Boolean Satisfiability Solvers: Techniques, Implementations and Analysis

João F. Lima

Abstract— This paper presents an overview of the most common techniques employed for solving the SAT problem. Such techniques have allowed the SAT solvers to be reliable enough in order to solve both random and practical instances. Another point focused in this paper is the applicability of Reconfigurable Systems in order to accelerate the SAT solving process. An emphasis to HW/SW based solutions will be done and an analysis of those systems will be done. This solution is widely used today, allowing a system to take advantage from both the high speed personal computer and, at the same time, parallelism and flexibility provided by reconfigurable systems.

Index Terms – Boolean Satisfiability, Hardware/Software partitioning, Reconfigurable Systems, Application Specific Processors.

I. INTRODUCTION

The Boolean Satisfiability (SAT) is a very well known NP-complete problem, which has been studied in a deeper way since 60's. Given a propositional or Boolean formula, the problem consists of determining if there is any variables' assignment that produces a given formula to be satisfied. Therefore, some operations and decisions must be performed attempting either to find a solution for a given instance or to detect that there is no solution. The entity that commands all the operations needed is referred to as ***SAT Solver***.

In the last decades, these solvers have begun to be intensively used in industrial applications, which has construct a faster solver as well as to increase the variable number that can be processed. SAT solvers are used in the electronic design automation (EDA) industry for a variety of tasks, including microprocessor verification [1], AI planning [2], automatic test pattern generation for circuits [3], cryptanalysis [4], verification and testing of digital systems [5] among others.

The SAT solvers can be separated in two categories: **Complete or Systematic and Incomplete Solvers**. The first are characterized by the use of an algorithm which always produces a result, i.e. either gives a possible assignment of variables for a formula or proves that formula is unsatisfiable (e.g.: Grasp [6] zChaff [7], SATO[8], BerkMin[9], MiniSat [10], etc.). Alternatively, Incomplete Solvers cannot ensure that an assignment of variables is reached, even if it exists, or prove that a formula cannot be satisfied, and they usually have a pre-set time to solve the instance (e.g. GSAT[11], WalkSAT[12]). The latter are based on Local Search instead of the Backtracking or Branching techniques used by complete solvers.

About the *Solver's* implementations, along the latest years different approaches were seen for developing such systems. From SW, to SW/HW and only HW solutions, different methods have been proposed.

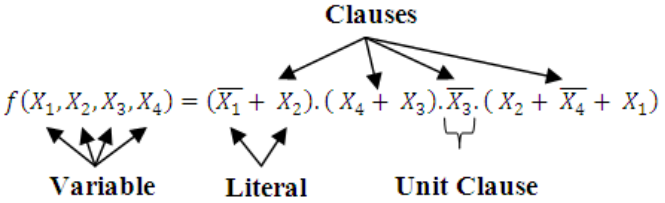
In this paper, some of the most commonly used techniques are presented. Since most of the future work will be done using a SW/HW approach, some of the implementations using that approach are presented, namely using reconfigurable hardware (e.g. FPGAs, CPLDs).

The remainder of the paper is organized in four sections. Section II provides some of the basic concepts such as problem's representation and some definitions essential to understand the SAT problem. Since most of the implementations employ a complete SAT solver, in Section III the basic template for all the complete solvers is presented, as well as a deep analysis of the most successful techniques concerned to this basic template. Section IV discusses some SW/HW implementations that take advantages from this partitioning, i.e., implementations that employ solving tasks in both SW and HW and not only some preprocessing before the solving procedure. Finally, concluding remarks are given in section V.

II. PROBLEM'S REPRESENTATION

A Boolean formula consists in a logic expression which depending on the variables values (TRUE or FALSE) can produce either a true (Satisfiable) or a false (unsatisfiable) result. The most common representations are: **CNF** (conjunctive normal form) or **DNF** (Disjunctive normal form). In the first case the formula is represented using products of sums instead of sums of products (which is the DNF representation).

For a better comprehension in the next sections, some of the basic definitions will be now present. Let us consider the following Boolean formula in CNF representation with four variables (X_1, X_2, X_3, X_4).



As noticed in the example above, a Boolean formula is composed of clauses and each element within it is referred to as a **literal**. A clause that only has one element is designated a **unit clause**. In some applications the formula may have a fixed number of literals, *k*, in each clause. In this case the problem is called *k*-Sat with $k \in N$ and $k \geq 2$.

Since all techniques and implementations are based in the CNF representation, it will be used in the next sections.

III. DPLL

Researchers have begun to take interest in the SAT problem long decades ago. Since that time, a large number of techniques and implementations were proposed trying to overcome some technical problems such as: memory explosion, resolution speed, number of variables, number of clauses, etc. However, even though great advances were reached, the basic template for most of the SAT solvers is still almost the same. The first template was proposed in 1960 and is referred as DP algorithm [13]. This algorithm is affected by memory explosion problem. In order to solve this problem, an improvement was proposed in 1962 by M. Davis, G. Logemann, and D. Loveland [14] which was accepted as a basic template for the complete SAT solvers. This is referred to as DPLL algorithm. A pseudo-code of this algorithm is the following:

```
status = preprocess();
if (status!=UNKNOWN) return status;
while(1) {
decide_next_branch();
while (true) {
    status = deduce();
    if (status == CONFLICT) {
        blevel = analyze_conflict();
        if (blevel == 0) return UNSATISFIABLE;
        else backtrack(blevel);
    }
    else if (status == SATISFIABLE)
        return SATISFIABLE;
    else break;
}
```

DPLL algorithm is based on the Branch and Search philosophy, i.e., the algorithm follows a determinate branch trying to find out one solution. In case that it is impossible to find it by the chosen branch, the algorithm does some kind of backtracking process and restarts by following another branch, repeating this until some solution is reached or, in the worst case, finds that there is no possible solution for that particular instance.

Let us now present an overview of the DPLL algorithm. Further details about each function will be given in later sections.

The algorithm initiates with all the variables unassigned. At this point the function ***preprocess*** is called. Within it, some processing will be done on the instance in order to get a more efficient representation due to some variations on this function's algorithm. In a later section, an example of preprocessing will be given. When all the preprocessing is done the construction of the solution begins. This process will start with the function ***decide_next_branch***. The latter determines, through some specific algorithm, which will be the next variable to be assigned. This decision is then propagated, i.e., when a variable is chosen, that decision will simplify the problem, and as a consequence some variables should be assigned with a specific value (**implication**) in order to satisfy some clause. We refer this process as ***deduce***. For example: Imagine that the variable is assigned to value '0' then the 2nd clause only have one more literal to be assigned (X_4). Thus the variable X_4 is implied because it must be assign to value '1' in order to satisfy the 2nd clause.

When all the implications are solved, the algorithm needs to test if those attributions inferred some conflict, i.e., if there is any variable that must be assigned to both value TRUE in one clause and value FALSE in another clause. If it happens, backtracking will be done by the function ***analyze_conflict***. Another task employed by this function (in the more recent solvers) is to take some information about the conflict and learning with it (**conflict-driven learning)** in order to prune search space in the future. Then, the output variable ***blevel*** is tested. If it reaches zero value it means that there is no more possible backtracking and so the instance is impossible to solve. In the other case, if there is no conflict, the solver will test if the satisfiable condition was reached. If it is TRUE all the process will be finished. If not, all the cycle explained above will be repeated until either the satisfiable condition is reached or it has been proved that the instance is impossible to be satisfied.

Now that we have a general idea of the presented functions, let us analyze each function more deeply.

A. Decide Next Branch (Branch Heuristics) or Decision Heuristics

Decide Next Branch (or *Decision Heuristics*) will choose the next variable to be assigned. The concept is easy to understand, however a very important issue will appear immediately: '*which variable should be assigned next?'*. The importance of choosing the right variable is a very well known problem and different procedures will lead to different search trees as well as different memory consumptions, solving time, etc.

In the earlier years, the proposed algorithms tried to satisfy the largest number of clauses at once or infuse the largest number of implications. In [15] and [16] such techniques have been applied and tested. These are based on statistics, which is useful when we are dealing with random SAT instances, but in structured problems they do not get relevant information, making this approach inefficient for those instances [17]. An alternative to this method was first presented in [18] when the author proposed the use of literal count heuristics i.e., the

algorithm counts in each phase of the solving process the number of the unsatisfied clauses in which a certain variable appears. It was proved that if we choose the variable to be assigned as the variable with the dynamic largest combined sum (**DLIS**), we can get satisfactory results. It must be noticed that with this technique the counters are state-dependent, which means that each time the function *decide_next_branch* is invoked, all the counters must be updated in order to determine the variable with DLIS. Another approach similar to DLIS is the **VSIDS** which was introduced in [7], nowadays being a reference. Thus, it will now be explained in more detail. Schemes which derivate from VSIDS - like BerkMin [9], MiniSat[10], Jerusat[19] and RSat [20] - will not be discussed in this paper.

VSIDS - VARIABLE STATE INDEPENDENT DECAYING SUM

In this scheme a variable is chosen by its weight (one counter per variable) being periodically decayed or boosted when it appears in a conflict clause (redundant clause added to the original formula due to a learning process)

Initially all the counters, *s(v),v=1..Number of variables*, have their own number of occurrences in the Boolean Formula. When a conflict clause with variable v_1 is added, $s(v_1)$ will be incremented. In every N decisions, all the counters are updated applying the following expression:

$$s(v) = r(v) + s(v)/2,$$

where *s(v)* is the counter of the variable *v* and *r(v)* is the number of occurrences of the variable *v* in the conflict clauses since the last counters update. When a variable must be assigned, the solver takes the variable with the highest score. This process takes about 10% of the solver's run-time.

B. Deduce

The function Deduce has to determine the consequences of the variable attribution which was decided by the function decide_next_branch. There are four different situations that can occur as a consequence of the variable assignment:

- All the clauses are satisfied and thus a solution has been reached.
- An implication has occurred, and thus, the **unit clause rule** should be applied, i.e., when one variable is implicated it should be assigned with the value that will satisfy the clause where it is inserted. This process, often referred to as ***Boolean Constraint Propagation (BCP)***, will be repeated until there are no more implications to solve.
- A conflict clause appears and so the function *analyze_conflict* must be called.
- There are unsatisfied clauses but there are no more variables (using unit clause rule) to be assigned and thus function *decide* must be called.

Even though the behavior of the function *deduce* is quite simple, the Boolean Constraint Propagation used for all the solvers consumes about 90% of the run-time which means that the BCP is the most important component in the SAT solvers and it is where a very efficient algorithm is needed in order to obtain the least propagation time. Some techniques for the BCP engine will now be presented.

BCP (Boolean Constraint Propagation)

The BCP engine is the most important component in the modern SAT solvers since it consumes most of the run-time, which led to a high attention from researchers. The BCP must be capable of detecting conflicts and implications after the variable assignment.

In the earlier years, this engine was projected using counters in each clause. This method is simple and it was used in very well known SAT solvers like GRASP[6] or SATZ [21].

Let us see the GRASP example. The method adopted consists of keeping two counters in each clause: one is counting the literals with value '1' and the other is counting the literals with value '0'. At the same time all the variables have two arrays, one indicating in which clauses they appear and the other containing their values (Negative or Positive). Now imagine that a variable assignment is taken, then all the clause counters in which the variable appears will be updated. The algorithm to detect if there is a conflict or an implication is reached is as follows:

- If a clause counter with respect to the value '0' is equal to the number of literals within that clause, then a **conflict** has occurred.
- If a clause counter with respect to the value '0' is equal to the *number of literals – 1* and the clause counter for the value '1' is zero then an **implication** has been reached.

Even though this approach is simple, it is inefficient for formulas that contain a large number of clauses and literals, since all the counters must be periodically updated and enough memory resources must be allocated for all the counters.

Later, in SATO[8] and in zChaff [7], methods without the use of counters were proposed. A more detailed description of these two methods is presented in the next subsections.

1. Head/Tail List in SATO

The method used in SATO is based on pointers. For each clause two pointers are kept: **Tail and Head**. Initially the Tail pointer is pointing to the first literal in the clause and the Head Pointer is pointing to the last literal. For each variable the solver maintains four linked lists: *clause_of_pos_head(v)*, *clause_of_neg_head(v)*, *clause_of_pos_tail(v)* and *clause_of_neg_tail(v)*. These lists keep the pointers to the clauses in which the variable *v*

is on the tail or on the head and its phase (Negative or Positive). Now imagine that the variable *v* is assigned to value '0' - then the lists *clause_of_neg_head(v)* and *clause_of_neg_tail(v)* will be ignored since the clauses to which they point will now be satisfied. On the other hand, for each clause pointed by the *clause_of_pos_head(v)* and *clause_of_pos_tail(v)*, the tail or the head will be moved to the next literal that is not assigned yet. In respect to the search process the following situations may occur:

- If a literal within a clause takes the value '1' the clause will be satisfied and then both clause pointers will not change;
- If a literal *l* is free and the tail pointer is not pointing to a free literal then the tail must be pointed to the literal *l*. This operation is called as **moving a literal to the tail**;
- If all the literals between the tail and the head are assigned to the value '0' and the tail is pointing to a free literal then an implication has reached;
- If the literal pointed by the tail has the value '0', as well as all the literals between the tail and the head, then a conflict has occurred.

This method is more efficient than a method based on counters since when a literal gets the value '1' it is not needed to visit the clauses where it occurs. The lesser computation effort of using pointers instead of counters is also a point in favor. Despite of these advantages, this method still has one great problem since when a backtracking is performed all the pointers must be backtracked as well. This leads to a huge effort by the computation system since it has to change all the pointers in all the clauses where the variables to be backtracked do occur.

2. Watched variables

In zChaff [7] the authors proposed a new method referred as to *2-watched variables*. This is similar to the previous implementation in such way that both methods use two literals to find either if a conflict has occurred or a variable is implied. This scheme uses two lists for each variable: *pos_watched(v)* and *neg_watched(v)* which contain the clauses were the variable *v* is a watched variable and its phase. Another feature is that the pointers within the clause do not have to be ordered and so initially there is no preference for the literals that must be watched.

Assume now that the logic value '1' was assigned to the variable *v*. In this case, the solver will search in the *neg_watched(v)* list for a literal *l* (within the same clause as the literal with the variable *v*) which is not set to the logic value '0', i.e., either not assigned or assigned to value '1'. As a consequence of this search process the following situations may take place:

- If the literal *l* exists and is not the other watched literal, then the pointer to the last assigned literal will be pointed to the literal *l*;
- If the literal *l* is the one who is pointed to by the second pointer and whether it is free (not assigned) then an implication was found, and as a consequence the unit clause rule must be applied;
- If the literal *l* is the other watched literal and its value is assigned to '1' then there is no operation to be performed since the clause is satisfied;
- If the literal *l* does not exist then a conflict has occurred;

The *2 watched-variables* method has the same advantages provided by Head/Tail in SATO comparing with the clause counters method. However, this method provides a great advantage in terms of the consumed time when a backtracking process is needed in which the undo process takes a constant time. This characteristic is achieved since when a backtracking is performed the two watched variables are still the same as when the conflict has occurred. This is true because the last watched variables always have the zero value assigned and thus when the process of unassignment takes place, these will be unassigned or assigned to the value '1', and, as a consequence, the pointers should not have to change saving time resources.

C. Conflict Analysis and Learning

As mentioned in the DPLL algorithm's overview, this engine (function) is responsible for the conflict analysis, finding a solution, promoting the next step to be followed and learning with conflicts in order to prune search space in the future.

In this section the basic definitions will be presented instead of a deeper analysis. This is because the future work will not be focused around this area.

In the original DPLL algorithm, a simple strategy was performed. Here, all variables have a flag which indicates if the variable was already tried in the both phases (Negative and Positive). Thus, when a clause conflict occurs, the solver tests if the last variable was already tried in both phases. If not, the remaining phase is assigned and the result of that operation will then be propagated. In case that the last assigned variable was already tested in both phases a **chronological backtracking** is performed, i.e., the solver always undoes the last decision and tries another variable. Even though this is a simple scheme it only works well for random instances [17]. It was successfully employed in very well known SAT solvers such as SATZ [21].

For real world applications, chronological backtracking is not the most efficient method. Instead of just undoing the last assignment, a more complex technique was developed in order to detect the reason of such conflict and thus backtracking to an earlier level of decision, saving computational system run-time. This technique is referred

to as **non-chronological backtracking**. The basic concept of this approach is to get information about the conflict and thus learn (**conflict-directed learning**), adding redundant clauses to the original formula. These redundant clauses do not change the Satisfiability of the original instance but they can prune search space in the future. GRASP[6], zChaff[7] and SATO[8] are examples where this approach was employed and in which good results have been achieved. Such learned clauses are often referred to as conflict clauses and the clauses that generate the conflicts are referred to as conflicting clauses.

IV. HARDWARE ACCELERATORS

Nowadays, due to the physical limitations and the clock frequency reaching its limit, the development of dedicated systems i.e., systems which have an optimized structure for a particular task became crucial in order to get higher performance and lower power. This is achieved by using a customized multi-core system, which means that the system has more than one specialized processing element. Another feature of such systems in order to achieve faster information transfer is the personalized on-chip intercommunication which is an important element.

Due to the high cost of the customized ASICs, FPGAs based design has become a com only used part of the complex systems. It allows the engineer or the researcher to design its own customized SoC (*System on Chip*) through the building of specialized low cost processing elements within the imposed time limit. With such advantages, the employment of the reconfigurable systems in order to solve the SAT problem brings no surprise.

In this paper, the SW/HW solution will be emphasized since future work will be done using this approach. This is because using both a personal computer (SW) and a highly customized system (HW) allows us to get higher speeds in sequential tasks (i.e., tasks that do not need a huge amount of memory access and parallelism) using the personal computer and to employ a customized system in order to have a more efficient execution of the highly parallelized hard tasks. Further details about well known fully HW and SW/HW implementations can be found in [22].

A. *Dandalis et al.*

In [23], a dynamical parallel system was proposed for solving implications during DPLL's *deduction* phase. In this system all the clauses are divided in *p* parallel groups (multi-core system) allowing implications to be found independently in each group and then propagated for the next clauses in the same group's chain and later to the other groups. An example with three groups is shown in Fig. 1 – Hardware Accelerator in [23].

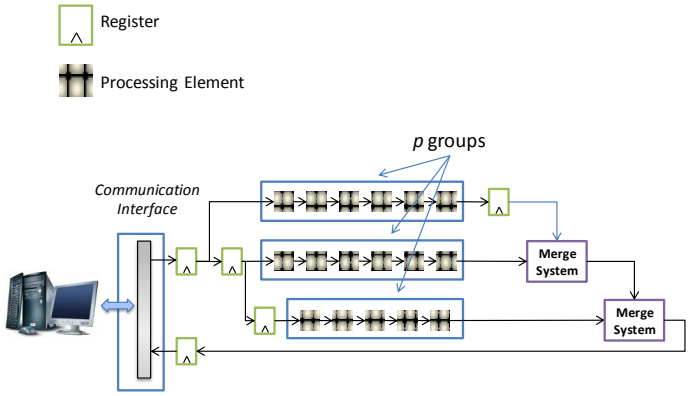


Fig. 1 – Hardware Accelerator in [23]

This system receives a variable that was assigned by the DPLL's decision heuristics in the host computer, and then it will enter in each group. It will then pass through each group chain which is composed of processing elements in order to find implications. Subsequently, a merging process will receive information from each group and, as a consequence, it will decide which will be the next variable to be assigned, repeating this process until no more variables are implicated. Finished this process, all the assignments information made in the dedicated system will be transmitted to the host computer. In case that the merging system detects a conflict, all the attribution process will stop and backtracking will then be performed by the host computer. All processing elements' memories that contain clause information are updated by a partial reconfiguration process.

This system is based on the FPGA partial reconfiguration feature. During the deduction process it will change the number of groups *p* trying to find the best template for the current variable assignment. However, issues like clauses order within the groups and their distribution in the groups were not mentioned. The contribution of this architecture was the attempt to find the best template for a particular instance. The system was simulated but not tested in real reconfigurable platforms, so important aspects like occupied resources, maximum frequency, number of partial reconfigurations and time to solve some benchmarks are not given and thus conclusions cannot be taken.

B. *Skliarova and Ferrari et al.*

In [24], the authors proposed mapping the SAT problem to a ternary matrix with the rows corresponding to clauses and columns representing the variables. The approach adopted consists in finding a vector, *v*, which is orthogonal to all matrix rows. Thus, if the vector *v* exists then the satisfiability condition was reached and each vector element gives the value to be assigned to the correspondent variable. Otherwise, the instance is impossible to be satisfied. The proposed architecture is shown in Fig. :

- **Control Unit –** Processing element which manages the execution;
- **Stack –** Memory used to store variables' assignments in order to support backtracking operations.
- **Matrices –** Four memory blocks for storing the initial matrices and their transpose. Since, ternary matrix cannot be synthesized, an approach of using two binary matrices was employed (one indicating the position of ones and the other storing the positions of zeros).
- **Registers –** Keep rows and columns that were deleted. Registers values are stored in the stack when an assignment is made and loaded if a backtracking is needed.
- **ALU –** Performs some operations such as counting the number of ones/zeros in a row/column, etc.

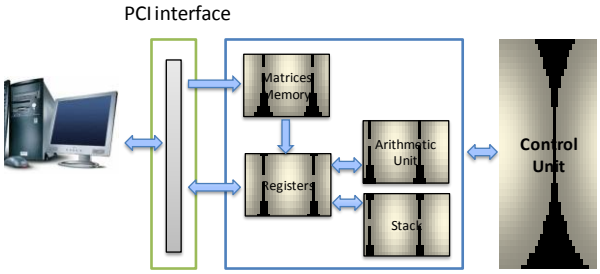


Fig. 2 - Hardware Accelerator in [24]

To this basic structure, an improvement was employed [24-25]. In the latter a hybrid algorithm using a *favorites list* containing some restrictions on the variables assignments was implemented. The solver uses this feature in the search process and when variables assignments are not consistent with the restrictions within the *favourites list*, a premature backtracking is performed.

In this system, the HW/SW partition is quite different from the last example and it is implemented as follows:

1. The problem is implemented by a software application in C++ and then the SAT solver is configured into the FPGA;
2. If the FPGA's memories have enough capacity to accommodate the initial matrices then matrix data will be transferred to the FPGA and all the execution will be done by the customized SAT solver within the reconfigurable device;
3. If not, the host computer begins to solve the SAT instance. At same time it keeps trying to fit the matrices into the FPGA. Due to the variable assignment process, at some point of the execution, that will be possible, and after that, all responsibility will be passed to the FPGA's customized core.

The entire system was implemented and tested with holex SAT benchmarks from DIMACS [26] and a speedup of two orders of magnitude compared to Grasp [6] was achieved for some instances. However, for some benchmarks it does not reach great results comparatively with other solvers.

C. Sousa et al.

In [27-29] a HW/SW system optimized for 3-SAT instances was proposed. This incorporates a conflict diagnosis engine, a backtracking controller and clause database management. The conflicts' engine provides the solver the functionality to learn and as a consequence adds conflict clauses to the original database, pruning search space in future. During problem solving, computationally intensive tasks such as execution of logical implications, selection of the next decision variable and detection of conflicts are employed to the reconfigurable hardware, and soft tasks such as conflict analysis, backtracking control and clause database management are performed by the host computer (software).

The proposed system is based on an *application-specific* architecture i.e., the system does not need to be synthesized for each SAT formula. Another system's feature is that it is based on a *virtual hardware scheme* with context switching. This allows it to accommodate a larger number of clauses with the aid of configuration pages stored in the on-board memory in which each configuration page stores a clause pipeline.

The variables' information is stored in two memory blocks and it is read sequentially from one memory block, then being processed in the clause pipeline and stored in the other memory block. After storing the variables' information in the second memory the next page will be loaded into the clause pipeline and the variables' information is written back to the first memory. A top –level view of the system is shown in Fig. . For a more detailed description see [30].

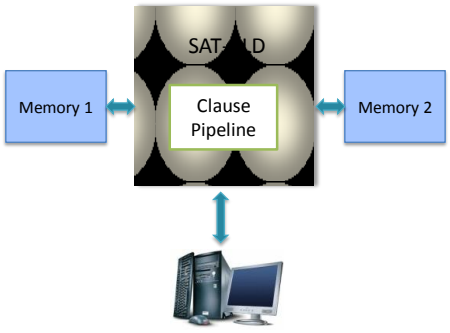


Fig. 3 - Hardware Accelerator in [27-29]

The hardware implementation results were presented in [29]. The system was implemented on the FPGA XCV2000E and four SRAM banks were used. The system achieved a clock frequency up to 47MHz and accommodated up to 7,680 variables and 214,304 clauses. However, the system was not tested because the

communication between the host computer and the FPGAs has not been implemented. Thus, some practical results were not achieved and only simulated results are presented.

D. Davis,Tan, Yu and Zhang et al.

In [31] a new approach for developing HW/SW Sat solvers was proposed relying on BlockRAMs available within a FPGA in order to provide a high parallel and memory bandwidth. This kind of memories allows information to be read or written in just one clock cycle. The proposed design relies on a highly parallel engine system in order to improve the time spent in the BCP.

Let us now analyze the proposed system (see Fig.).

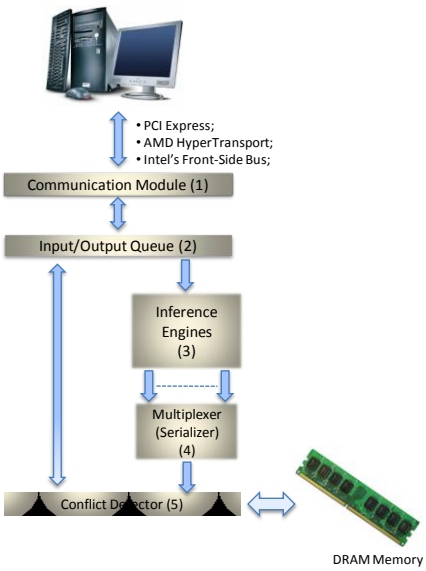


Fig. 4 - Hardware Acceleator in [31]

The system is composed of five components:

1. ***CPU communication module*** **–** This module receives the variable assignment decisions and returns inference results to the host CPU. High speed connections as AMD HyperTransport, Intel's Front-Side Bus and PCI Express were analyzed. In order to maximize the effective bandwidth the authors propose that communications between PC and FPGA should use batches i.e, the entire information must only be transmitted when the transmission buffer is full and then send all the information until the buffer is empty.
2. ***Input/Output Queue*** **–** Receives either the variable chose by the host computer to be assigned or an implication that must be propagated. This module also sends information about the variables implied to the host PC and announces if there is a conflict.
3. ***Parallel Inference Engines*** **–** After a pre-processing process, the clauses that compose the instance are partitioned and distributed by the parallel inference engines. The partition is made following the rule that one variable can only appear once in a parallel engine. Thus, when a variable is assigned, a parallel evaluation can be performed allowing to detect if an implication has occurred. Another feature of this module is the clause search process. The latter receives a variable and a corresponding value to be assigned and searches in a binary tree which is the clause where this variable occurs. The inference engine structure is shown in Fig. .

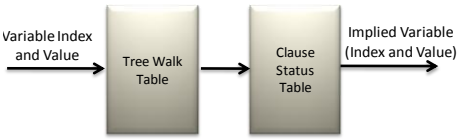


Fig. 5 - Inference Engine in [31]

Observing Fig. , two memories can be seen: the **Tree Walk Table** and the **Clause status Table**. The first, implements the search binary tree while the second maintains the SAT clauses. Then an auxiliary combinatorial circuit tests if an implication was reached giving in this case to the next component (Multiplexer) the index and the value of the implied variable.

For a more detailed description see [31].

4. ***Multiplexer(Serializer)*** **–** This module receives all the information that comes from the *parallel inference engines* and serializes it. It uses a 2-level priority-encoder and can be attached to up to 256 inference engines.
5. ***Conflict Inference Detection*** **–** This module receives the serialized information and detects clauses' conflicts. Given a new implication from the inference engines, it detects if there is a conflict and tests if the same implication was already produced by a different clause. If a conflict is detected, then the system will notify the host computer and both backtracking and a learning process will be invoked. The DRAM memory is used to translate the local index of the implied variable to its global variable and clause ID.

About the implementation results, the authors only present some tests to the communication module and some synthesis results. However, a careful analysis shows that the targeted FPGA does not allow such number of variables and clauses to be supported (216 variables and 216 clauses were reported). In relation to the search process, it is only efficient when we are treating instances with a few number of clauses and variables. When all the memories are full a more efficient implementation is directly mapping the clause without any search process. Another problem in this

system is that it cannot efficiently take advantage of the DDR memory *burst* accesses since in order to detect a conflict or detect which are the implicated variables, a non-continuous memory access must be performed.

V. CONCLUSION

This paper is dedicated to the description and analysis of the *state of the art* techniques and implementations of SAT solvers. It presents the basic template for complete SAT solvers as well as a detailed description of some techniques concerned to this template. Since the use of reconfigurable systems and HW/SW approach is a widely adopted method, some of the implementations that take advantage of this scheme are also presented and a brief analysis is done. We can also conclude that future developments for solving SAT problem should take advantage of the new FPGAs capabilities such as BRAMs, allowing the clauses to be partitioned and accessed in parallel. Despite of these new features, the biggest challenge in order to improve the SAT performance is to determine the best processing elements' architecture, the best clause search process and backtracking parallelization.

ACKNOWLEDGMENT

The author would like to acknowledge Pr. Valery Sklyarov and Pr. Iouliia Skliarova for their constructive comments and help during the study.

REFERENCES

[1] M. Velev and R. Bryant, "Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors," *IEEE/ACM Design Automation Conference*, 2001.

[2] H. Kautz and B. Selman, "Planning as satisfiability," *European Conference on Artificial Intelligence*, pp. 359–363, 1992.

[3] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. Comput.-Aided Design 11*, vol. 1, pp. 6–22, 1992.

[4] I. Mironov and L. Zhang, "Cryptanalysis of Hash Functions," *Microsoft Research*.

[5] A. Gupta and M. Prasad, "A survey of recent advances in sat-based formal verification," *International J. Softw. Tools Technol. Transfer 7*, vol. Vol. 2, pp. 156–173, 2005.

[6] J. P. Marques-Silva and K. A. Sakallah, "GRASP – a new search algorithm for satisfiability," *ICCAD*, pp. 220–227, 1996.

[7] M. W. Moskewicz, *et al.*, "Chaff: Engineering an Efficient SAT Solver," *DAC*, 2001.

[8] H. Zhang, "SATO: An efficient propositional prover," *CADE*, vol. In 14th CADE, volume 1249 of LNCS, pp. 272–275, 1997.

[9] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust sat-solver," *DATE*, pp. 142–149, 2002.

[10] N. E´en and N. S¨orensson, "MiniSat: A SAT solver with conflict-clause minimization," *8th SAT*, 2005.

[11] B. Selman and H. J. Levesque, "A new method for solving hard satisfiability problems," *AAAI*, pp. 440–446, 1992.

[12] B. Selman and H. Kautz, "Local search strategies for satisfiability testing," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. Vol. 26, pp. 521–532, 1996.

[13] M. Davis and H. Putnam, "Computing procedure for quantification theory," *Journal of ACM*, vol. 7, pp. 201-215, 1960.

[14] M. Davis, *et al.*, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394-397, 1962.

[15] J. W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," *Ph.D Thesis, University of Pennsylvania*, 1995.

[16] R. G. Jeroslow and J. Wang, "Solving propositional satisfiability problems," *Annals of Mathematics and Artificial Intelligence*, vol. 1, pp. 167-187, 1990.

[17] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," 2002.

[18] J. P. Marques-Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," *9th Portuguese Conference on Artificial Intelligence*, 1999.

[19] A. Nadel, "The Jerusat SAT solver," *Master's thesis, Hebrew University of Jerusalem*, 2002.

[20] K. Pipatsrisawat and A. Darwiche, "RSat 1.03: SAT solver description," *Technical Report D–152, Automated Reasoning Group, Computer Science Department,UCLA*, 2006.

[21] C. M. L. Anbulagan, "Heuristics based on unit propagation for satisfiability problems," *International Joint Conference on Artificial Intelligence*, 1997.

[22] I. Skliarova and A. B. Ferrari, "Reconfigurable Hardware SAT Solvers: A Survey of Systems," *IEEE Transactions on Computers*, vol. vol. 53, pp. 1449-1461, 2004.

[23] A. Dandalis and V. K. Prasanna, "Run-Time Performance Optimization of an FPGA-Based Deduction Engine for SAT Solvers," *ACM Trans. Design Automation of Electronic Systems*, vol. vol. 7, pp. 547-562, 2002.

[24] I. Skliarova and A. B. Ferrari, "A Software/Reconfigurable Hardware SAT Solver," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. vol. 12, pp. 408-419, 2004.

[25] I. Skliarova and A. B. Ferrari, "A Hardware/Software Approach to Accelerate Boolean Satisfiability," *Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pp. 270-277, 2002.

[26] "DIMACS challenge benchmarks," *ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/*, 2009.

[27] J. Sousa, *et al.*, "A Configware/Software Approach to SAT Solving," *Ninth IEEE Int. Symp. Field-Programmable Custom Computing Machines*, 2001.

[28] R. Ripado and J. Sousa, "A Simulation Tool for a Pipelined SAT Solver," *Proc. XVI Conf. Design of Circuits and Integrated Systems*, pp. 498-503, 2001.

[29] N. Reis and J. Sousa, "On Implementing a Configware/Software SAT Solver," *Proc. 10th IEEE Int'l Symp. Field-Programmable Custom Computing Machines*, pp. 282-283, 2002.

[30] J. d. Sousa, *et al.*, "A Configware/Software Approach to SAT Solving," *Ninth IEEE Int. Symp. Field-Programmable Custom Computing Machines*, 2001.

[31] J. D. Davis, *et al.*, "Practical Reconfigurable Hardware Accelerator for Boolean Satisfiability Solvers," *DAC 2008*, 2008.