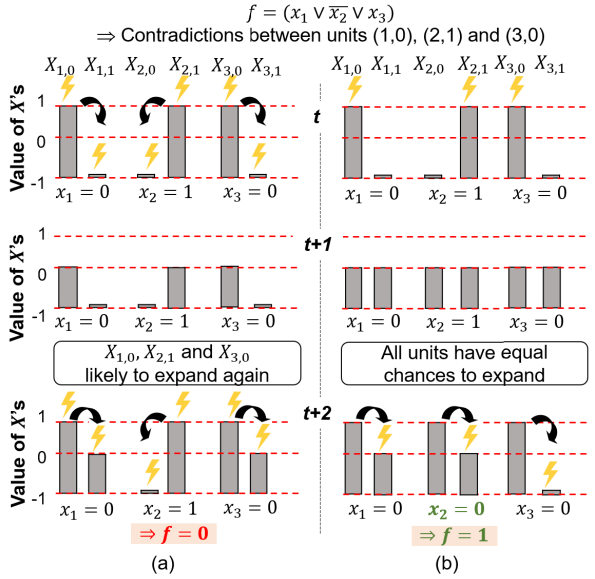

FIGURE 4. Comparison of (a) *INTER* and (b) *COLLAPSE* (for $f = (x_1 \vee \overline{x_2} \vee x_3)$).

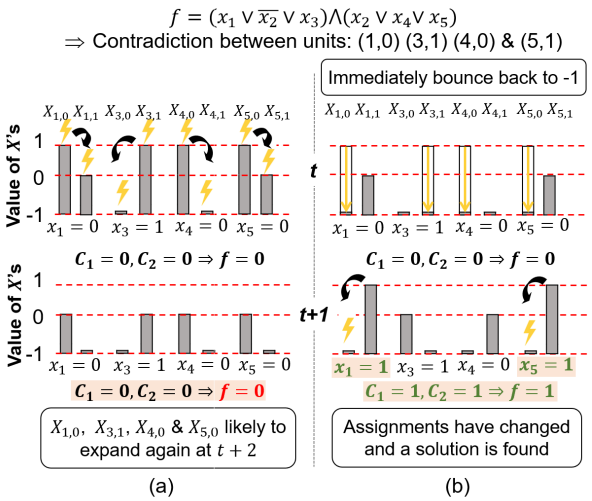

FIGURE 5. Comparison of (a) *CONTRA* and (b) *HyperCONTRA* (for $C_1 = (x_1 \vee \overline{x_2} \vee \overline{x_3})$ and $C_2 = (x_2 \vee x_4 \vee x_5)$).

controls by observing how the values of X's change upon application of stimuli. Specifically, due to the dependency between multiple variables, bouncebacks are frequently initiated in accordance with *INTER* and *CONTRA*. This may result in units cycling through previous expansion states, impeding further variable updates and delaying convergence toward a solution. Therefore, to let our SAT solver more efficiently arrive at a solution, we extend *INTER* and *CONTRA* to enable more direct control over the state of intermediate variables, leading to less number of iterations.

1) *COLLAPSE*

Fig. 4(a) illustrates how L's is set when unsatisfactory variable assignments in a clause are detected by *INTER* rules, using the formula $f = (x_1 \vee \overline{x_2} \vee x_3)$ as an example. The state of X's, represented by bar height, are shown for three pairs of units over three consecutive iterations (from top to bottom). When L's is set to 1, a lightning bolt is displayed in the figure. When $X_{1,0}$, $X_{2,1}$, and $X_{3,0}$ are all 1 (i.e., $x_1 = 0$, $x_2 = 1$, and $x_3 = 0$), the clause is unsatisfied (i.e., $f = 0$). Hence, an *INTER* contradiction is detected by formula (8), which results in setting the units' elements of L's to 1. However, in each pair of units, an *INTRA* contradiction also occurs in accordance with formula (7), resulting in L's being set to 1 and the counterpart units being blocked accordingly. Consequently, $X_{1,0} = X_{2,1} = X_{3,0} = 0$ and $X_{1,1} = X_{2,0} = X_{3,1} = -1$ are set at iteration $t + 1$. In this state, $X_{1,0}$, $X_{2,1}$, and $X_{3,0}$ are likely to return to 1 again sooner than their counterparts, leading to a repeat of the same behavior at iteration $t + 3$. This expresses a lack of progress that delays convergence to a solution. Since such behaviour may be repeated in consecutive iterations, variable assignments may remain unchanged, leaving the solver susceptible to getting stuck at a local minimum for several iterations upon encountering an *INTER* contradiction.

We thus propose an additional type of bounceback control, *COLLAPSE*, to mitigate such back-and-forth behaviors. Whereas *INTER* handles when to turn elements of L's on, *COLLAPSE* considers when to turn elements of L's *off*. As shown in Fig. 4(b), when unsatisfactory variable assignments are found, it collapses part of *INTER* by *disabling the bounceback of the counterpart units* at iteration t. This results in a situation where counterparts $X_{i,0}$ and $X_{i,1}$ each have an equal chance to become 1, leading to a higher probability of changing variable assignments and satisfying clause f at iteration $t + 2$. *COLLAPSE* is derived based on *INTER* and formulated as follows:

$$\begin{aligned} COLLAPSE \ni (&\{(i,0)|x_i \in C_k\} \cup \{(i,1)|\overline{x_i} \in C_k\}, \\ &\{(i,1)|x_i \in C_k\} \cup \{(i,0)|\overline{x_i} \in C_k\}) \end{aligned} \quad (12)$$

Since *COLLAPSE* specifies when to turn L's off, this bounceback rule is handled as B_{OFF} (i.e., $B_{OFF} = COLLAPSE$). Then the formula (6) is updated as follows:

$$L_{i,v}(t) = \begin{cases} & (P,Q) \in B_{ON} \ \& \ (P,Q) \notin B_{OFF} \\ 1, & \text{such that } (i,v) \in Q \\ & \text{and } \forall (j,u) \in P, X_{j,u}(t-1) > 0 \\ 0, & \text{otherwise} \end{cases} \quad (13)$$

Note that B_{ON} works similarly to the original AmoebaSAT, and B_{OFF} supplements it to encourage more efficient changes in the states of intermediate variables.

2) *HyperCONTRA*

Fig. 5(a) describes the variable assignments for two clauses, $C_1 = (x_1 \vee \overline{x_2} \vee \overline{x_3})$ and $C_2 = (x_2 \vee x_4 \vee x_5)$, both of which include x_2. The states of these variables are illustrated across two consecutive iterations, from top to bottom. If $x_1 = x_4 = x_5 = 0$ and $x_3 = 1$ are set, x_2 would

need to be 0 and 1 concurrently to satisfy both C_1 and C_2, respectively. In this case *CONTRA* takes effect, and elements of L's corresponding to these units are set to 1. In addition, their counterpart units' L's values are also set to 1 due to *INTRA*. Although all X's values then decrease their volume at iteration $t+1$, the clauses are not yet satisfied since the variable assignments have not yet changed. Recalling the definitions of the intermediate variables, the reader will find that it takes another two iterations at minimum for the variable assignments to change. In other words, the clauses can be satisfied no earlier than at iteration $t+3$. Moreover, since units (1, 0), (3, 1), (4, 0), and (5, 0) each have a larger volume than its counterpart, they are more likely to expand again in the following iteration (i.e., $t+2$) and repeat the same behavior as that of iteration t. As the repetition of these behaviors likely fails to change the variable assignments, the solver is prone to getting stuck in a local minima.

We find that the aforementioned bounceback controls through L's are indirect and inefficient when *CONTRA* takes effect. Therefore, we propose to extend *CONTRA* to forcefully change the corresponding values of X's by setting them to -1 immediately (i.e., in the current iteration t). These requests to immediately change X's are represented by straight downward arrows in Fig. 5(b). This extension, referred to as *HyperCONTRA*, also refrains from setting counterparts' L values to 1 and can encourage the quick change of variable assignments. Then, a solution may be found in the following iteration at the earliest, as depicted in Fig. 5(b). The updates to formula (2) as required by *HyperCONTRA* are defined as follows:

$$X_{i,v}(t) = \begin{cases} -1, & (i,v) \in B^H_{ON} \\ X_{i,v}(t-1)+1, & (i,v) \notin B^H_{ON} \ \& \ Y_{i,v}(t) = 1 \\ & \& \ X_{i,v}(t-1) < 1 \\ X_{i,v}(t-1)-1, & (i,v) \notin B^H_{ON} \ \& \ Y_{i,v}(t) = 0 \\ & \& \ X_{i,v}(t-1) > -1 \\ X_{i,v}(t-1), & \text{otherwise} \end{cases} \tag{14}$$

where $B_{ON} = INTRA \cup INTER$ (this update of B_{ON} applies to the formula (6)) and $B^H_{ON} = HyperCONTRA$. Note that although *HyperCONTRA* is defined similarly to *CONTRA*, the variables to be updated are different – while *CONTRA* updates L's, *HyperCONTRA* directly updates X's.

Because *COLLAPSE* and *HyperCONTRA* are orthogonal, they can be both applied, i.e., $B_{ON} = INTRA \cup INTER$, $B_{OFF} = COLLAPSE$, and $B^H_{ON} = HyperCONTRA$.

IV. HARDWARE-AWARE DESIGN OPTIMIZATIONS

Although the original AmoebaSAT algorithm already well-matches with hardware implementation due to its inherent parallelism, to fully enjoy the parallelism, we further applied several optimizations and extensions in the hardware design. This section describes these design techniques we applied to the hardware parts.

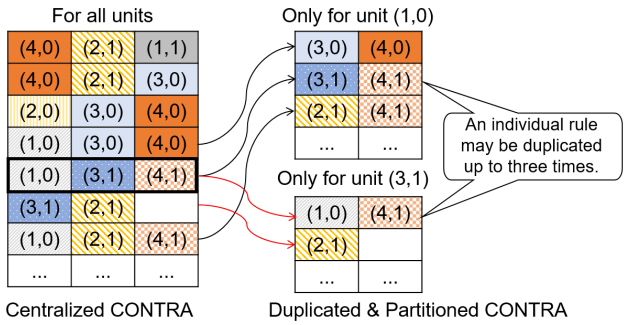

FIGURE 6. Localization of the bounceback rules.

A. OPERATIONAL TYPES AND DEFINITION

Floating-point operations typically consume a lot of hardware resources in the dedicated hardware designs. Therefore, especially on resource-limited FPGAs, the floating-to-fixed-point conversion is commonly applied. In this work, we also applied this conversion to Z's so that in total 16 bits are used, which did not affect the precision at all.We also minimized the bitwidth of the other variables (two bits for X's and one bit for the others) using the `ap_int` library provided by the high-level synthesis tool we used (Xilinx Vivado HLS).

To save the hardware resource consumption, we realized the oscillation for Z's by replacing a logistic map adopted in [12] with a tent map as already explained in the formula (5). While the logistic map calculates $4Z_{i,v} \times (1 - Z_{i,v})$ and hence uses a multiplier (i.e., consuming several DSPs), the tent map is realized by a shifter only. Then, along with the floating-to-fixed-point conversion and the tent map, we explored the best ϵ value over 500 Monte-Carlo simulation runs with the 0.01 interval on the benchmark instances we used in the experiments. By comprehensive examinations, we set $\epsilon = 0.32$ to all instances in our evaluation in Section V.

B. RULE LOCALIZATION

As illustrated in Fig. 3, our AmoebaSAT solver is implemented based on the units to extract the variable-level parallelism. In each iteration, it is sufficient for each unit to check only its related bounceback rules, whose amount is relatively small especially for realistic instances due to their inherent community structure [20].

We thus partially duplicate the bounceback rules so that each unit has its related rules in the local lookup table (e.g., "Unit$_{i,0}$ rules" for the Unit$_{i,0}$ in Fig. 3). The concept of this rule localization through duplication and partitioning is illustrated in Fig. 6. As shown in the figure, all units can in parallel check their rules at the cost of resource utilization. The conflict detection of each unit still follows the formulas of updating L's (explained in Sections II and III) which turns on or off according to the expansion statuses of the conflicting units.

For an instance composed of N variables and M clauses, our implementation has $2N$ separate units, each one of which has its own rules by the aforementioned rule localization.

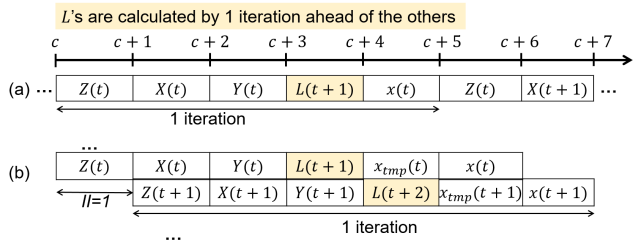

FIGURE 7. Loop optimization: (a) the baseline and (b) pipelining.

Since the total rules# are originally constructed from M clauses, the total area in the solution search part realized in hardware depends on both the variables# N and clauses# M, i.e., $O(NM)$. This will be evaluated in our evaluations later in Section V-B.

C. LOOP OPTIMIZATIONS

Each of the intermediate variables and the bounceback rules is implemented as an array and iteratively updated in a function within a loop. The functions are then individually implemented in a single cycle as shown in Fig. 7(a). Although pipelining this loop would be one of the most resource-and-performance-efficient optimizations as have been done to various HLS-based designs, in the AmoebaSAT implementation, it is difficult to naively apply loop pipelining due to some dependency and limitation. First there are two types of loop-carried Write-After-Read (WAR) dependencies in our implementation when we aim at pipelining the loop with the minimum initiation interval (II = 1). One is, for example, on L's – the old values of L's (set in the previous iteration) are read to update Y's, and new values are then written to L's as formulated in the formula (6). The other happens during the satisfiability checking (read on x's set in the previous iteration) and update (write on x's). The difference between these WAR dependencies are that while the WAR on L's is only within a unit, the WAR on x's relates to all units since the satisfiability checking can be done only after x's are all updated. These WAR dependencies hinder the loop pipelining and end up with sequentially executing all functions (see Fig. 7(a)).

Another dependency is irregular memory accesses where access patterns (or intervals) on the variables/arrays are determined by an indirect reference or calculation and thus are not statically analyzable. For example, as shown in the code snippet below where a function of turning L's on according to *INTER* (represented in an array `f`) is described, the arrays of the intermediate variables X's and the bounceback rules L's are irregularly accessed.

```
void Lon_inter(int X[2N], int f[M][3], int L[2N]){
  for(int i=0;i<M;i++){
    id1=f[i][0]; // obtain the 1st index (unit)
    id2=f[i][1]; // obtain the 2nd index (unit)
    id3=f[i][2]; // obtain the 3rd index (unit)
    if((X[id1]>0)&(X[id2]>0)&(X[id3]>0)){
      // set L=1 for these three units
      L[id1]=L[id2]=L[id3]=1;
}}}
```

Finally, the sequential access limitation to the FPGA built-in memories also hinders the possibility of achieving II = 1 if the arrays are implemented on the memories (e.g., the arrays `X`, `f`, and `L` in the code above).

In this work, we applied two optimization techniques in order to resolve the above issues and pipeline loops with as short II as possible – (1) removing the loop-carried WAR dependencies and (2) flattening the arrays to resolve the indirect and sequential memory accesses. The optimization (1) is two-fold; On one hand, the WAR dependency on L's was resolved by a Xilinx Vivado HLS pragma "`dependence false`" with the direction "`WAR`", which helped to write and read L's in one cycle (e.g., the cycle $c + 3$ in Fig. 7(b)). On the other hand, the WAR dependency on x's was resolved by introducing a temporary array x_{tmp} so that the variable assignments in the previous iteration can be read from x_{tmp}'s while updating x's with the current assignments simultaneously (e.g., the cycle $c + 5$ in Fig. 7(b)) – the functions handling x_{tmp}'s and x's are implemented as the "Check f from temporary solution x_{tmp}" and "Update solution x" blocks, respectively, in Fig. 3. Next, the optimization (2) utilized another pragma "`array_partition complete dim = 0`" to completely flatten the arrays into registers so that the array access condition can be calculated and accessed in parallel. Finally, by applying a pragma "`pipeline`" in combination with these two optimizations, we successfully achieved the loop pipelining with II = 1 at the cost of resource utilization.

Note that irregular memory accesses frequently appear not only in SAT algorithms but also in a variety of database-oriented applications. Several existing works presented HLS approaches incorporating a set of pre-designed arbiters and buffers to resolve this issue. For example, [23] achieved up to 5.0-8.0× speedup with 2.5-6.8× resource overhead for database-oriented benchmarks. Hence, some readers would think that those arbiter approaches are also applicable to our target algorithm and our array-flattening approach is naive. Through holistic examinations on how to resolve this issue, we observed that the approach we employed is the most performance-effective as it achieved approximately 8,200× speedup with 23× resource overhead against the baseline implementation with no pragma – this speedup is infeasible by the arbiter method. Also, the readers can find that the speedup effect is nearly cancelled out by the resource overhead in [23]. We will further study the improvement of irregular memory access optimizations to suppress the resource overhead in future.

V. EXPERIMENTS AND RESULTS

This section first provides our experimental setup and then demonstrates the effectiveness of our work.

A. EXPERIMENTAL SETUP

We applied the algorithmic extensions explained in Sections III-B.1 and III-B.2 with hardware-aware optimization

techniques explained in Section IV to implement in total four versions of our AmoebaSAT solver as follows:

- **Ours-B**: The AmoebaSAT algorithm with the original bounceback rules (Section II-B and [12]). Among our SAT solvers, this version is regarded as the baseline design.
- **Ours-C**: The AmoebaSAT algorithm extended with *COLLAPSE* only (Section III-B.1).
- **Ours-H**: The AmoebaSAT algorithm extended with *HyperCONTRA* only (Section III-B.2).
- **Ours-CH**: The AmoebaSAT algorithm extended with both *COLLAPSE* and *HyperCONTRA* (Sections III-B.1 and III-B.2).

We used Xilinx Vivado HLS and Vivado v2016.3 for high-level and logic syntheses on a Zynq board (xc7z030ffv676-3) which was also used in [17]. We compared our solvers with three counterparts: the original AmoebaSAT algorithm running as software (hereafter **SW**), a hardware AmoebaSAT solver [17] that adopted a largely simplified algorithm[2] (hereafter **ISQED**), and a hardware WalkSAT solver [21][3] (hereafter **WalkSAT**). The initialization part of our work (i.e., Our-B/C/H/CH) and SW were evaluated on a Cortex-A9 of the same Zynq board at the clock frequency of 1 GHz.

While randomly generated instances have been widely used in the literature including [17], [21], our work targets real-life, IoT applications whose community structure is more hierarchical and sparse than random ones [20]. Therefore, we conducted two sets of evaluations using different types of instances in SATLIB [26]. In the first evaluation, in order to demonstrate the effectiveness of our work over state-of-the-arts, we used six randomly generated instances composed of 100 to 250 variables (specified with a name "`uf<variables#>-<index#>`") including ones that were also used in [17], [21]. Then, in the second evaluation, in order to show the efficiency of our work in handling real-life applications, we used three SAT-encoded flat graph colouring instances composed of 150 to 300 variables (specified with a name "`flat<vertices#>-<index#>`"; the vertices# represent the complexity of the graph colouring problem before encoding to SAT) since some real-life applications can be expressed in a graph colouring problem [16]. In both evaluations, we discuss the results of our work in terms of the performance (i.e., iterations#, clock frequency, and execution time), the resource usage (i.e., Slices# that represent the circuit area), and the area-delay-product (ADP)[4] which is calculated by the product of Slices# and the execution time. Since the iterations# to find a solution depend on the initialization, we ran each solver 100 times with random seeds and calculated the average iterations#.

[2]This algorithm removed the intermediate variables to directly determine the variable assignments of x's. Therefore, the unique feature in the original AmoebaSAT of "learning from the failures" was almost given away. Interested readers are referred to [17] for further details of the algorithm.

[3]Kanazawa *et al.* have presented extended versions of the hardware WalkSAT solver (e.g., [24], [25]) based on their earlier work [21]. While the work [21] may be still applicable to embedded/IoT applications, their recent target has been shifted to huge industrial applications. Therefore, comparing against [21] is the fairest to demonstrate the effectiveness of our work.

[4]This is a frequently-used metric to evaluate the efficiency of circuits. The smaller ADP indicates the better efficiency.

B. EVALUATION ON RANDOM INSTANCES

The first evaluation was conducted to compare our work and the state-of-the-art works for six randomly-generated instances. Table 2 tabulates the results of different AmoebaSAT-based SAT solvers, followed by the comparison against WalkSAT afterwards: the execution time of **SW** and the performance (iterations#, clock frequency and the execution time) and the area (Slices#) of **ISQED** and our work **Ours-B/C/H/CH**. In the table, the execution time reports the solution searching time only since the solution search will be repeatedly done according to the surrounding environment and sufficiently longer than the pre-processing which should be done once only at the beginning. Among the six evaluated instances, three instances (`uf100-0285`, `uf150-0100`, and `uf225-028`) were used in ISQED [17] (the unused instances show '-'). Cycles# in ISQED and Ours-B/C/H/CH both equal the iterations#.

As seen from the table, our hardware solvers are all significantly faster (39-39,351× in terms of the execution time) than SW even though SW runs on the processor core with the 4.8-13.7× higher clock frequency than our solvers. This indicates that our solvers can well extract the inherent parallelism. Compared with ISQED, Ours-C/H/CH achieved significant reduction in iterations# to find a solution. Besides Table 2, in order to clearly show the comparison between ISQED and our work, Fig. 8 describes the iterations# and Slices# of our work normalized by ISQED for the three commonly-used instances. The x-axis describes the instance names and the left and right y-axes show the Slices# and iterations#, respectively. Fig. 8 shows that our solvers are superior to ISQED in terms of the iterations# especially for larger instances at the cost of Slices# – our solvers achieved the iterations# reduction by 4-7× for `uf150-0100` and 33-38× for `uf225-028` with the 7-12× Slices# compared with ISQED. In addition, the results from Table 2 show that for the largest instance `uf225-028`, Ours-C/H/CH achieved the 2.12-2.67× better ADP than ISQED. These results demonstrate that our solvers can more efficiently search a solution by learning from the failure with the help of the bounceback rules.

Next as comparing the four versions of our solver, we find that thanks to the extensions in the bounceback rules presented in Section III, Ours-C/H/CH have significantly less iterations# than Ours-B. Although the extended bounceback rules induce some area overhead, this effect is negligible considering the significant cycles# reduction. Among the four versions, Ours-H achieved the least iterations# for two instances and Ours-C achieved the shortest execution time for three instances. Ours-CH which combined the two extensions did not achieve the best but is constantly good. In all solvers, the clock frequency degrades for larger instances as inter-unit wires tend to be critical and get longer (more detailed

TABLE 2. Synthesis and simulation results of SW [12] and four versions of Ours (the best iterations# and execution time for each instance are highlighted by gray cells).

	Instance	uf100-0285	uf150-0100	uf200-01	uf225-028	uf250-01	uf250-0100
	Variables#	100	150	200	225	250	250
	Clauses#	430	645	860	960	1,065	1,065
SW [12]	Exec. Time (ms)	3842.78	3027.13	1890.70	2307.95	4011.90	2974.78
ISQED [17]	Iterations#	85,192	364,755	–	541,955	–	–
	Clk Freq. (MHz)	239.1	199.1	–	176.1	–	–
	Exec. Time (ms)	0.36	1.83	–	3.08	–	–
	Slices#	542	899	–	1,309	–	–
Ours-B	Iterations#	567,224	2,607,976	1,417,492	618,973	83,591	4,813,976
	Clk Freq. (MHz)	209.0	159.2	85.8	153.5	78.7	63.1
	Exec. Time (ms)	2.71	16.38	16.52	4.03	1.06	76.29
	Slices#	4,780	6,727	10,951	11,938	14,277	13,586
Ours-C	Iterations#	411,600	54,359	1,511,725	16,572	11,120	142,482
	Clk Freq. (MHz)	205.7	151.4	102.4	110.95	109.1	95.9
	Exec. Time (ms)	2.00	0.36	14.76	0.15	0.10	1.49
	Slices#	5,619	7,957	10,936	10,108	11,790	12,976
Ours-H	Iterations#	222,731	92,158	187,062	14,188	10,551	2,347,038
	Clk Freq. (MHz)	106.4	101.7	103.7	106.8	73.1	68.1
	Exec. Time (ms)	2.09	0.91	1.80	0.13	0.14	34.43
	Slices#	6,165	8,241	11,748	12,963	14,179	14,382
Ours-CH	Iterations#	226,473	92,118	261,465	14,371	8,855	559,412
	Clk Freq. (MHz)	74.5	93.2	91.8	91.8	74.2	76.5
	Exec. Time (ms)	3.04	0.99	2.85	0.16	0.12	7.39
	Slices#	6,772	8,150	11,056	12,159	13,227	13,268

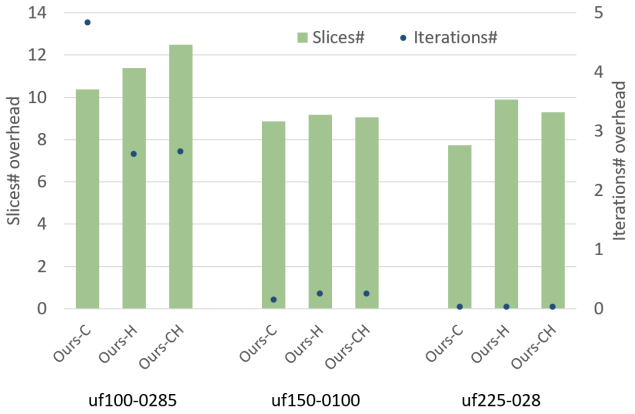

FIGURE 8. Slices# and iterations# of Ours-C/H/CH (normalized by ISQED [17]).

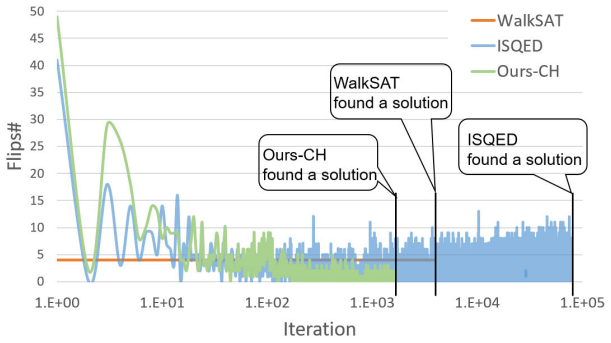

FIGURE 9. Variable flips# per iteration (for uf225-028).

discussions will be given in Section V-D). Suppressing the clock degradation will be thus a subject of our future work.

For one commonly-used instance uf225-028, **WalkSAT** was evaluated on Virtex2 in [21], where the circuit consumed 17,234 Slices and ran at 85.2MHz while taking on average 25.74 cycles per iteration. In order to exclude the device difference, based on the comparison done in [17], we made an estimation to compare our solvers and WalkSAT – if our solvers were implemented on the same Virtex2 device, we would achieve the 23.29-33.90× speedup with the 1.69-2.18× Slices# (i.e., the 3.24× ADP improvement) over WalkSAT. These results also demonstrate that our solvers are superior to WalkSAT.

To further demonstrate the efficiency of our work in terms of the solution search, we evaluated flips# per iteration of Ours-CH, ISQED, and WalkSAT for the instance uf225-028. We implemented ISQED by software to measure the flips# per iteration and the total iterations# to find a solution. Although the results of ISQED and Ours-CH were taken from a random run, this does not hinder the natural behavior of these methods. The results of WalkSAT were referred to [21], where the four-thread implementation was done to evaluate four clauses simultaneously. In Fig. 9, the x and y-axes represent the iteration and the flips#, respectively. Note that the x-axis is described in the logarithmic scale. As shown in the figure, while WalkSAT constantly flips four variables per iteration, ISQED and Ours-CH can flip more variables at once – in this evaluation, they both conducted more than 40 flips in the first iteration. When the variable assignments are far from a solution (i.e., in the earlier iterations of the solution search), they keep such large flips# because of the frequent bounceback signals on many variables. In ISQED, however, such aggressive flips continue even when variable assignments are getting closer to a solution (i.e., in the later iterations), resulting in unsatisfying

TABLE 3. Synthesis and simulation results of Ours-C for three GCP instances.

Instance	flat50-1	flat75-10	flat100-100
Variables#	150	225	300
Clauses#	545	840	1,117
Iterations#	4,066	6,360	19,184
Clk Freq. (MHz)	151.2	144.3	116.4
Exec. Time (ms)	0.027	0.044	0.164
Slices#	5,283	8,158	9,877

even stable variable assignments. This is why ISQED takes large iterations# to finally find a solution. Unlike ISQED, Ours-CH avoids useless flips in the later iterations by learning from the failure (i.e., considering the statuses of intermediate variables caused by the recent bounceback controls) and conducting flips not to degrade stable assignments, resulting in the significant iterations# reduction. In other words, *Ours-CH determines the flips# according to the number of unsatisfied constraints*.

In summary, we observe that while WalkSAT can flip only as many variables as the number of implemented threads in each iteration, AmoebaSAT can evaluate more flips. In our work, such feature of AmoebaSAT is further encouraged with the help of the extended bounceback controls so that useless flips# can be avoided unlike ISQED.

C. EVALUATION ON SAT-ENCODED GRAPH COLOURING INSTANCES

The second evaluation was conducted to show the efficiency of our work in handling real-life applications such as 5G network scheduling problems [16]. Because those problems can be potentially formulated as group colouring problems (GCPs),[5] we selected three instances that represent GCPs (`flat50-1`, `flat75-10`, and `flat100-1`). Since the vertices# in the GCPs (i.e., 50, 75, and 100) are tripled when encoding to SAT, the SAT-encoded instances that we solved have 150, 225 and 300 variables, respectively. We implemented **Ours-C** for these instances as we found that Ours-C achieves the best ADP for larger instances. We show the Slices# and the average iterations# over 100 runs in Table 3.

As well as the results in Table 2, the linear increase in Slices# and the clock degradation are seen along with the instance complexity (i.e., the variables# and clauses#). The execution time is 0.027-0.16ms, which indicate that our work can handle, for example, the 5G network scheduling problem composed of 50 to 100 nodes (covering 250-500 IoT devices [16]) considering that each node (i.e., fog access point) has to response to its respective IoT devices within an interval of 1ms. These results satisfy the requirement of the target application and demonstrate the practicality of our work.

[5]Each node or fog access point in the network can be represented as a vertex in GCPs.

D. DISCUSSIONS

As discussed in Section IV-B and demonstrated from the results in both Sections V-B and V-C, we found that the Slices# of our solvers linearly increase along with the instance size that is defined as the product of variables# N and clauses# M. The difference in the Slices# can be also explained by the different ratio of clauses# over variables# (around 4.3 and 3.7 in the random and GCP instances, respectively). These results indicate that our solvers are scalable and practical for the size of real-life IoT applications such as [16].

On the other hand, it is hard to explain the results of the other metrics (clock frequency and iterations#) between two sets of instances only by the ratio difference. For example, although the clock frequency is similar for the instances with 150 variables (`uf150-0100` and `flat50-1`), more degradation is observed in the random instances by increasing the instance size. Even though the largest GCP instance (`flat100-100`) has more variables (300 variables) than the largest random instances (225 variables), the latter's clock frequency is about 20MHz lower than the former's. In other words, the clock frequency is not affected only by the ratio (or the variables# and clauses#). This is also the case for the iterations#.

Therefore, to study in depth the instance features, we analyzed and compared the instances with the same variables# from the two groups (150 and 225 variables). We measured the dependent units# for which each unit needs to check its related rules in each iteration (hereafter connections#). Then we made a histogram between the connections# and the incidence as shown in Fig. 10. From the figure, it is obvious that the GCP instances have a considerably smaller variance with much more locality than the random instances – while the former have an average of 15 and up to 28 connections# per unit, the latter have an average of 126 and up to 253 connections# per unit. Such structured connections in the GCP instances form a sparsity and a hierarchy, called "community" [20].

The community structure well explains both the results of the clock frequency and iterations#. Since the dense connections make it difficult to layout the complex inter-unit buses, the random instances (with the high density) tend to have long critical paths and hence large clock degradation by increasing the instance size. Similarly, for the random instances, it is more difficult to find the variable assignments that can satisfy all units that are complicatedly dependent each other, resulting in more iterations# to find a solution.

From the above insights and findings, we can conclude that real-life applications (e.g., 5G network control [16]) tend to have the low density or the well-structured community. Intuitively, some real-life applications would have much lower density as the ratio of the SATLIB's GCP instances (i.e., 3.7) looks relatively high for real-life applications. Therefore, it is very useful and effective to take the community structure into account to further improve our AmoebaSAT solver. The community-aware implementation can be done

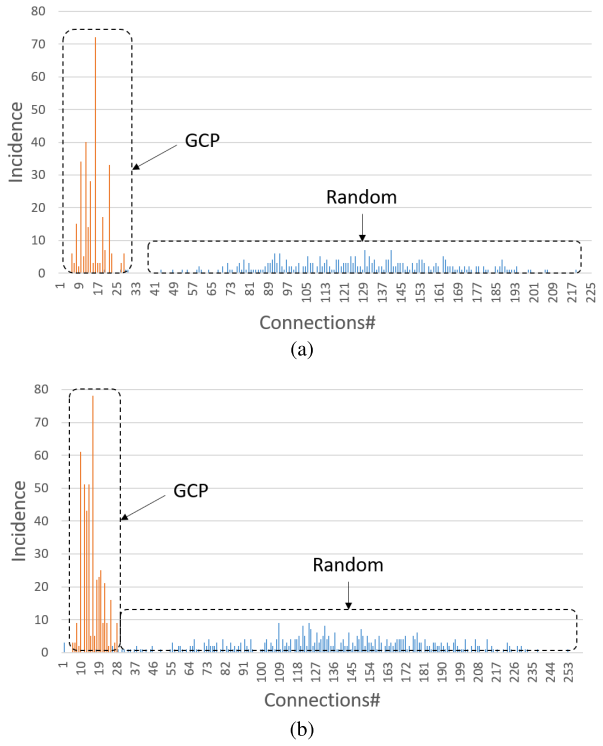

FIGURE 10. Connection# distribution of the random and GCP instances: (a) 150 variables and (b) 225 variables.

in both hardware and software; In the hardware, the inter-unit bus structure to share the bounceback signals can be hierarchically realized by following the inter-community structure. Also in the software, the strength of the bounceback signals defined by some local parameters (e.g., the ϵ value; uniquely set in this paper) may be adjusted according to the community structure. Such hardware/software co-design extensions will more usefully exploit the application features to quickly solve SAT-encoded problems.

VI. CONCLUSION

In this paper, we proposed an FPGA-based AmoebaSAT solver and implemented it using a hardware/software co-design approach. While the sequential pre-processing part of the algorithm can be realized as software to exploit the high clock frequency of the FPGA built-in microprocessor, the parallel computation component implements solution search and can be realized as hardware to fully extract the inherent parallelism of the algorithm. We then applied both software and hardware optimization techniques to find a solution in as few iterations as possible.

In our evaluations, we demonstrated that our work efficiently extracted the fine-grained parallelism to outperform two state-of-the-art FPGA-based solvers, especially for larger SAT instances. Also, we observed the scalability and practicality of our solver in terms of both area and solution search for SAT-encoded GCP instances representing real-life IoT applications. Furthermore, we conducted in-depth analysis on several instances and revealed the importance of considering the community structure typically present in practical applications.

In this paper, among the three versions of the bounceback rule extensions, the best bounceback rule extension depended on the SAT instance. Finding the best version for a specific application will be considered in our future work. Also, we will develop community-aware optimization techniques in both software and hardware to further accelerate the solution search.

ACKNOWLEDGMENT

The authors would like to thank Corey Waxman for useful discussions.

REFERENCES

[1] M. Bellmore and G. L. Nemhauser, "The traveling salesman problem: A survey," *Oper. Res.*, vol. 16, no. 3, pp. 538–558, Jun. 1968.
[2] S. Pettie and V. Ramachandran, "An optimal minimum spanning tree algorithm," *Automata, Lang. Program.*, vol. 1853, pp. 49–60, Jan. 2000.
[3] D. B. Shmoys, "Computing near-optimal solutions to combinatorial optimization problems," in *Combinatorial Optimization* (Discrete Mathematics & Theoretical Computer Science), vol. 20. Feb. 1996, pp. 335–397.
[4] M. Capra, R. Peloso, G. Masera, M. R. Roch, and M. Martina, "Edge computing: A survey on the hardware requirements in the Internet of Things world," *Future Internet*, vol. 11, no. 4, p. 100, Apr. 2019.
[5] R. D. Franceschi, M. Fischetti, and P. Toth, "A new ILP-based refinement heuristic for vehicle routing problems," *Math. Program.*, vol. 105, nos. 2–3, pp. 471–499, Feb. 2006.
[6] N. Absi and S. Kedad-Sidhoum, "MIP-based heuristics for multi-item capacitated lot-sizing problem with setup times and shortage costs," *RAIRO-Oper. Res.*, vol. 41, no. 2, pp. 171–192, Apr. 2007.
[7] S. Z. Selim and K. Alsultan, "A simulated annealing algorithm for the clustering problem," *Pattern Recognit.*, vol. 24, no. 10, pp. 1003–1008, Jan. 1991.
[8] T. Starkweather, D. Whitley, and K. Mathias, "Optimization using distributed genetic algorithms," in *Parallel Problem Solving from Nature* (Lecture Notes in Computer Science). New York, NY, USA: Springer-Verlag, vol. 496, Apr. 2006, pp. 176–185.
[9] A. Lucas, "Ising formulations of many NP problems," *Frontiers Phys.*, vol. 2, p. 5, Feb. 2014.
[10] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *Proc. Nat. Conf. Artif. Intell.*, vol. 1, Sep. 1999, pp. 337–343.
[11] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," in *Proc. Int. Conf. Theory Appl. Satisfiability Test. (SAT)*, Jun. 2012, pp. 16–29.
[12] M. Aono, S.-J. Kim, S. Kasai, H. Miwa, and M. Naruse, "Amoeba-inspired spatiotemporal dynamics for solving the satisfiability problem," *Adv. Sci., Technol. Environmentol.*, vol. B11, pp. 37–40, Jun. 2015.
[13] J. King, S. Yarkoni, J. Raymond, I. Ozfidan, A. D. King, M. M. Nevisi, J. P. Hilton, and C. C. McGeoch, "Quantum annealing amid local ruggedness and global frustration," *J. Phys. Soc. Jpn.*, vol. 88, no. 6, Jun. 2019, Art. no. 061007.
[14] M. Aramon, G. Rosenberg, E. Valiante, T. Miyazawa, H. Tamura, and H. G. Katzgraber, "Physics-inspired optimization for quadratic unconstrained problems using a digital annealer," *Frontiers Phys.*, vol. 7, p. 48, Apr. 2019.
[15] M. Yamaoka, T. Okuyama, M. Hayashi, C. Yoshimura, and T. Takemoto, "CMOS annealing machine: An in-memory computing accelerator to process combinatorial optimization problems," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Apr. 2019, pp. 1–8.
[16] A. Pratap, R. Gupta, V. S. Siddhardh Nadendla, and S. K. Das, "On maximizing task throughput in IoT-enabled 5G networks under latency and bandwidth constraints," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, Jun. 2019, pp. 217–224.
[17] K. Hara, N. Takeuchi, M. Aono, and Y. Hara-Azumi, "Amoeba-inspired stochastic hardware SAT solver," in *Proc. 20th Int. Symp. Qual. Electron. Design (ISQED)*, Mar. 2019, pp. 151–156.

[18] A. H. N. Nguyen, M. Aono, and Y. Hara-Azumi, "Amoeba-inspired hardware SAT solver with effective feedback control," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2019, pp. 241–244.
[19] M. Aono, S. Kasai, S.-J. Kim, M. Wakabayashi, H. Miwa, and M. Naruse, "Amoeba-inspired nanoarchitectonic computing implemented using electrical brownian ratchets," *Nanotechnology*, vol. 26, no. 23, Jun. 2015, Art. no. 234001.
[20] Z. Newsham, V. Ganesh, S. Fischmeister, G. Audemard, and L. Simon, "Impact of community structure on sat solver performance," in *Proc. Theory Appl. Satisfiability Test.*, vol. 8561, Jul. 2014, pp. 252–268.
[21] K. Kanazawa and T. Maruyama, "An approach for solving large SAT problems on FPGA," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 1, pp. 10:1–10:21, Dec. 2010.
[22] A. Luca, A. Ilyas, and A. Vlad, "Generating random binary sequences using tent map," in *Proc. Int. Symp. Signals, Circuits Syst. (ISSCS)*, Jun. 2011, pp. 1–4.
[23] G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang, "Architecture and synthesis for area-efficient pipelining of irregular loop nests," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 11, pp. 1817–1830, Nov. 2017.
[24] K. Kanazawa and T. Maruyama, "FPGA acceleration of SAT/Max-SAT solving using variable-way cache," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–4.
[25] K. Kanazawa and T. Maruyama, "An approach for solving SAT/MaxSAT-encoded formal verification problems on FPGA," *IEICE Trans. Inf. Syst.*, vol. E100.D, no. 8, pp. 1807–1818, Aug. 2017.
[26] *SATLIB—Benchmark Problems*. [Online]. Available: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

ANH HOANG NGOC NGUYEN received the B.E. degree in electronics and communications engineering from the Hanoi University of Science and Technology, Vietnam, in 2016, and the M.E degree in information and communications engineering from the Tokyo Institute of Technology, Japan, in 2018. She is currently pursuing the Ph.D. degree in information and communications engineering with the Tokyo Institute of Technology. Her research interests include high-level designs for embedded systems and bio-inspired computing. She has been a recipient of the Japanese Government Scholarship (MEXT), since 2016, and a member of ACM.

MASASHI AONO received the degree from the Faculty of Environment and Information Studies, Keio University, Japan, in 1999, and the Ph.D. degree from Kobe University, in 2004. He has been serving as a Tenured Associate Professor with Keio University, since 2017. His central research interest has been the information processing mechanisms of biological systems that adapt to dynamic and versatile environments. He has advanced his unique study on the development of amoeba-inspired computers, working consecutively as a Researcher at RIKEN, Earth-Life Science Institute of Tokyo Institute of Technology, and JST PRESTO. For this innovative work, he received the Young Scientists' Prize from the Minister of Education, Culture, Sports, Science and Technology of Japan, in 2017. His strong passion to accelerate the implementation of his own amoeba-inspired computing technologies to control soft robots motivated him to establish Amoeba Energy Co., Ltd., in 2018.

YUKO HARA-AZUMI (Member, IEEE) received the Ph.D. degree in information science from Nagoya University, Japan, in 2010. She was a JSPS Postdoctoral Research Fellow with Ritsumeikan University, from 2010 to 2012, during which she was also a Visiting Scholar at the University of California, Irvine, USA, and the Karlsruhe Institute of Technology, Germany. In 2012, she joined the Nara Institute of Science and Technology as an Assistant Professor. Since 2014, she has been with the Tokyo Institute of Technology, where she is currently an Associate Professor. Her research interests include system-level design automation, especially on high-level and logic synthesis, microprocessor architectures, and hardware/software co-design for the embedded/IoT systems. She serves as organizing and program committees of several premier conferences, including DAC, ICCAD, DATE, CASES, ASP-DAC, FPL, and so on.