A Reconfigurable, Pipelined, Conflict Directed Jumping Search SAT Solver

Mona Safar, M. Watheq El-Kharashi
Ain Shams University,
Department of Computer and Systems Engineering
Cairo, Egypt

Mohamed Shalan
American University in Cairo
Cairo, Egypt

Ashraf Salem
Mentor Graphics Egypt
Cairo, Egypt

***Abstract*—Several approaches have been proposed to accelerate the NP-complete Boolean Satisfiability problem (SAT) using reconfigurable computing. In this paper, we present a five-stage pipelined SAT solver. SAT solving is broken into five stages: variable decision, variable effect fetch, clause evaluation, conflict detection, and conflict analysis. The solver performs a novel search algorithm combining state-of-the-art SAT solvers advanced techniques: non-chronological backjumping, dynamic backtracking and learning without explicit traversal of implication graph. SAT instance information is stored into FPGA block RAMs avoiding synthesizing overhead for each instance. The proposed solver achieves up to 70x speedup over other hardware SAT solvers with 200x less resource utilization.**

Keywords—Boolean Satisfiability, Conflict-directed jumping.

I. INTRODUCTION

State-of-art SAT solvers employ different variations of Davis, Putnam, Logemann, and Loveland (DPLL) search procedure [1]. Much of the performance improvement achieved by those solvers is related to the implementation of conflict analysis, which enables the solver to perform nonchronological backjumping and conflict driven learning. A conflict is detected when one or more clauses are unsatisfied. Conflict analysis identifies the set of predecessor variables of this conflict. Conflict directed backjumping backtracks nonchronologically to the most recent assigned variable from this set. Conflict driven learning records new clause constructed from the conflict set to prevent re-exploring the same space [2]. Such advanced techniques have been ignored by the majority of hardware SAT solvers or are executed on some coupled software running on an attached host processor.

In this paper, we present a five-stage pipelined reconfigurable SAT solver that exploits the fine granularity and massive parallelism of FPGAs to evaluate the SAT formula Conjunctive Normal Form (CNF) clauses. The proposed solver executes an algorithm that combines advanced techniques: non-chronological backjumping [3], dynamic backtracking [4, 5] and learning [6]. The whole algorithm is executed in hardware overcoming any communication with a host processor.

SAT instance information is stored in FPGA on-chip memory. Thus, mapping different SAT problem instances into our reconfigurable SAT solver requires only reloading some FPGA Block RAMs, eliminating compilation, synthesis, and place-and-route overhead. This enables achieving real speedup compared to current state-of-the-art software SAT solvers.

The rest of the paper is organized as follows. Section II reviews related work. Section III presents the proposed pipelined solver. Section IV illustrates the conflict directed jumping search algorithm and how it is executed by the solver. Section V emphasizes hazard spots and how they are handled. Section VI shows the experimental results. Finally, section VII presents concluding remarks.

II. RELATED WORK

Different approaches have been explored to solve SAT on reconfigurable hardware. A good survey of hardware SAT solvers can be found in [7]. Hardware SAT solvers are mainly classified into *instance specific* and *application specific*. In an instance-specific SAT solver, a specialized hardware circuit is generated for each SAT instance. An application-specific solver provides a reconfigurable solution, where the hardware circuit is designed and compiled into hardware just once, and then customized with the specific data of each given SAT instance.

Except for Zhong et al. [8] and Gulati et al. [9, 10], hardware SAT solvers ignore conflict analysis and non-chronological backtracking techniques or leave their implementation to coupled software running on the host processor [11-13]. Zhong et al. [8] introduced a modular architecture based on a regular ring structure. The clauses of SAT CNF formula are mapped into similar complex clause modules grouped in a series on the ring. The clause module computes the implications and performs conflict diagnosis, in case of conflict, by traversing backward through the recorded implications. Learning is implemented in the software running on the host; the new constraint information is sent to the host to create new circuits for the new clause module. Speed-up of up to 91 times, compared to GRASP [3], was reported.

Gulati et al. [9] proposed a custom ASIC implementation in which clauses are implemented in banks. A column in the bank corresponds to a variable, a row to a clause, and a clause cell to a literal. The bank architecture stores the implication graph and consequently generates implications and conflicts. A hierarchical arrangement of communication units is employed for communication between the banks and the decision engine. A 4-order of magnitude speedup was reported compared to a modified version of MiniSAT, with static order decision strategy and no conflict clause simplification. Gulati et al. [10] presented an FPGA-based version of their architecture. A SAT instance is partitioned into a number of *bins* that fit into available hardware resource. The on-chip PowerPC manages bin transfers. A speedup of 17x over MiniSAT was reported. Both Zhong et al. [8] and Gulati et al. [10] reported large hardware resources utilization due to the complexity of the implemented clause module.

978-3-9810801-7-9/DATE11/©2011 EDAA

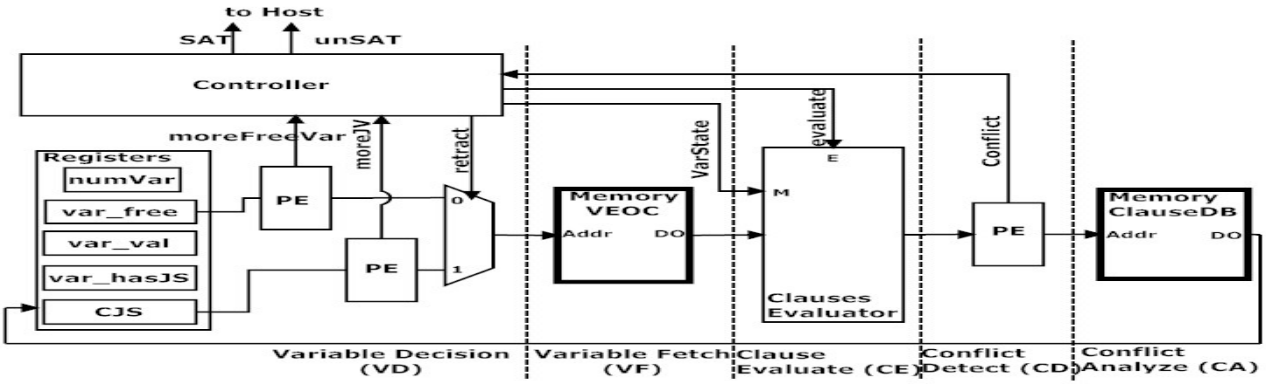

Fig. 1. Overall pipelined SAT solver architecture.

Hiramoto et al. [14] proposed an approach to implement non-chronological backtacking and clause recording in hardware without conflict analysis by partitioning a given SAT formula such that chronological backtrack behaves similar to non-chronological backtrack. However, the partial formula might contain variables which are not related to the conflict and hence the solver backtracks to redundant variables. No real hardware realization was reported.

III. Proposed Pipelined SAT Architecture

Solving SAT mainly involves picking a free variable, one after the other, assigning a logic value to it, evaluating effect of such assignment on the studied instance's satisfiability, and in case of conflict, analyzing the conflict to find a variable whose value is to be reverted to resolve this conflict. Instead of sequential execution of this procedure, we propose a pipelined SAT solver with the following stages. The overall architecture of the proposed pipelined SAT solver is presented in Fig.1.

A. Variable Decision (VD)

This stage is the main controller of the whole architecture. It performs variable decision and indicates the final result, whether the instance is satisfiable or not and the satisfying variable assignment, if any. In the normal flow, VD selects a variable that is free and gives it a value. The current implementation does not incorporate look-ahead methods such as dynamic variable selection and Boolean Constraint Propagation (BCP) [15]. VD fetches the first (lower indexed) free variable from a statically pre-ordered set of variables. We adopt the convention that we try assigning a variable logic '0' before logic '1' so that a variable will not take a logic '1' value unless implied to. If no more free variables exist, the instance is satisfiable. The **SAT** signal is raised and the search ceases.

Variables assignment status is stored in two N-bit registers, **var_free** and **var_val**. The i^{th} bit in register **var_free** denotes whether the i^{th} variable is free or not. In the latter case, the i^{th} bit in register **var_val** represents the logic value of the variable. A static order for variables decision is realized by utilizing an N-input Priority Encoder (PE) whose input is wired to the **var_free** register. This priority encoder raises the **moreFreeVar** signal and outputs the least index of the free variables, if any.

In case of conflict, VD performs conflict-directed jumping as illustrated in section IV. The JV (Jump Variable), variable whose assignment is to be undone to resolve the conflict, is determined. If conflict persists and there are no more conflicting variables to jump to, the instance is rendered unsatisfable. The index of the selected variable, new variable to assign or JV to free or reassign, is fed into the next stage.

B. Variable effect Fetch (VF)

A memory module VEOC (Value Effect On Clauses) stores the effect of variables on clauses. Each word represents how assigning a '0' to the associative variable affects the evaluation of one clause. Obviously, a variable does not affect a clause if it does not appear in it, satisfies a clause if it is assigned to '0' and its negative literal appears in it, and tends to unsatisfy a clause if it is assigned to '0' and its positive literal appears in it. In the latter case, the clause cannot be considered unsatisfied unless all other variables in it contribute with the same effect. There is no need to store the effect of assigning '1' to a variable since it is just the opposite. Signal **varState** indicates whether the variable is assigned '0' or '1'. After assigning, reassigning, or freeing a variable, its effect on all formula clauses is read from memory and fed into the next stage.

C. Clause Evaluation (CE)

Since a CNF SAT formula is a product of clauses, it is satisfiable if all of its clauses are satisfied and is unsatisfiable if any of its clauses is unsatisfied. It is subtle to evaluate each clause after each new variable assignment to ensure that it does not falsify any clause. Our clause evaluator consists of M 7-bit shift registers, one for each clause. For modularity, the maximum number of literals in each clause is limited to three. Each register is first initialized to the value "0001000". It is either right shifted, left shifted, or standstill according to whether current assigned variable's value satisfies, unsatisfies, or does not affect the clause, respectively. All clauses are evaluated in parallel. A clause is unsatisfied when the leftmost bit of its corresponding shift register is '1' [16].

D. Conflict Detection (CD)

A core step in any SAT solver is to detect whether a conflict arises for a partial or total variable assignment. Since the presence of a '1' at the leftmost bit of a shift register, in the previous stage, indicates a conflict, the leftmost bits of all the shift registers are fed to a priority encoder. In case of conflict, the priority encoder determines the index of the first unsatisfied clause and raises the **conflict** signal [17].

E. *Conflict Analysis (CA)*

A memory module ClauseDB (Clause DataBase) acts as a clause database, each word represents a clause with its associated variables' indexes. Using the clause index forwarded from conflict detection stage, the indexes of variables in the unsatisfied clause are fetched. Variables of this clause are designated as the conflicting variables since at least the assigned value for one of them must be flipped to resolve the conflict. A conflict Jump Set (CJS) is initialized with the conflicting variables. This represents the first stage in our conflict analysis technique. The rest is performed in the VD stage as will be illustrated in-depth in the following section.

IV. CONFLICT DIRECTED JUMPING SEARCH ALGORITHM

VD and CA stages are responsible for performing conflict analysis, non-chronological jumping and learning. In the proposed solver, the VD will always assign the selected free variable to logic '0', thus a variable will not take a logic '1' value unless implied to. Each variable has an associated Jump Set (JS), which when the variable is assigned to '1', stores the assigned variables which forces the variable to '1'. Thus, the JS records the conflict set, nogood or learnt constraint, which implies the JV to be assigned a logic '1' value. The term JS is used to emphasize the fact that if a '1' valued variable is to be reassigned to '0', then the solver must jump to one of variables in jump set to revert its value for this new assignment to be satisfiable.

For instance, consider we have clause (x1 + -x5 + x8) in a given instance, x1 is assigned to '0' and x5 is assigned to '1'. On assigning '0' to x8, the clause is unsatisfied. Hence, x8 is forced to '1' and its JS stores the set {x1 = 0, x5 = 1}. If x8 value is to be changed, at any time throughout the search process, then either x1 or x5 values must be changed to remove this constraint.

Upon conflict detection, the retraction process is initiated in the VD stage. The recursive function **Conflict_Directed_Jumping( )**, illustrated in Fig. 2., fetches the Jump Variable (JV) as the highest indexed variable in CJS, initialized in CA stage to variables in the unsatisfied clause. If JV has not been fully explored, i.e. both logic values have not been tried, it is reassigned. For the current implementation, if JV is assigned '0', then its value is changed to '1'. The variable's associated JS is reset to store all other conflicting variables to which the solver could have jumped to resolve the detected conflict.

On jumping, the variables previously assigned between JV and the variable that causes the conflict are not freed, they keep their current assignment as in dynamic backtracking [4, 5] but without search space reordering. This behaves as if the branch under the left branch of JV, logic '0' branch, is copied as is under the right branch corresponding to logic '1'. Much of the search space re-explored by conventional backjumping is pruned.

If JV is already assigned logic '1', the persistence of its associated JS is checked to find out if the cause that forced the variable to '1' still holds. A JS set does not persist if one or more of the variables in the set has been reassigned or freed since the set has been associated to the variable. This indicates that the constraint that previously implied the JV to be assigned logic '1' has been relaxed. In such case, the JV is reassigned to zero. If JS persists, it is merged into the current CJS and the JV is freed. The function **Conflict_Directed_Jumping( )** is recalled to fetch a new JV from the updated CJS and the process is repeated. If no more jump variables exist, the instance is rendered unsatisfiable. The **unSAT** signal is raised and the search ceases.

```
Conflict_Directed_Jumping(CJS)
begin
  JV ← get Jump Variable; highest indexed variable in CJS
  if (JV is NULL)
     unSAT ← True;                          // Instance is unsatisfiable
  else
     if (both logic values has been tried and associated JS persist)
        free JV;
        CJS ← CJS U JS[JV] – JV;            // Merge conflict set
        Conflict_Directed_Jumping(CJS);
     else
        reassign JV;
        JS[JV] ← CJS – JV;                  // Set/update JV associated JS
end procedure
```

Fig. 2. Proposed conflict directed jumping algorithm.

On freeing the variable, its associated learnt JS is not freed or deleted so that in the decision stage when a free variable is assigned, the JS is utilized to further prune the search space. If the free variable has a previously associated JS and this JS persists, then the variable is assigned a logic '1' value. Thus, learning is achieved without explicit new clause recording avoiding the consumption of new hardware resources.

The designation CDJ is used to emphasize the difference from Conflict Directed Backjumping (CDB) [3, 15] and dynamic backtracking [4, 5]. The conflict jump set is not created only at dead end nodes where all the values of the node's associated variable have been tried. But, it is also created when conflict arises on assigning logic '0' to a variable and instead of reassigning it to '1', the solver jumps forward (forjump) to a node at deeper depth in the explored search tree. In dynamic backtracking, the search space is reordered dynamically by moving the backjump variable deeper in the search space. Conflict, if any, will arise at the search tree leaf node and might be solved by only backjumping to a shallower node.

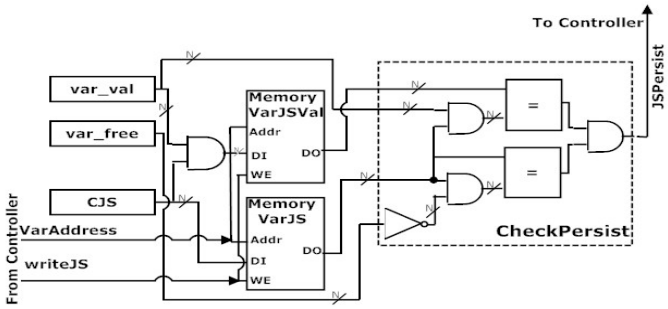

Fig. 3. Circuit to check persistence of addressed variable's jump set.

For physical realization, the JS associated with each variable is stored in two memory modules: **VarJS** (Variable

Jump Set) storing the indices of the variables in the JS of each variable, and **VarJSVal** (Variable Jump Set's Value) storing the assigned values of these variables at the time the JS was updated. An N-bit register **CJS** is used to store the conflict jump set. A high bit indicates that the corresponding variable is a candidate jump variable. The **CJS** is fed into an N-input priority encoder which outputs the highest index of the conflicting variables in CJS; the jump variable **JV**. The **moreJV** signal is de-asserted in case there is no more conflicting variable to jump to revealing the unsatisfiability of the SAT problem. The circuit for checking persistence of a JS is shown in Fig. 3.

Example: Consider the following SAT formula:

$$Ø = (x1 + -x4) . (x1 + - x9) . (-x2 + x4 + x8) .$$

$$(x4 + x7 + x9) . (x1 + -x7 + x10). (x2 + -x7 + -x10).$$

The explored search space is illustrated in Fig.4. At step (a), all the variables 1 to 8 have been assigned to logic '0'. A conflict arises on assigning logic '0' to variable 9. The clause (x4 + x7 + x9) is unsatisfied. The variable is reassigned logic '1' and its associated JS is set to {-4, -7}. Now clause (x1 + -x9) is unsatisfied and x9 is freed. The CJS is {-1, -4, -7}. Thus, the JV is variable 7. At step (b), x7 is reassigned logic '1' and its associated JS is set to {-1, -4}. Variable 9 can now be safely assigned logic '0'. Due to clauses (x1 + -x7 + x10) and (x2 + -x7 + -x10), variable 10 cannot be assigned either logic. Its associated JS is set to {-1, 7}. The CJS is {-1, -2, 7}. Again, x7 is the JV. It is already assigned logic '1', hence, it is freed and its JS is merged into the CJS to have the set {-1, -2, -4}.

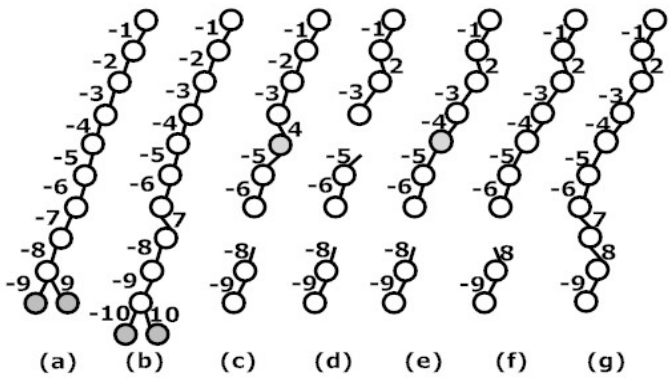

Fig.4. Example explored search space.

At step (c), x4 is reassigned logic '1' and its JS is set to {-1, -2}. A conflict arises at clause (x1 + -x4). Thus, x4 is freed and CJS is {-1, -2}. At step (d), the solver jumps to x2 and reassign it to '1'. Its JS is associated to {-1}.

At step (e), the next free variable is x4. On assigning '0' to x4, clause (-x2 + x4 + x8) is falsified. The CJS is {2, -4, -8} and hence x8 is the JV, which when reassigned to '1' resolves this conflict at step (f). At step (g), x7 is assigned logic '1' since its recorded JS {-1, -4} persists. This avoids re-exploring the conflict at step (a).

It is true that some recorded jump sets do not represent a new constraint. For instance, in the previous example, the jump set {-4, -7} associated with x9 represents the clause (x4 + x7 + x9), which is already one of the formula clauses. However, some recorded jump sets represent new added clauses. For instance, the JS {-1, -2} associated to x4 represents the learnt clause (x1 + x2 + x4). Also, JS {-1} associated to x2 represents the new recorded clause (x1 + x2).

V. PIPELINED SAT HAZARDS RESOLUTION

The pipeline normal execution flow is shown in Fig. 5, where FV, NFV, and N2FV stand for Free variable, Next Free Variable, and Second Next Free Variable, respectively and so on. To assign x variables and assure their consistency for the problem satisfiability, only x + 4 clock cycles are needed.

Variable number	Clock cycle								
	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8	i+9
FV	VD	VF	CE	CD	CA				
NFV		VD	VF	CE	CD	CA			
N2FV			VD	VF	CE	CD	CA		
N3FV				VD	VF	CE	CD	CA	
N4FV					VD	VF	CE	CD	CA

Fig. 5. Pipelined SAT solver flow in case no conflict arises.

In the following, we examine all possible scenarios of deviation from the normal flow, emphasizing hazard spots and illustrating how they are handled.

Case 1: Conflict arises

Consider assigning '0' to FV, at cycle i+1 in Fig. 5, causes conflict. Conflict is detected four cycles later. During these cycles variables (NFV to N4FV) are assigned. After conflict detection, it takes the solver one cycle to determine JV. In order not to stall the pipeline, at clock cycle i+6, one more free variable (N5FV) is assigned. If no more FVs exist, a dummy variable is injected into the pipeline. At clock cycle i+7, the solver jumps to JV.

A data hazard may occur due to the conflict signal propagating throughout pipeline stages in cycles i+5 till i+9. The conflict signal is still falsely raised since effect of reassigning JV does not take place yet. The solver may incorrectly further jump or detect the unsatisfiability of the problem. This hazard is simply avoided by masking out conflict signal during cycles i+5 to i+9.

Case 2: Out of order conflict handling

This case covers an extension scenario of case 1. After the solver jumps and reassigns either the JV, if any of the variables (NFV, N2FV, N3FV, N4FV, or N5FV) assignments or the reassignment of JV causes conflict, this will be caught at cycle i+11 and conflict-directed jumping will be reconducted. If more than one of those assignments causes conflict, a situation may arise where out of order handling of conflicts occurs. This means that although Vi is assigned before Vj, the conflict resulting from the assignment of Vj is detected and resolved, if possible, before that of Vi. The order in which conflicts are handled does not affect the correctness of the solution since the solver does not ignore any conflict and handles each conflict

independently. However, considering performance in terms of pruning the search space, we differentiate between three situations: similar performance, positive improvement by pruning redundant search space, and performance degradation by exploring useless search space. For instance, consider Vi and Vj lead to conflict (i < j). Clause Cm is the first (lower indexed) unsatisfied clause due to the assignment of Vi and Jm is the corresponding JV for resolving the conflict. Clause Cn is the first unsatisfied clause due to the assignment of Vj and Jn is the corresponding JV. Since our solver always handles at first the lower indexed clause in the clause database, one of the followings can occur:

- If m ≤ n or Jm = Jn, no disorder takes place. Pipeline explored search space is the same as that of the non pipelined flow.
- If n < m and Jn < Jm, i.e., at shorter depth of search tree, positive improvement is achieved by pruning redundant search space.
- If n < m and Jn > Jm, i.e., at longer depth of search tree, useless search branches maybe explored, degrading the performance.

Case 3: No more Free Variables

This case deals with the scenario when a leaf node of the decision tree is reached and no more FVs exist. Since a conflict is detected four cycles after assigning the conflicting variable, the solver needs to wait those cycles to assure that none of the latest assigned variables causes a conflict. Instead of stalling the pipeline, a dummy variable is injected into the pipeline

VI. EXPERIMENTAL RESULTS

The proposed architecture has been implemented on Xilinx Virtex-II Pro (XC2VP30-FF896) development board. The implemented hardware circuit supports up to 511 variables and 511 clauses, runs at 120 MHz, occupies 22677 LUTs, 82% of available LUTs, and 65 - 512x32 FPGA block RAMs (BRAMs), 47% of the available. We developed a configuration generator in C++ for mapping different SAT instances to our reconfigurable hardware SAT solver by generating BRAM initialization data and executing Xilinx Data2MEM utility to update the once and for all generated configuration bitstream file. The SAT solver configuration generator is executed on Intel Pentium M/1.86 GHz/1 GB running Windows XP. The whole configuration process including downloading the bitstream to FPGA takes an average of 0.24 second.

Table I shows a comparison of our SAT solver with Zhong et al. [8] in terms of the number of clock cycles needed to solve the problem. The experimented instances are limited by Zhong et al. published results [8]. Our proposed pipelined CDJ-based SAT solver achieves an average of 134x and up to 1116x speedup.

The clause cell, basic building module, of Zhong et al. architecture occupies a 4 X 16 CLBs of utilized Xilinx XC4036EX FPGA in which a CLB contains 2 4-input LUT and 1 3-input LUT [8]. Considering only clauses hardware utilization and neglecting hardware resources occupied by controller and global signal, an implemented circuit handling 511 clauses will occupy 98112 LUTs, 4.3x greater than LUTs utilization of our whole solver.

TABLEI. COMPARING PROPOSED SOLVER RUNTIME IN HARDWARE CLOCK CYCLES WITH ZHONG ET AL.

Instance	Zhong et al. [8]	Proposed solver	Speedup
par8-1-c	700	2435	0.287
aim-50-2_0-yes1-2	7151	6241	1.146
aim-100-1_6-yes1-1	2985176	17043	175.2
aim-100-3_4-yes1-4	775641	11285085	0.069
hole7	1789295	1831751	0.977
hole8	25934711	17161273	1.511
hole9	409537410	176990981	2.314
aim-50-1_6-no-1	93396	2457	38.01
aim-50-2_0-no-1	9763745	8747	1116.23
aim-50-2_0-no-4	294182	156111	1.884
pret60_40	181611844	1331687	136.4

Gulati et al. [10] implemented their solver on Xilinx XC2VP30, the same board we use for implementation. The board can only accommodate 16 variables and 24 clauses, whereas for our architecture, it accommodates 511 variables and 511 clauses. Our architecture is 237 times cheaper in terms of required resources. Gulati et al. derived a performance model and projected the runtime for a Xilinx XC4VFX140. Columns 2 and 3 of Table II show the reported projected runtimes in [10] for hardware runtime and PowerPC software runtime to handle SAT instance's partitions. Columns 4 and 5 show runtimes after removing projection for fair comparison with our solver over the same board. Considering only raw hardware runtimes, a speedup of 5.77x is achievable. This speedup rises to 70x considering the time spent by PowerPC.

To illustrate the feasibility of our approach, we compared the SAT solver performance with SATzilla[18], a portfolio-based algorithm selection for SAT, and a five medals winner in the 2009 SAT competition. SATzilla employees the latest sophisticated software techniques used in the state-of-art software SAT solvers like BCP, learning, backjumping and restart. We run SATzilla_C (for crafted instances) on Intel Xeon/3.2 GHz/ 2 GB running Linux with the same competition settings. Since, performance of a SAT solver varies from instance to instance; we report accumulative runtime over instances from the same test suite [19]. Table III illustrates that, an up to 9700 raw speedup ratio is achieved, considering only raw hardware execution runtime. Considering reconfiguration time, a total speedup of up to 8x is achieved.

For the small SAT instances currently supported by the implemented hardware, the average reconfiguration time of 0.24 second dominates the overall execution time. This is apparent in the uf20-91 test set with 1000 instances. Though a 9734.1x raw speedup is achieved, the total speedup is shadowed by time to reconfigure 1000 instances (1000 * 0.24). The reconfiguration time could be effectively reduced on using PCI-X, rather than Xilinx platform USB cable, or loading several CNF instances onto the board DRAM, form which one instance at a time is loaded into on-chip BRAM. For larger instances, the reconfiguration time is negligible and the benefit

TABLEII. COMPARING PROPOSED SOLVER RUNTIME IN SECONDS WITH GULATI ET AL.

Instance	Gulati et al. [10] targeting XC4VFX140 [sec]		Gulati et al. targeting XC2VP30 [sec]		Our proposed solver [sec]	Speedup	
	HW	*PowerPC*	*HW*	*PowerPC*		*HW*	*Overall*
aim-50-2_0-yes1-2	0.00000149	0.000000299	0.00000699	0.00007008	0.000052008	0.13	1.482
hole6	0.00223	0.0005	0.01045313	0.1171875	0.00181082	5.77	70.49
hole8	0.121	0.0304	0.5671875	7.125	0.14301060	3.97	53.79
uuf100-0457	0.0258	0.00439	0.1209375	1.02890625	0.50912100	0.24	2.258

from hardware execution parallelism is more apparent allowing a more significant speedup to be realized.

TABLEIII. COMPARING PROPOSED CDJ-BASED SAT SOLVER EXECUTION TIME WITH SATZILLA2009_C.

Test set	SATzilla 2009_C [sec]	Proposed solver [sec]	Speedup	
			HW	*Total*
Pigeon hole (4 not satisfiable instances)	22.85	1.64	13.98	8.81
aim-50 (24 satisfiable/ not satisfiable)	27.91	0.0285	979.1	4.82
aim-100 (20 satisfiable/ not satisfiable)	25.41	0.93492	27.18	4.43
aim-200 (10 satisfiable/ not satisfiable)	14.98	3.24691	4.617	2.65
uf20-91 (1000 satisfiable)	73.88	0.00759	9734.1	0.31

VII. CONCLUSIONS

We presented a Conflict Directed Jumping (CDJ)-based search algorithm for solving the satisfiability problem best suited for hardware. The proposed algorithm can efficiently explore the search space pruning much of the redundant areas. Unlike conflict-driven backjumping techniques used in hardware solvers, our proposed algorithm does not rely on backward traversal of an implication graph saving a lot of hardware resource utilization. Variables assignment done between the decision level at which jump variable was assigned and the decision level at which conflict is detected are not retracted. We also presented a reconfigurable, pipelined SAT solver that utilizes the hardware massive parallelism to implement the proposed CDJ-based search algorithm. SAT instance information is stored in FPGA on-chip memory. Thus, mapping different SAT problem instances into our reconfigurable SAT solver requires only reloading some FPGA Block RAMs. The solver achieves up to 70x speedup over other hardware SAT solvers with 200x less resource utilization.

REFERENCES

[1] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-proving," Communications of the ACM, vol. 5, no. 7, pp. 394–397, July 1962.

[2] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," in Proceedings of the 14th International Conference on Computer-Aided Verification (CAV 2002), pp. 17–36, July 2002.

[3] J. P. Marques-Silva and K. A. Sakallah, "A Search Algorithm for Propositional Satisfiability," IEEE Transactions on Computers, vol. 48, no. 5, pp. 506-521, May 1999 .

[4] M. L. Ginsberg, "Dynamic Backtracking," in Journal of Artificial Intelligence Research, vol. 1 , no. 1, pp. 25-46, August 1993.

[5] M. L. Ginsberg and D. A. McAllester, "GSAT and Dynamic Backtracking," Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming, pp. 243-265, August 1993.

[6] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2001), pp. 279–285, November 2001.

[7] I. Skliarova and A. B. Ferrari, "Reconfigurable Hardware SAT Solvers: A Survey of Systems," IEEE Transactions on Computers, vol. 53, no. 11, pp. 1449-1461, November 2004.

[8] P. Zhong, M. Martonosi, and P. Ashar, "FPGA-based SAT Solver Architecture with Near-zero Synthesis and Layout Overhead," IEE Proceedings on Computers and Digital Techniques, vol. 147, no. 3, pp. 135-141, May 2000.

[9] K. Gulati, M. Waghmode, S. P. Khatri, and W. Shi, "Efficient, Scalable Hardware Engine for Boolean Satisfiability and Unsatisfiable Core Extraction," IET Computers and Digital Techniques vol 2, issue 3, pp. 214–229, May 2008.

[10] K. Gulati, S. Paul, S. P. Khatri, S. Patil, and A. Jas, "FPGA-based hardware acceleration for Boolean satisfiability," ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 14, no. 2, March 2009.

[11] J. de Sousa, J.P. Marques-Silva, and M. Abramovici, "A Configware/Software Approach to SAT Solving," Proceedings of the 9th IEEE international Symposium on Field-Programmable Custom Computing Machines, pp. 239-248, May 2001.

[12] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, "Designing an Efficient Hardware Implication Accelerator for SAT Solving," Proceedings of the 11th international conference, SAT 2008, May 2008.

[13] J. D. Davis, Z. Tan, F. Yu, and L. Zhang,. "A Practical Reconfigurable Hardware Accelerator for Boolean Satisfiability Solvers," Annual ACM IEEE Design Automation Conference, Proceedings of the 45th annual Design Automation Conference (DAC), pp. 780-785, June 2008.

[14] S. Hiramoto, M. Nakanishi, S. Yamashita, and Y. Nakashima, "A Hardware SAT Solver Using Non-chronological Backtracking and Clause Recording Without Overheads," Proceedings of 3rd international workshop, ARC 2007, pp. 343-349, March 2007.

[15] R. Dechter and D. Frost, "Backjump-based Backtracking for Constraint Satisfication Problems," Artificial Intelligence, vol. 136, no. 2, pp. 147-188, April 2002.

[16] M. Safar, M. Shalan, M. W. El-Kharashi, A. Salem: A shift Register based Clause Evaluator for Reconfigurable SAT Solver. IEEE/ACM Design, Automation, and Test in Europe (DATE 2007)

[17] M. Safar, M. W. El-Kharashi, A. Salem, FPGA based accelerator for 3-SAT clauses conflict analysis in SAT solvers: Correct Hardware Design and Verification Methods (Charme 2005) 384–387

[18] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based Algorithm Selection for SAT," Journal of Artificial Intelligence Research, vol. 32, pp. 565–606, June 2008.

[19] SATLIB Benchmark Suite, http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html