Amoeba-Inspired Hardware SAT Solver with Effective Feedback Control

Abstract—**Bio-inspired domain-specific computing architectures are able to handle specific tasks such as image recognition and combinatorial optimization more quickly than general-purpose ones. In this paper, we focus on an amoeba-inspired algorithm, "AmoebaSAT," to solve Boolean satisfiability (SAT) problems. A hardware SAT solver is useful for a variety of control applications in Internet-of-Things edge-computing systems whose control constraints can be reduced to a SAT problem. We develop efficient AmoebaSAT solvers on an FPGA by realizing various feedback controls to find a solution quickly. To extract the inherent parallelism of the AmoebaSAT, a high-level design approach (i.e., high-level synthesis and its pragmas) is applied together with hardware-friendly code transformations/algorithmic extensions. We demonstrate that our FPGA-based AmoebaSAT solvers can achieve significant iterations reduction and speedup to find a solution compared with state-of-the-art SAT solvers. Furthermore, we show the effectiveness of our work in the scalability (i.e., resource utilization in FPGA) and the parallelism.**

I. Introduction

In the Internet-of-Things (IoT) era, a variety of IoT edge devices intelligently behave or control their target objects according to their installed sensors and the interactions with their neighboring devices [1]. One of essential requirements for those devices is "real-time processing." For this purpose, efficient computing on the devices to enable self-sustaining, instantaneous decisions (i.e., without directives from the Cloud) is a key, and thus processing by parallelized hardware rather than sequential software is suitable. Obviously, depending on the number and type of sensors and control-target objects, hardware designs of such devices have a different complexity of the constraints to be considered. Since the complexity and diversity of IoT applications are increasing, developing a new hardware design approach that can efficiently express and handle such constraints is crucial to encourage the further advance of the IoT society [2].

In these years, some works have developed dedicated hardware solvers for satisfiability (SAT) problems [3]–[7] or combinatorial optimization problems [8]–[10] by focusing on a fact that the control constraints of the IoT applications can be reduced to those general problems. Most of those works target huge industrial problems composed of thousands of variables and aim to accelerate a solution search by implementing an FPGA-based solver with a huge amount of caches/off-chip memories (i.e., not considering to be deployed on embedded/IoT devices). On the other hand, some works targeting IoT applications (with up to hundreds of variables [3], [7]) have employed SAT algorithms with the fine-grained parallelism and implemented highly-parallelized SAT solvers to efficiently find a solution. Particularly because it is known that bio-inspired algorithms can outperform human experts in various application domains, [5]–[7] have developed hardware SAT solvers based on an amoeba-inspired algorithm (AmoebaSAT [11]) and have outperformed a solver based on WalkSAT (a widely-used conventional SAT algorithm) [4].

In this paper, we propose a SAT solver that is also based on the AmoebaSAT algorithm but is more efficient and faster than [5]–[7]. We found that these existing works simplified the algorithm too much for the hardware implementation, incurring the inefficiency in repeating a useless search for complex problems (the details will be demonstrated in Section IV). Unlike these works, our work has two essential features to find a solution quickly: (1) exploiting the intermediate variables as well as the original algorithm to judge if the recent variable assignments were good or not and (2) making the feedback controls more effective to avoid getting far from a solution in the search space. Considering the deployment of our solver on an FPGA, we utilize a high-level design technology (i.e., high-level synthesis (HLS)) to enhance the design productivity and present HLS-aware optimization techniques that are helpful for various applications (e.g., irregular memory access). Our evaluation demonstrated that our work can achieve significant speedup (an average of 2,454× and up to 11,923×) against a software SAT solver and 33-38× reduction in the iterations taken to find a solution over state-of-the-art hardware SAT solvers while having the scalability in the circuit area (Slices#). Also, we showed in-depth analyses on how effectively variable assignments are evaluated (flipped) in our work thanks to the aforementioned two features.

The contributions of our work are as follows:

- We employ a hardware/software codesign approach to develop an FPGA-based SAT solver that is faster than state-of-the-arts.
- We exploit and extend a bio-inspired SAT algorithm (AmoebaSAT) with effective feedback controls (which are named "bounceback" controls) to efficiently search for a solution – aggressively flips variable assignments when the current assignments are far from a stable condition (i.e., a solution) and avoids useless flips when getting closer to a solution (i.e., most variables are stabilized).
- We exploit a high-level design approach (i.e., HLS) and HLS-aware optimization techniques (e.g., targeting irregular memory access) which are useful for other applications such as database-oriented ones.

The remainder of this paper is organized as follows: Section II briefly explains the AmoebaSAT algorithm. Section III presents high-level design techniques and algorithmic extensions. Section IV demonstrates the effectiveness of our work over state-of-the-arts. Section V concludes this paper.

II. AmoebaSAT: Amoeba-Inspired SAT Algorithm

A. SAT and Hardware SAT Solvers

The satisfiability problem is a problem to determine if there exists at least a variable assignment that satisfies a given formula (which we call "instance" hereafter). The satisfiability problem is particularly called a "Boolean satisfiability problem (SAT)," when the value of each variable is either TRUE ('1') or FALSE ('0'). SAT is NP-complete and significantly important not only in the computer science's theory but also in practical IoT applications. A SAT instance is often represented as a Conjunctive Normal Form (CNF), which is composed of conjunctions (AND operations; $\wedge$) of a number of clauses, each of which is expressed with disjunctions (OR operations; $\vee$) of multiple logical variables (or literals). For example, for an instance with two variables, $f = (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2})$, SAT solvers determine that f is *satisfiable* ($f = 1$) by $x_1 = x_2 = 0$. Otherwise (i.e., if there is no variable assignment to make $f = 1$), f is determined to be *unsatisfiable*. Although this can be solved in a brute force manner, it incurs an exponential complexity (i.e., 2^n combinations, where n represents the number of input variables). Hence, a variety of stochastic local search (SLS) algorithms that more quickly determine the satisfiability of instances have been developed, e.g., by applying pruning or heuristics to reduce the search space [3].

Most of such existing SAT algorithms have been developed only as software, mainly because of two reasons. First, most of those algorithms sequentially check and resolve a conflict, where for the same variable x, there exists a clause that requires $x = 1$ and another that requires $x = 0$ at the same time. Second, those algorithms had targeted industrial applications (e.g., software verification) with thousands of variables and a huge amount of conflict rules, requiring a large volume of memory. For such applications, solving SAT off-line by CPU has been sufficient [12]. However, in these years there is a need of developing hardware SAT solvers that can find a solution on-line for IoT edge-computing applications (mostly with up to hundreds of variables) [13]. Those hardware SAT solvers employ an SLS-based algorithm, where variables are flipped with some probability to check and resolve as many conflicting assignments as possible in parallel. As sketched in Fig. 1, such hardware SAT solvers [4], [5], [7], irrespective of instance/application-specific implementations [3], would be deployed on an IoT edge-computing system and continuously find a solution (corresponding to the control signals for the system $x_0, \cdots, x_i$) depending on the circumstance at the moment (corresponding to the sensor values $x_{i+1}, \cdots, x_N$). For the hardware SAT solvers, it is essential to accelerate the solution search by exploiting the parallelism of the baseline SAT algorithm at the granularity of variables or clauses.

B. AmoebaSAT Algorithm

Among SLS-based SAT algorithms, WalkSAT has been popular to realize hardware SAT solvers [3], [4], [14]. In each iteration, it flips one variable selected from clauses containing conflicts. Hence, its parallelism is bounded by the clause-level granularity. In contrast, a recently-developed SAT algorithm, called "AmoebaSAT," holds a finer-grained parallelism (i.e., the variable level) [11]. Because more acceleration can be expected by the finer-grained parallelism, in this work we develop a hardware/software-codesigned SAT solver based on AmoebaSAT. Furthermore, unlike the existing AmoebaSAT-based works [5]–[7] that largely simplified the internal structures (as described below), we exploit and extend the original algorithm that can learn from the failures to quickly find a solution (the extensions will be explained in Section III-B).

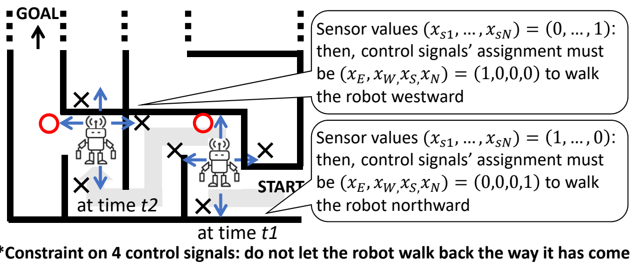

GOAL
Sensor values $(x_{s1}, \dots, x_{sN}) = (0, \dots, 1)$: then, control signals' assignment must be $(x_E, x_W, x_S, x_N) = (1,0,0,0)$ to walk the robot westward
Sensor values $(x_{s1}, \dots, x_{sN}) = (1, \dots, 0)$: then, control signals' assignment must be $(x_E, x_W, x_S, x_N) = (0,0,0,1)$ to walk the robot northward
START
at time $t2$
at time $t1$
*Constraint on 4 control signals: do not let the robot walk back the way it has come

Fig. 1. An example use-case of hardware SAT solvers (in case of generating four control signals x's according to their dependency (constraint) and $N+1$ sensor values x's)

TABLE I
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 volume of the unit (i, v) ($v \in \{0, 1\}$). 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 ($X_{i,v}$).
$L_{i,v} \in \{0, 1\}$	Control variable representing light stimulation for 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 best value depends on how the oscillation is generated.

AmoebaSAT was derived from the spatiotemporal deformation dynamics of an amoeba in its adaption to the surrounding environment. The amoeba is a single-cell organism with multiple pseudopod-like branches that can expand to search for food sources while avoiding aversive light stimuli. If a branch expands to an undesirable direction, a light is projected on the branch, and the branch shrinks so that the amoeba avoids the light and explores other dark conditions to continue the search – this feedback is called "bounceback." To reduce to a SAT problem with N variables, AmoebaSAT employs $2N$ units as analogues of expandable branches. Definitions and descriptions of variables used in AmoebaSAT are summarized in Table I.

To express the amoeba's ability to learn from the failures, the AmoebaSAT algorithm [11] sets a pair of ternary variables $X_{i,0}$ and $X_{i,1}$ to determine the assignment of x_i. At the iteration t, the variables $x_i(t)$ and $X_{i,v}(t)$ ($v \in \{0, 1\}$ hold

Eqn. (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 unit (i, v) represents an expandable branch to set $x_i = v$, and its equilibrium volume $X_{i,v}$ is based on its resource supply $Y_{i,v}$, which determines whether the unit (i, v) should expand in the next iteration or not. The state transition of $Y_{i,v}$ depends on the dynamics of the fluctuated variable $Z_{i,v}$[1] 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 defined as follows:

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

Eqn. (3) expresses that AmoebaSAT supplies a resource to the unit (i, v) based on its control signal $L_{i,v}$ (the concept and equation of $L_{i,v}$ will be explained in detail in Section II-C). When there is no light stimuli (i.e., $L_{i,v} = 0$), the unit (i, v) can freely increase its volume. In other words, the resource can be supplied as long as the fluctuation $Z_{i,v}$ does not exceed a value of error occurrence (i.e., $1 - \epsilon$). 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* [15] to generate oscillatory dynamics considering the simplicity and suitability for implementing fixed-point operations on FPGA (the details will be explained in Section III-B).

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

C. Bounceback Rules

Light stimulation $L_{i,v}$ used in Eqn. (3) is an essential concept in the AmoebaSAT algorithm to find if the variable assignments in the previous iteration $(t-1)$ incurred a conflict or not. In other words, the AmoebaSAT *learns from the failures* through L's. If there is a conflict, the "bounceback control" is imposed on the corresponding units to cut off the resource supply in the next iteration (i.e., $L_{i,v} = 1$). Thus, this bounceback control is a set of rules constructed from the clauses of the target SAT instance as expressed in Eqn. (6).

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

where (P, Q) represents two contradictory units that cannot be true at the same time and B_{ON} represents the set of the bounceback rules (i.e., turning L's on). B_{ON} is composed of three types of rules, named $INTRA$, $INTER$ and $CONTRA$, i.e., $B_{ON} = INTRA \cup INTER \cup CONTRA$. $INTRA$ prohibits an inconsistent assignment (i.e., 0 and 1) of each variable x_i at the same time. Note that this paper expresses two inconsistent assignments P and Q as (P, Q). Then, $INTRA$ can be defined as follows:

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

$INTER$ constraints the variable assignments in a clause. For example, for a clause $C = (x_1 \vee x_2 \vee \overline{x_3})$, if $x_1 = 0$ and $x_2 = 0$ hold, x_3 needs to be 0 to satisfy the clause C. Thus, while the unit $(3, 0)$ should be expanded, the unit $(3, 1)$ should be bounced-back. In other words, $INTER$ is determined to clear the unsatisfied assignments in each clause C_k:

$$INTER \ni \begin{cases} (P, (i, 0)) & (\text{if } x_i \in C_k), \\ (P, (i, 1)) & (\text{if } \overline{x_i} \in C_k). \end{cases} \tag{8}$$

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

$$P \ni \begin{cases} (j, 0) & (\text{if } v_j \in C_k), \\ (j, 1), & (\text{if } \overline{v_j} \in C_k). \end{cases} \tag{9}$$

Finally, $CONTRA$ manages the contradiction between units involved in multiple clauses. For example, for two clauses $C_1 = (\overline{x_3} \vee x_4 \vee \overline{x_1})$ and $C_2 = (x_3 \vee x_4 \vee \overline{x_1})$, if $x_4 = 1$ and $x_1 = 1$, $x_3 = 1$ needs to be held to satisfy C_1. However, $x_3 = 0$ also needs to be held to satisfy C_2. These requirements will lead the bounceback on both $X_{3,0}$ and $X_{3,1}$, i.e., prevent x_3 from being either 0 or 1. $CONTRA$ finds out such contradictions based on $INTER$. For each i, we check the set $INTER$ to generate two sets $\mathbf{P}_{i,0} = \{ P \mid (P, \{(i, 0)\}) \in INTER\}$ and $\mathbf{P}_{i,1} = \{ P \mid (P, \{(i, 1)\}) \in INTER\}$, and define the set $CONTRA$ by joining each element $(P_{i,0}, P_{i,1})$ in a product set of $\mathbf{P}_{i,0}$ and $\mathbf{P}_{i,1}$ as follows:

$$\forall (P_{i,0}, P_{i,1}) \in \mathbf{P}_{i,0} \times \mathbf{P}_{i,1} (\ CONTRA \ni (P_{i,0} \cup P_{i,1}, P_{i,0} \cup P_{i,1}) \ ) \tag{10}$$

As a summary of the aforementioned equations, Fig. 2 describes the simplified pseudo code of the AmoebaSAT algorithm. We also illustrate how the intermediate variables work for a variable x_i. As shown in the example of Fig. 3, we assume that x_i is initialized as 0. X's, Y's, and L's are also all initialized as 0 at the iteration $t = 0$ (Lines 4-5). Then, at the iteration $t = 1$, because $Z_{i,0}$ exceeded $1 - \epsilon$, $Y_{i,0} = 0$ is set according to Eqn. (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, $Y_{i,1} = 1$ increments $X_{i,1}$ 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 Eqn. (1). Because of $X_{i,1} = 1$, $INTRA$ prohibits to supply the resource to $X_{i,0}$, resulting in $L_{i,0} = 1$ and $L_{i,1} = 0$ in the next iteration $t = 2$ (Lines 6-10). 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 6-10) is iteratively performed until a solution is found (i.e., when all units satisfy the condition that:

[1]For $2N$ units, $Z_{i,v}$'s are independent each other.

```
AmoebaSAT
01: Input: a target instance $f$, the maximum iteration $t_{max}$
02: $t = 0$
03: Construct the bounceback rules $B_{ON}$ by Eqn. (7), (8), & (10)
04: For $(i, v) = (1, 0)$ to $(N, 1)$
05:     Initialize $X_{i,v}(0) = Y_{i,v}(0) = L_{i,v}(1) = 0$
        Initialize $Z_{i,v}(0)$ with a random value in (0.0;1.0)
         and $Z_{i,v}(0) \neq 0.5$
06: While $t < t_{max}$ do
07:     Obtain $x_i(t)$ by Eqn. (1)
08:     If $f = 1$, a solution is found. Return 'Found'.
09:     Calculate $X_{i,v}(t)$, $Y_{i,v}(t)$, $Z_{i,v}(t)$, and $L_{i,v}(t+1)$
        by Eqn. (2), (3), (5), & (6)
10: End While
11: Return 'Not found'.
```

Fig. 2. A pseudo code of the AmoebaSAT algorithm

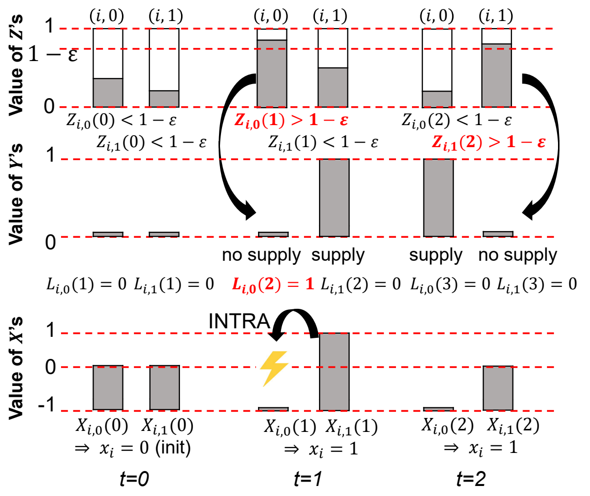


Fig. 3. An example of the status changes in intermediate variables

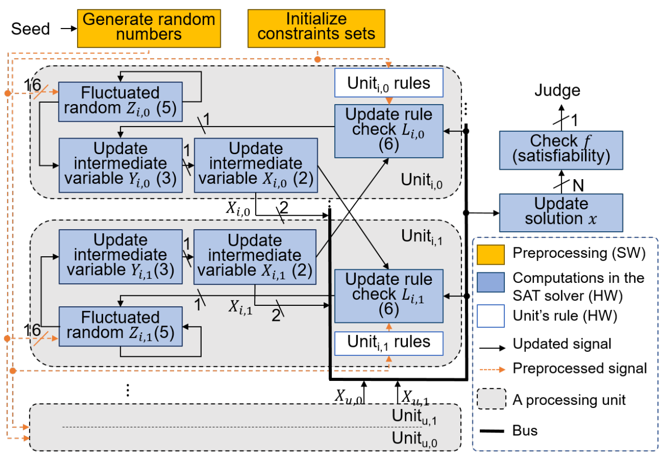
$\forall(i, v), X_{i,v} = 1 \iff L_{i,v} = 0$ or $X_{i,v} \leq 0 \iff L_{i,v} = 1$; Line 8) or the maximum iteration t_{max} is reached (Line 11).

As explained in the previous section, AmoebaSAT considers the updates of all variables in parallel and thus has the finer-grained parallelism than conventionally used SAT algorithms (e.g., WalkSAT) that update a single variable per iteration. In this work, we aim at developing a faster FPGA-based SAT solver than state-of-the-arts by exploiting two important features of the AmoebaSAT algorithm: (1) *the variable-level parallelism* and (2) *the ability of learning from the failures (i.e., using intermediate variables)*. These two features encourage an effective solution search by flipping as many variables as possible per iteration when the current variable assignments are far from a solution (i.e., especially at earlier iterations). Also, through an algorithmic extension (described in Section III-B), our solver can avoid useless flips as getting closer to a solution (i.e., at later iterations).

III. High-Level Design of Hardware AmoebaSAT Solver

A. Overview

Our SAT solver is realized in a hardware/software codesign manner as depicted in Fig. 4: while the initialization parts



Seed
Generate random numbers
Initialize constraints sets
16
Fluctuated random $Z_{i,0}$ (5)
Update intermediate variable $Y_{i,0}$ (3)
Update intermediate variable $X_{i,0}$ (2)
1
Unit$_{i,0}$ rules
Update rule check $L_{i,0}$ (6)
Unit$_{i,0}$
$X_{i,0}$
2
Unit$_{i,1}$
Update intermediate variable $Y_{i,1}$(3)
Update intermediate variable $X_{i,1}$ (2)
Update rule check $L_{i,1}$ (6)
$X_{i,1}$
Fluctuated random $Z_{i,1}$(5)
Unit$_{i,1}$ rules
$X_{u,0}$
$X_{u,1}$
Unit$_{u,1}$
Unit$_{u,0}$
Judge
Check f (satisfiability)
N
Update solution x
Preprocessing (SW)
Computations in the SAT solver (HW)
Unit's rule (HW)
Updated signal
Preprocessed signal
A processing unit
Bus

Fig. 4. An overview of the proposed SAT solver

that are performed only once at the beginning are realized as software, the iteratively performed parts are all designed as hardware. Each unit of AmoebaSAT is considered as a processing unit updating its own intermediate variables in each iteration. A shared bus is used to deliver the equilibrium information X's to other related units to update their bounceback control signals L's. Since many of today's FPGA devices support a hard microprocessor core [16], we target such an FPGA device for the implementation and utilize the high-level design technology (i.e., HLS) to realize the hardware parts on the reconfigurable fabrics of the FPGA. Although in this paper we utilize a Xilinx FPGA device and Vivado HLS, our work is not limited to the specific FPGA device or HLS tool.

The original AmoebaSAT algorithm already well-matches with hardware implementation due to its inherent parallelism. To fully enjoy the parallelism, however, we applied several hardware-conscious optimizations and extensions. Thus, hereafter, our descriptions focus on those design techniques we applied to the hardware parts.

B. Extensions of Effective Bounceback Controls

Although the bounceback controls in [11] 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 rules by observing how the values of X's change before and after the bounceback is applied. Due to the dependency between multiple variables, the bounceback controls are mostly activated by $INTER$ and $CONTRA$. Therefore, to let our SAT solver more quickly find a solution, we extended these rules by hardware-conscious ways that realize more direct controls to the statuses of intermediate variables, leading to less cycle counts.

1) COLLAPSE: Fig. 5(a) describes how $INTER$ sets L's when unsatisfactory variable assignments in a clause are found. If $X_{1,0}$, $X_{2,1}$, and $X_{3,0}$ are set to 1 for a clause $C = (x_1 \vee \overline{x_2} \vee x_3)$, according to Eqn. (6), L's are set to 1 not only to these units but also to their counterpart units. 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 the iteration $t + 1$. This indicates that $X_{1,0}$, $X_{2,1}$, and

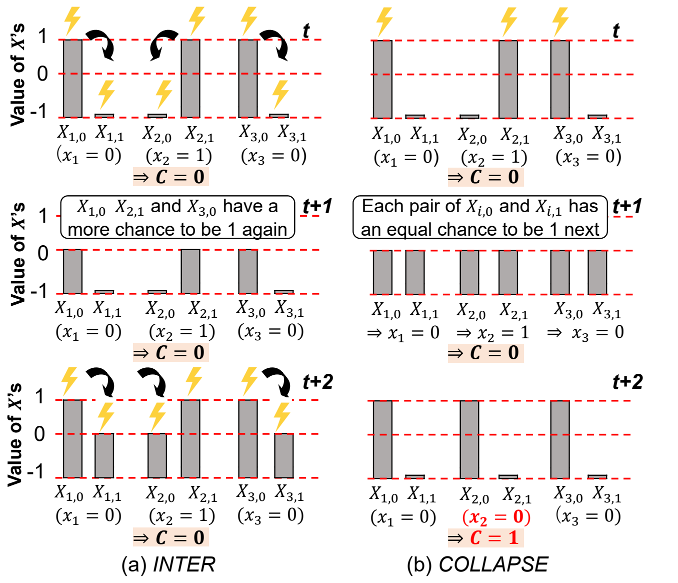


Fig. 5. Comparison of $INTER$ and $COLLAPSE$ (for $C = (x_1 \vee \overline{x_2} \vee x_3)$)

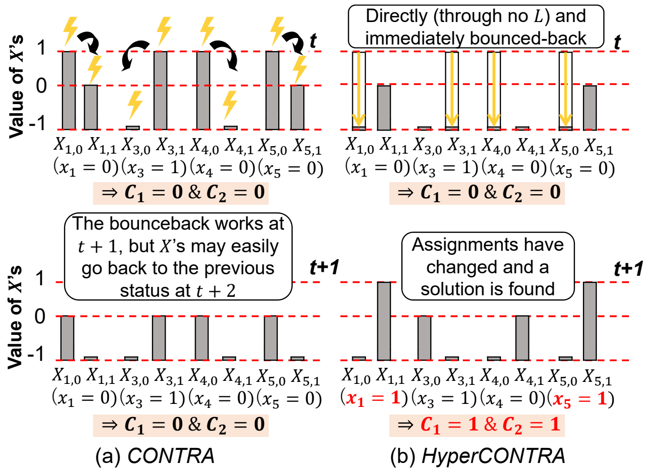


Fig. 6. Comparison of $CONTRA$ and $HyperCONTRA$ (for $C_1 = (x_1 \vee \overline{x_2} \vee \overline{x_3})$ and $C_2 = (x_2 \vee x_4 \vee x_5)$)

$X_{3,0}$ can become 1 again more easily than their counterparts, leading to repeat the same behaviors at the iteration $t+3$ and delaying to find a solution.

We thus propose to add a new bounceback rule, $COLLAPSE$, to mitigate such back-and-forth behaviors. While $INTER$ focuses only on when to turn L's on, $COLLAPSE$ considers when to turn L's *off*. For the same example, as shown in Fig. 5(b), when unsatisfactory variable assignments are found, it collapses part of $INTER$ by *disabling the bounceback to the counterpart units* at the iteration t. This results in the situation that each pair of $X_{i,0}$ and $X_{i,1}$ has an equal chance to become 1, leading to a higher probability to change the variable assignments and satisfy the clause C at the iteration $t+2$. $COLLAPSE$ can be derived based on $INTER$ and formulated as follows:

$$\begin{aligned} COLLAPSE \ni (\{(i,0)|x_i \in C_k\} \cup \{(i,1)|\overline{x_i} \in C_k\}, \\ \{(i,1)|x_i \in C_k\} \cup \{(i,0)|\overline{x_i} \in C_k\}) \end{aligned} \quad (11)$$

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

$$L_{i,v}(t) = \begin{cases} & (P,Q) \in B_{ON} \ \& \ (P,Q) \notin B_{OFF} \\ 1, & \text{such that } (i,v) \in Q \\ & \text{and } \forall (j,u) \in P, X_{j,u}(t-1) > 0 \\ 0, & \text{otherwise} \end{cases} \quad (12)$$

Note that B_{ON} works similarly to the original AmoebaSAT and B_{OFF} supplementarily works to encourage more efficient changes in the statuses of intermediate variables.

2) HyperCONTRA: Fig. 6(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. If $x_1 = x_4 = x_5 = 0$ and $x_3 = 1$ are set, x_2 needs to be 0 and 1 to satisfy C_1 and C_2, respectively, at the same time. Here $CONTRA$ happens, and L's are set to 1 not only to these units but also to their counterpart units by $INTRA$. Although all X's then decrease their volume at the iteration $t+1$, the clauses are not satisfied yet since the variable assignments have not changed yet. Reviewing the definitions of the intermediate variables, the readers will find that it takes another two iterations or more to change the variable assignments (i.e., the clauses can be satisfied no earlier than at the iteration $t+3$).

We found that the bounceback controls through L's are indirect and inefficient when $CONTRA$ happens. Therefore, we extend $CONTRA$, which we call $HyperCONTRA$, to powerfully change the statuses of X's by setting the corresponding X's to -1 immediately (i.e., in the current iteration). This extension also stops setting L's to 1 for the X's of the counterparts. These extensions can encourage the change of variable assignments quickly. Then, a solution may be found in the next iteration at the earliest as depicted in Fig. 6(b). The update on Eqn. (2) by introducing $HyperCONTRA$ is as follows:

$$X_{i,v}(t) = \begin{cases} -1, & (i,v) \in B_{ON}^H \\ X_{i,v}(t-1)+1, & (i,v) \notin B_{ON}^H \ \& \ Y_{i,v}(t) = 1 \\ & \& \ X_{i,v}(t-1) < 1 \\ X_{i,v}(t-1)-1, & (i,v) \notin B_{ON}^H \ \& \ Y_{i,v}(t) = 0 \\ & \& \ X_{i,v}(t-1) > -1 \\ X_{i,v}(t-1), & \text{otherwise} \end{cases} \quad (13)$$

where $B_{ON} = INTRA \cup INTER$[2] and $B_{ON}^H = HyperCONTRA$. Note that while $HyperCONTRA$ is defined similarly to $CONTRA$, the variables to be updated are different (i.e., $CONTRA$ updates L's, and $HyperCONTRA$ directly updates X's).

Since the extensions in Sections III-B1 and III-B2 are orthogonal, they can be both applied; $B_{ON} = INTRA \cup INTER$, $B_{OFF} = COLLAPSE$, and $B_{ON}^H = HyperCONTRA$.

[2]This update of B_{ON} applies to Eqn. (6).

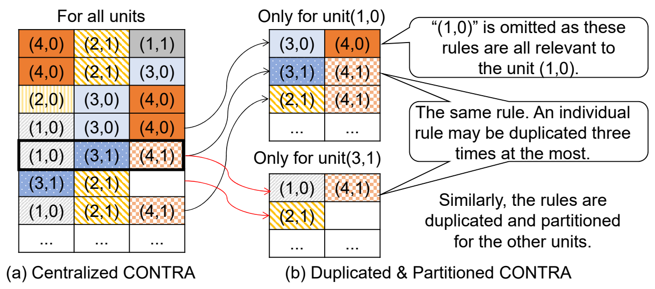

For all units
(4,0) (2,1) (1,1)
(4,0) (2,1) (3,0)
(2,0) (3,0) (4,0)
(1,0) (3,0) (4,0)
(1,0) (3,1) (4,1)
(3,1) (2,1)
(1,0) (2,1) (4,1)
Only for unit(1,0)
(3,0) (4,0)
(3,1) (4,1)
(2,1) (4,1)
Only for unit(3,1)
(1,0) (4,1)
(2,1)
"(1,0)" is omitted as these rules are all relevant to the unit (1,0).
The same rule. An individual rule may be duplicated three times at the most.
Similarly, the rules are duplicated and partitioned for the other units.
(a) Centralized CONTRA
(b) Duplicated & Partitioned CONTRA

Fig. 7. Localization of the Bounceback Rules

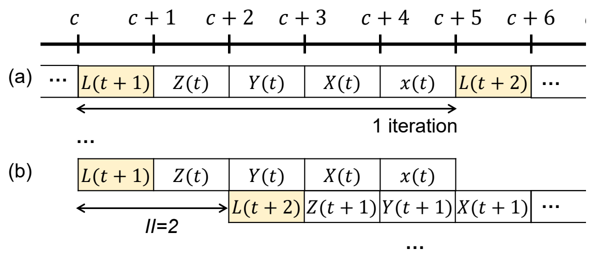

c $c+1$ $c+2$ $c+3$ $c+4$ $c+5$ $c+6$
(a)
$L(t+1)$ $Z(t)$ $Y(t)$ $X(t)$ $x(t)$ $L(t+2)$
1 iteration
(b)
$L(t+1)$ $Z(t)$ $Y(t)$ $X(t)$ $x(t)$
$L(t+2)$ $Z(t+1)$ $Y(t+1)$ $X(t+1)$
II=2

Fig. 8. Loop optimizations: (a) the baseline and (b) Pipelining

C. Optimizations of Operational Types and Definition

It is well-known that floating-point operations consume a lot of hardware resources. 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, where in total 16 bits are used (composed of a 1-bit sign and a 15-bit fractional part since Z's are in the range of $(0.0,\ 1.0)$). We also minimized the bitwidth of the other variables (two bits for X's and one bit for the others) using a pragma provided by the high-level synthesis tool we used (Xilinx Vivado HLS). Since these techniques are frequently used and have been presented in a number of literature (e.g., [17], [18]), we omit the details.

To save the hardware utilization, we realized the oscillation for Z's by replacing a logistic map adopted in [11] with a tent map as already explained in Eqn. (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 simulations 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 IV.

D. Rule Localization

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 [19]. Therefore, we partially duplicate the bounceback rules so that each unit has its related rules in the local lookup table (as shown in Fig. 4). The concept of this rule localization through duplication and partitioning is illustrated in Fig. 7. As shown in the figure, all units can in parallel check their rules at the cost of resource utilization.

E. Loop Optimizations

The intermediate variables and bounceback rules are individually implemented as an array and iteratively updated in a loop. Loop pipelining is one of the most resource-and-performance-efficient optimizations and has been widely applied to HLS-based designs. However, as shown below (which is a function of turning L's on according to $INTER$ represented in f), the arrays of the intermediate variables and bounceback rules (e.g., X and L) are accessed according to the indices reference (or calculation) because which units are bounced-back or get the resource supply are determined by the current assignments of the other related variables. Therefore, unlike regular memory accesses whose patterns (or intervals) are statically analyzable, such irregular memory accesses are a bottleneck of applying the loop pipelining.

```
void L_on_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;
}}}
```

Another bottleneck of parallelization is the sequential memory accesses when the arrays are mapped to the FPGA built-in memories.

This work utilized the pragmas provided by the Xilinx Vivado HLS to resolve the bottlenecks caused by the irregular memory access. We utilized the `pipeline` pragma along with the `array_partitioning dim=0` pragma to achieve the optimizations shown in Fig. 8; the former pipelines the loop, and the latter flattens the arrays into registers. By combining them, the array accesses can be statically determined and parallelized at the cost of resource utilization. In this work, with a help of the array flattening, we achieved the initiation interval (II) with every two cycles, i.e., `II=2`.

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, one of the state-of-the-arts achieved up to 5.0-8.0× speedup with 2.5-6.8× resource overhead for database-oriented benchmarks [20]. 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 (the details are omitted due to the space limitation), 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 [20]. We will further study the improvement of irregular memory access optimizations to suppress the resource overhead in future.

IV. Experiments

A. Setup

We applied the optimization techniques presented in Sections III-C, III-D, and III-E to the HLS-input `C` program of the AmoebaSAT algorithm and compared our HLS-generated SAT solvers against three state-of-the-art SAT solvers (a software solver and two hardware solvers): (1) the original AmoebaSAT implemented as software [11], (2) a WalkSAT-based hardware solver [4][3], and (3) a simplified hardware AmoebaSAT solver with no ability of learning from the failures [7]. Hereafter, they are specified as "**SW**", "**WalkSAT**", and "**Simple**", respectively. Since we extended the original AmoebaSAT algorithm in two ways (review Section III-B), we realized in total four versions of our SAT solver as follows:

- **Ours-B**: Employs the original AmoebaSAT presented in Section II-B. Among our SAT solvers, this is regarded as the baseline design.
- **Ours-C**: Employs the AmoebaSAT algorithm extended with $COLLAPSE$ only (Section III-B1).
- **Ours-H**: Employs the AmoebaSAT algorithm extended with $HyperCONTRA$ only (Section III-B2).
- **Ours-CH**: Employs the AmoebaSAT algorithm extended with both $COLLAPSE$ and $HyperCONTRA$ (Sections III-B1 and III-B2).

Recall that we target embedded/IoT applications whose instance has up to hundreds of variables, we selected six instances with 100 to 250 variables from SATLIB[4]. The hardware SAT solvers were evaluated in terms of the circuit area (i.e., resource utilization) and performance (i.e., iterations, clock frequency, and execution time). We used Xilinx Vivado HLS and Vivado v2016.3 for high-level and logic syntheses, respectively, targeting an FPGA zynq board (xc7z030ffv676-3) which was also used in [7]. The software SAT solver (i.e., SW) was evaluated on a Cortex-A9 (implemented on the same Zynq board) at the clock frequency of 1 GHz. Moreover, to demonstrate the parallelism that our SAT solvers achieved, we evaluated the number of flips per iteration.

B. Results

We first compare the four versions of **Ours** against SW and Simple as tabulated in Table II. The first and second rows present the information on the instances. While only the execution time is described for SW[5], synthesis results (i.e., Slices#[6] and clock frequency), simulation results (i.e., iterations#), and the execution time are described for each hardware solver. Since the iterations# taken to find a solution depends on the initialization, the iterations# were obtained by an average of 100 runs. Note that while Simple works in one cycle per iteration, **Ours** all work in two cycles per iteration.

From the table, we find that thanks to the sufficient parallelism realized in hardware, **Ours** all achieved significant speedup (an average of 2,454× and up to 11,923×) against SW in spite of the much higher clock frequency of the Cortex-A9. The three extended versions of **Ours** also achieved significant iterations# reduction than Simple, especially for larger instances (e.g., 33-38× reduction for `uf225-028`) – although Simple was the best for the smallest instance, these extended versions of **Ours** outperformed Simple for the other instances which are more practical. Although **Ours-B** was still slower than Simple for `uf225-018` due to the circuitous bounceback, **Ours-B** may reduce iterations# for further larger instance as the gap between Simple and **Ours-B** becomes smaller for larger instances. Among the three extended versions, **Ours-C/H/CH** achieved the best performance for three/two/one instance(s). Although none of them is uniformly the best, we found that **Ours-CH**, which integrates both of the two extensions, is constantly quite good for most instances. The best extension should depend on the feature of the instance (i.e., the community structure or complexity [19]), which we will intensively study and handle in our future work. Contrary to the significant speedup, we also find that **Ours** all consumed large Slices# due to the duplication of the bounceback rules. However, the overhead is smaller than the effects of iterations# reduction and speedup and is almost constant independent of the variables# and the extension version. Also, the Slices# linearly increase with the variables#. Thus, we can say that **Ours-C/H/CH** are scalable and useful for practical applications. The suppression on the clock degradation is another subject of our future work (e.g., by increasing the pipeline stages, we may achieve a better clock frequency without sacrificing the loop parallelism).

Among the instances in the table, `uf225-028` was also utilized for the evaluation in [4], where the results of 17,234 Slices#, 85.2MHz, and 25.74 cycles per iteration were achieved on Virtex2. According to the comparisons done in [7], we found that if the same Virtex2 device was used for **Ours-C/H/CH**, they would achieve on average 6.34-9.16× speedup with approximately 1.92-2.08× Slices# overhead compared with [4]. These three designs also achieved the better area-delay-product (ADP), where **Ours-CH** makes the best ADP improvement of 3.69×. These results also well demonstrate the effectiveness of our solvers over another state-of-the-art based on a different SAT algorithm (WalkSAT) [4].

To further evaluate the effectiveness of the parallelism our work achieved, we evaluated flips# per iteration taken by WalkSAT [4], Simple [7] and **Ours-CH**. In Fig. 9, the x and y-

[3]Kanazawa et al. have presented extended versions of the hardware WalkSAT solver (e.g., [14]) based on their earlier work [4]. While the work [4] may be still applicable to embedded/IoT applications, their recent target has been shifted to huge industrial applications. Therefore, comparing against [4] is the fairest to demonstrate the effectiveness of our work.

[4]Among a variety of instances in SATLIB, the instances with 250 variables are the largest. They are all specified in the CNF format and named as "`uf<variables#>-<index#>`," where <variables#> and <index#> are expressed by three and three-four digits, respectively.

[5]SW takes the same iterations# as **Ours-B**.

[6]DSPs and BRAMs are not used in **Ours** thanks to the optimizations described in Sections III-C and III-E.

TABLE II
SYNTHESIS AND SIMULATION RESULTS OF **SW** [11], **Simple** [7], AND FOUR VERSIONS OF **Ours**: "–" MEANS THAT SIMPLE WAS NOT EVALUATED FOR THE CORRESPONDING INSTANCE IN [7]. 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`
	Clause#	430	645	860	960	1,065	1,065
SW	Exec. Time (ms)	3,842.78	3,027.13	1,890.70	2,307.85	4,011.90	2,974.78
Simple [7]	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)	106.5	91.9	76.9	70.5	73.8	75.8
	Exec. Time (ms)	10.65	56.76	36.87	17.56	2.27	127.02
	Slices#	4,436	6,520	10,235	11,613	12,961	12,991
Ours-C	Iterations#	411,600	54,359	1,511,725	16,572	11,120	142,482
	Clk Freq. (MHz)	113.0	100.5	62.5	77.2	66.1	76.7
	Exec. Time (ms)	7.28	1.08	48.38	0.59	0.34	3.72
	Slices#	4,385	6,550	10,356	11,395	12,990	13,017
Ours-H	Iterations#	222,731	92,158	187,062	14,188	10,551	2,347,038
	Clk Freq. (MHz)	102.6	72.1	59.7	67.8	56.5	54
	Exec. Time (ms)	4.34	2.56	6.27	0.42	0.37	86.93
	Slices#	5,259	8,032	12036	12364	17,183	15,453
Ours-CH	Iterations#	226,473	92,118	261,465	14,371	8,855	559,412
	Clk Freq. (MHz)	88.7	74.2	55.3	70.7	54.0	54.4
	Exec. Time (ms)	5.11	2.48	9.46	0.41	0.41	20.57
	Slices#	5,356	7,437	13,794	12,183	15,453	14,653

axes represent iterations# and flips#, respectively. Note that the x-axis is described in the exponential scale. We referred [4] to obtain the iterations# of WalkSAT and implemented Simple by software to measure the flips# per iteration and iterations# to find a solution. The results of the AmoebaSAT-based solvers (Simple and **Ours-CH**) were taken from a random run, but this does not hinder the natural behavior of these methods. While WalkSAT [4] was implemented to evaluate four clauses simultaneously and thus constantly flips four variables in each iteration, Simple and **Ours-CH** can flip more variables at once – in case of `uf225-028`, 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 large flips because of the frequent bounceback signals on many variables. In Simple, however, such aggressive flips continue even when getting closer to a solution (i.e., in the later iterations), resulting in unsatisfying some stable variable assignments. This is why Simple takes quite large iterations# to finally reach to a solution. Unlike Simple, **Our-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 demonstrated the considerable speedup (an average of 2,454× and up to 11,923×) in all versions of **Ours** against SW and the significant iterations# reduction and speedup for large instances in **Ours-C/H/CH** against Simple [7] and WalkSAT [4]. Although the current implementations have non-negligible Slices# overhead, the scalability along with the variables# can be found. Also, we revealed that **Ours-C/H/CH** achieve the better parallelism than WalkSAT by allowing more flips# at once and can also avoid useless flips to find a solution more quickly than Simple.

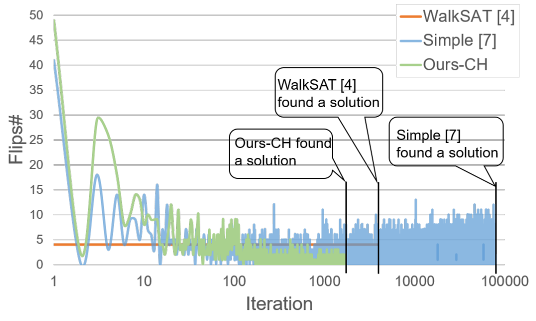


Fig. 9. Variable flips# per iteration (for `uf225-028`)

V. CONCLUSIONS

In this paper, we proposed an FPGA-based SAT solver using a high-level design approach (HLS). By exploiting and extending a bio-inspired SAT algorithm, AmoebaSAT, which can find a solution quickly with the effective feedback controls and utilizing HLS-aware optimization techniques, our SAT solvers achieved significant iterations# reduction and speedup to reach a solution more quickly than state-of-the-arts. By in-depth analyses on the solution search by our work and state-of-the-arts, we revealed the effectiveness of our work by more appropriately flipping the variables than state-of-the-arts according to the number of unsatisfied constraints. Although our work has the scalability in Slices# along with the variables#, mitigating the Slices# overhead is a subject of our future work.

REFERENCES

[1] H. El-Sayed *et al.*, “Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment,” *IEEE Access*, vol. 6, pp. 1706–1717, 2018.

[2] K. Devarajegowda *et al.*, “Python based framework for hdsls with an underlying formal semantics,” in *Proc. of Int’l Conf. on Computer-Aided Design*, 2017, pp. 1019–1025.

[3] A. A. Sohanghpurwala, M. W. Hassan, and P. Athanas, “Hardware accelerated sat solvers – survey,” *Journal of Parallel and Distributed Computing*, vol. 106, pp. 170–184, 2017.

[4] K. Kanazawa and T. Maruyama, “An approach for solving large sat problems on fpga,” *ACM Trans. on Reconfigurable Technology and Systems*, vol. 4, no. 1, pp. 10:1–10:21, 2010.

[5] N. Takeuchi *et al.*, “A circuit-level amoeba-inspired sat solver,” *CoRR*, vol. abs/1812.11792, 2018.

[6] N. Takeuchi, M. Aono, and N. Yoshikawa, “Superconductor amoeba-inspired problem solvers for combinatorial optimization,” *Physical Review Applied*, vol. 11, no. 4, 2019.

[7] K. Hara *et al.*, “Amoeba-inspired stochastic hardware sat solver,” in *Proc. of Int’l Symp. on Quality Electronic Design*, 2019, pp. 151–156.

[8] K. Yamamoto *et al.*, “A time-division multiplexing ising machine on fpgas,” in *Proc. of Int’l Symp. on Highly Efficient Accelerators and Reconfigurable Technologies*, 2017, pp. 3:1–3:6.

[9] M. Aramon *et al.*, “Physics-inspired optimization for quadratic unconstrained problems using a digital annealer,” *Frontiers in Physics*, vol. 7, no. 48, pp. 1–14, 2019.

[10] T. Takemoto *et al.*, “A 2 ×30k-spin multichip scalable annealing processor based on a processing-in-memory approach for solving large-scale combinatorial optimization problems,” in *Proc. of Int’l Solid- State Circuits Conference*, 2019, pp. 52–54.

[11] M. Aono *et al.*, “Amoeba-inspired spatiotemporal dynamics for solving the satisfiability problem,” *Advances in Science, Technology and Environmentology*, vol. B11, pp. 37–40, 2015.

[12] C. Ansótegui, M. L. Bonet, and J. Levy, “On the structure of industrial sat instances,” in *Proc. of Int’l Conf. on Principles and Practice of Constraint Programming*, 2009, pp. 127–141.

[13] M. Capra *et al.*, “Edge computing: A survey on the hardware requirements in the internet of things world,” *Future Internet*, vol. 11, no. 4, 2019.

[14] K. Kanazawa and T. Maruyama, “An approach for solving sat/maxsat-encoded formal verification problems on fpga,” *IEICE Trans. on Information and Systems*, vol. E100.D, no. 8, pp. 1807–1818, 2017.

[15] A. Luca, A. Ilyas, and A. Vlad, “Generating random binary sequences using tent map,” in *Proc. of Int’l Symp. on Signals, Circuits and Systems*, 2011, pp. 1–4.

[16] B. H. Fletcher, “Fpga embedded processors – revealing true system performance,” in *Proc. of Embedded Systems Conference*, 2005, pp. 1–18.

[17] M. Gort and J. H. Anderson, “Range and bitmask analysis for hardware optimization in high-level synthesis,” in *Proc. of Asia and South Pacific Design Automation Conference*, 2013, pp. 773–779.

[18] S. Bansal *et al.*, “High-level synthesis of software-customizable floating-point cores,” in *Proc. of Design, Automation Test in Europe Conference Exhibition*, 2018, pp. 37–42.

[19] Z. Newsham *et al.*, “Impact of community structure on sat solver performance,” in *Proc. of Theory and Applications of Satisfiability Testing*, 2014, pp. 252–268.

[20] G. Liu *et al.*, “Architecture and synthesis for area-efficient pipelining of irregular loop nests,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1817–1830, 2017.