Université de Montréal

Parallelization of SAT on Reconfigurable Hardware

An Architectural Exploration of Techniques

Par

Teodor Ivan

Département d'informatique et recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2013

Résumé

Quoique très difficile à résoudre, le problème de satisfiabilité Booléenne (SAT) est fréquemment utilisé lors de la modélisation d'applications industrielles. À cet effet, les deux dernières décennies ont vu une progression fulgurante des outils conçus pour trouver des solutions à ce problème NP-complet. Deux grandes avenues générales ont été explorées afin de produire ces outils, notamment l'approche logicielle et matérielle.

Afin de raffiner et améliorer ces solveurs, de nombreuses techniques et heuristiques ont été proposées par la communauté de recherche. Le but final de ces outils a été de résoudre des problèmes de taille industrielle, ce qui a été plus ou moins accompli par les solveurs de nature logicielle. Initialement, le but de l'utilisation du matériel reconfigurable a été de produire des solveurs pouvant trouver des solutions plus rapidement que leurs homologues logiciels. Cependant, le niveau de sophistication de ces derniers a augmenté de telle manière qu'ils restent le meilleur choix pour résoudre SAT. Toutefois, les solveurs modernes logiciels n'arrivent toujours pas a trouver des solutions de manière efficace à certaines instances SAT.

Le but principal de ce mémoire est d'explorer la résolution du problème SAT dans le contexte du matériel reconfigurable en vue de caractériser les ingrédients nécessaires d'un solveur SAT efficace qui puise sa puissance de calcul dans le parallélisme conféré par une plateforme FPGA. Le prototype parallèle implémenté dans ce travail est capable de se mesurer, en termes de vitesse d'exécution à d'autres solveurs (matériels et logiciels), et ce sans utiliser aucune heuristique. Nous montrons donc que notre approche matérielle présente une option prometteuse vers la résolution d'instances industrielles larges qui sont difficilement abordées par une approche logicielle.

Mots-clés: SAT, solveur matériel, solveur matériel parallèle sur FPGA

Abstract

Though very difficult to solve, the Boolean satisfiability problem (SAT) is extensively used to model various real-world applications and problems. Over the past two decades, researchers have tried to provide tools that are used, to a certain degree, to find solutions to the Boolean satisfiability problem. The nature of these tools is broadly divided in software and reconfigurable hardware solvers. In addition, the main algorithms used to solve this problem have also been complemented with heuristics of various levels of sophistication to help overcome some of the NP-hardness of the problem. The end goal of these tools has been to provide solutions to industrial-sized problems of enormous size. Initially, reconfigurable hardware tools provided a promising avenue to accelerating SAT solving over traditional software based solutions. However, the level of sophistication of software solvers overcame their hardware counterparts, which remained limited to smaller problem instances. Even so, modern state-of-the-art software solvers still fail unpredictably on some instances.

The main focus of this thesis is to explore solving SAT on reconfigurable hardware in order to gain an understanding of what would be essential ingredients to add (and discard) to a very efficient hardware SAT solver that obtains its processing power from the raw parallelism of an FPGA platform. The parallel prototype solver that was implemented in this work has been found to be comparable with other hardware and software solvers in terms of execution speed even though no heuristics or other helping techniques were implemented. We thus show that our approach provides a very promising avenue to solving large, industrial SAT instances that might be difficult to handle by software solvers.

Keywords: SAT, hardware solver, FPGA parallel SAT solver

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic and Logical Unit
ASIC	Application Specific Integrated Circuit
BA	Boolean Analyzer
BCP	Boolean Constraint Propagation
CAD	Computer aided design
CDCL	Conflict-driven-clause-learning
CNF	Conjunctive normal form
DIMACS	Centre for Discrete Mathematics and Theoretical Computer Science
DNF	Disjunctive Normal Form
DP	Davis-Putnam
FPGA	Field programmable gate array
FSM	Finite State Machine
HDL	Hardware Description Language
I/O	Input-Output
IC	Integrated circuit
LE	Logic Element
LFSR	Linear Feedback Shift Register
LUT	Look-up table
MB	Megabytes
MHz	Megahertz
MOM	Maximum Occurrences in Clauses of Minimum Size
OPB	On-chip Peripheral Bus
PLB	Processor Local Bus
RAM	Random Access Memory
ROM	Read-only memory
SAT	Boolean satisfiability problem
UPT	Average time to resolve implications or detect a conflict
VHDL	Very High Speed Integrated Circuit Hardware Description Language

À Yaroslava Chtompel

Acknowledgments

I would like to express my profound gratitude to Professor El Mostapha Aboulhamid for enabling me to pursue my studies in his laboratory where I have gained a great deal of technical and other knowledge in these past few years. I would also like to take this opportunity to thank my family, and in particular my wife Yaroslava, for their support, help and encouragement, without which I would surely not have been able to finish my work.

Chapter 1 – Introduction

The Boolean Satisfiability problem (SAT) is considered to be fundamental in the field of computation. As SAT is NP-complete [26], it is computationally intractable as there exists no known polynomial time algorithm capable of solving problem instances with 3 or more variables per clause (unlike instances with 2 variables in each clause [27]). It should also be noted that SAT's importance is further highlighted by the fact that it was discovered to be at the core of the study of the NP-completeness of computational complexity theory. For example, Richard Karp showed [28] that there exists a polynomial time many-one reduction from SAT to 21 combinatorial and graph computation problems (e.g. KNAPSACK, EXACT_COVER) thereby implying that these problems are NP-complete as well.

From an applied standpoint, SAT is used to model various real-life problems and applications such as automated reasoning, computer aided design (CAD), computer-aided manufacturing, machine vision, robotics, integrated circuit (IC) design and computer architecture design [29] despite its computational hardness. In addition, SAT has also been known to play a role in a great variety of decision and optimization problems which can be thought of as its extensions. Indeed, these problems either use SAT as a core problem solving engine or employ some of its various techniques and methods. Examples of such problems include the Satisfiability Modulo Theories, pseudo-Boolean constraints, maximum satisfiability, model counting and Quantified-Boolean Formulas [30].

1.1 Software SAT Solvers

The tools used to provide solutions to SAT instances are mainly software-based. The past two decades have seen a vertiginous improvement in the ability of these solvers. One of the main driving forces behind the advancement of these SAT solvers has been the SAT Solver Competition [31], a recurring event that is geared toward the objective evaluation of the current progress of state-of-the-art SAT solving techniques. Examples of such successful solvers include GRASP [32], CHAFF [33] and MiniSAT[1]. There exists a multitude of features

[1] Solver site: http://minisat.se

fundamental to the efficiency and fast execution speed of these tools, the four main of which are conflict-driven clause learning (CDCL), random search restarts, Boolean constraint propagation (BCP) by use of lazy data structures and conflict-based adaptive branching [34]. To further challenge the limits of SAT solving, researchers have continued to discover and implement additional performance enhancing techniques such as random restart strategies and conflict clause minimization. These methods have augmented the level of sophistication and complexity of modern SAT solvers. However, it should be noted that despite this rapid advancement there is still some ambiguity in the research community as to the relative usefulness and interactions of all these features as well as to the reasons why software SAT solvers fail to generate solutions on many problem instances [34].

1.2 Reconfigurable Computing and FPGA

Alternatively to software-based solutions, SAT may be analyzed by means of reconfigurable hardware using field programmable gate arrays (FPGAs). Indeed, in the past decade, reconfigurable computing based on FPGA devices has matured into a stable discipline that has provided solutions to computing problems that feature substantial advantages over those offered by traditional multi-purpose processors. In addition to the fact that FPGA devices offer increased flexibility as they can be reconfigured, they are also able to generate, by means of their extensive parallelism, very fast application execution times. It is not uncommon to see accelerations of several orders of magnitude over general purpose processors even though circuit clock speeds are orders of magnitude lower [35]. Other added benefits of reconfigurable devices are their low power consumption and reduced energy (each application's circuitry is optimized for the problem at hand) as well as reduction in component count and size, improved time-to-market and upgradeability [35].

A general overview [12] of the basic FPGA design methodology is illustrated in Figure 1. Initially, a designer will describe the circuit hardware using either a hardware description language (HDL) or a schematic editor. Next, logic synthesis deals with the generation of a netlist of logic gates and other blocks present in FPGA devices that is independent of the intended FPGA technology. At this point, the designer can make use of a functional simulation tool to ensure of the logical correctness of the circuit. Following logic synthesis, the

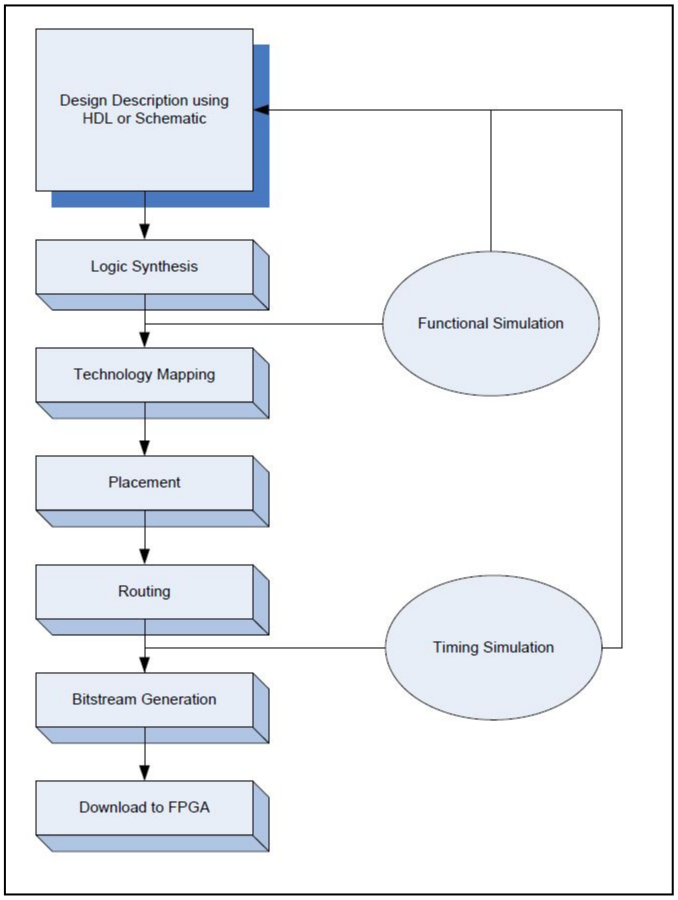


Figure 1: Basic FPGA design flow [12]

previously obtained generic netlist is used to obtain a specialized circuit of look-up tables (LUTs) as it is mapped towards a specific FPGA LUT-based architecture. This process attempts to minimize the FPGA area, the circuit delay and the power consumption as much as possible [36]. During the placement step, physical resources of the FPGA device are selected for the specialized netlist by means of an optimal strategy. Placement is extremely important for maximal circuit frequency and power consumption as it directly influences a circuit's routability [37]. The next step of the design methodology is routing, a difficult process as it is limited to the particular FPGA device's resources such as wires, programmable switches and multiplexers [12]. Finally, before allowing a CAD tool to generate the bitstream necessary to program the FPGA, a timing simulation may be performed to ensure that circuit timing constraints are met. Of course, as errors and bugs are discovered during simulations, the designer may review the circuit's schematic or HDL code in order to correct them.

1.3 Objectives of the Current Work

The Boolean satisfiability problem is inherently massively parallel. As detailed below, verifying that a particular problem clause is satisfied can be done independently of the verification of all other remaining clauses with the only limiting factor being the number of evaluation units available. As such, it is natural to assume that SAT would map very well to FPGAs given their ability to perform many computations in parallel. However, an FPGA board has several limitations and restrictions that impact the amount of parallelism extracted when mapping a problem. Examples of this include the amount of on and off-chip memories that are available, the maximal frequencies at which reading from these memories is possible as well as their number of physical read ports. Thus, one of our principal objectives has been to explore and evaluate the impact of FPGA on-chip memory on SAT resolution as well as the different trade-offs necessary to garner maximal parallelism for execution speed. To characterize these issues, a solver prototype was implemented using VHDL and targeted to an Altera Cyclone II DE2-70 FPGA board. In addition, a software counterpart with an identical execution model was also developed and used to simulate the hardware on problem instances.

As SAT instances come in many shapes and sizes, a focus of the current work has been to allow for many of them to be analyzed easily. Therefore, a secondary objective of this work has been to take advantage of the reconfigurability of the FPGA device so as to create hardware that is specialized (in terms of the size of the resulting circuit) to the problem at hand. This also allows for simpler, tailored solutions that avoid complex control and other structures and whose theoretical limitations are imposed not by the solver's design but by FPGA capacity.

Finally, an important objective of our work has been to identify, explore and evaluate suitable parallelization techniques applicable to the Boolean satisfiability problem. In particular, the methodology used in the experimental section of this work rests heavily upon evaluating potential solutions in one clock cycle. In addition, one of our goals in developing this prototype was to identify bottlenecks and other problem areas in order to propose a solution that would pave the way towards the development of a mature, efficient solver capable of tackling industrial-size instances comprising millions of variables and clauses.

1.4 Contributions

This work has brought forth four main contributions which are listed below.

- One of the most important contributions of this work is the exploration of the various techniques and methods that can be applied when solving SAT on FPGA.
- A testing platform composed of an HDL synthesizable model and a Java software simulator of a hardware SAT prototype relying solely on FPGA parallelism for execution speed was built and characterized.
- Trouble areas that restrict the use of FPGAs in SAT solving have been identified. Examples of these limitations include the limited size of on-chip memory as well as the use of registers (which effectively play the role of a multi-port random access memory (RAM) but claim a large amount of hardware resources).
- We quantify our technique by means of simulation using problems from the DIMACS[2] set and present comparisons with results available from other state-of-the art hardware and software solvers.

1.5 Organization

This thesis is organized as follows. Chapter 1 has offered an introduction to the Boolean Satisfiability problem as well as to reconfigurable computing and FPGAs. The objectives, contributions and organization of this thesis were also presented. Chapter 2 will describe the Boolean Satisfiability problem, the Davis-Putnam-like (DP-like) algorithm used in this work as well as some of its possible enhancements. Chapter 3 presents the various hardware architectures that are employed in solving SAT, beginning with an older Boolean Analyzer (BA) machine and ending with fairly recent hardware solvers. A summary of these tools is presented at the end of the chapter. Chapter 4 will detail the testing platform created to assess our solver prototype and Chapter 5 will present the experimental results obtained during the testing phase. The work will conclude with Chapter 6 as a discussion and future research directions are given.

[2] Obtained from http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

Chapter 2 – The Boolean Satisfiability Problem

The Boolean Satisfiability problem is one of determining whether or not a given Boolean formula φ can evaluate to true. As stated beforehand, SAT is an NP-complete problem as the fastest known algorithms that can solve it require an asymptotically exponential amount of time in the size of φ to either find a solution or to assert that no solution exists. In addition, it is almost impossible to claim that faster algorithms exist although nobody has yet proven the contrary [38].

When considering an formula φ of a SAT instance in conjunctive normal form (CNF), one must take into account three components or sets [29]:

- A set of variables $x_0, x_1, \ldots, x_n$, where n is the number of variables of φ.
- A set of literals. A literal is an appearance of variable x as either itself or its negation $\overline{x}$ (for example x_1 and $\overline{x_1}$ are the two possible literals of variable x_1).
- A set of m distinct clauses:
    - Each clause consists of disjunctions of literals combined with the or ($\vee$) logical connective.
    - The whole Boolean formula φ consists of conjunctions of clauses combined with the and ($\wedge$) logical connective.

To satisfy the Boolean formula φ is to determine if a variable assignment exists such that the conjunctive normal form of φ evaluates to true. Figure 2 below illustrates an example[3] of a Boolean formula φ featuring 5 clauses (represented by $\gamma_{\_}$) and 7 variables. This example will also be used in subsequent chapters for illustrative purposes.

2.1 Problem Representation

SAT is usually expressed in conjunctive normal form. However, an equivalent way of expressing SAT is the disjunctive normal form (DNF). By using De Morgan's law[4] of duality,

[3] Example borrowed and adapted from: http://www.cs.cmu.edu/~mtschant/15414-f07/lectures/grasp-ex.pdf
[4] http://en.wikipedia.org/wiki/De_Morgan's_laws

one can easily obtain one form from the other by simply inverting all literal polarities ($\overline{x}$ becomes x and vice-versa) and by replacing all ∨ logical connectives with ∧ logical connectives. In this manner, $\overline{\varphi}$ is obtained from φ.

A variable is represented in a clause by a literal which can be either false or true. A variable can also not have any influence on a clause because neither of its literals is present. For example, clause γ_2 in Figure 2 has two literals which are true (x_2 and x_5) whereas all other variables are absent. To represent problem clauses, three variable states are needed: present as true, present as false and not present. Thus, 2 bits are required to express this information: "10", "01" and "11". The "00" state is not necessary and therefore it is not used. Figure 3 illustrates this encoding on the previous 5 clauses of formula φ. The matrix is read from right to left and from top to bottom. Each row represents a clause and each column represents a variable. For example, the first row of Figure 3 represents the first clause, γ_0, and

$$\left\{\begin{array}{lcl}\gamma_0 & = & (x_4 \vee x_3 \vee x_1 \vee x_0) \\ \gamma_1 & = & (\overline{x_4} \vee x_3 \vee x_1 \vee x_0) \\ \gamma_2 & = & (x_5 \vee x_2) \\ \gamma_3 & = & (x_6 \vee \overline{x_3} \vee x_0) \\ \gamma_4 & = & (\overline{x_6} \vee \overline{x_3} \vee x_1)\end{array}\right\}$$

$$\varphi(x_0, x_1, x_2, x_3, x_4, x_5, x_6) = \gamma_4 \wedge \gamma_3 \wedge \gamma_2 \wedge \gamma_1 \wedge \gamma_0$$

Figure 2: A 5-clause 7-variable formula

$$\begin{bmatrix} (\text{"11"}) & (\text{"11"}) & (\text{"10"}) & (\text{"10"}) & (\text{"11"}) & (\text{"10"}) & (\text{"10"}) \\ (\text{"11"}) & (\text{"11"}) & (\text{"01"}) & (\text{"10"}) & (\text{"11"}) & (\text{"10"}) & (\text{"10"}) \\ (\text{"11"}) & (\text{"10"}) & (\text{"11"}) & (\text{"11"}) & (\text{"10"}) & (\text{"11"}) & (\text{"11"}) \\ (\text{"10"}) & (\text{"11"}) & (\text{"11"}) & (\text{"01"}) & (\text{"11"}) & (\text{"11"}) & (\text{"10"}) \\ (\text{"01"}) & (\text{"11"}) & (\text{"11"}) & (\text{"01"}) & (\text{"11"}) & (\text{"10"}) & (\text{"11"}) \end{bmatrix}$$

Figure 3: Representation of a Boolean formula φ

the rightmost column displays literal information for variable x_0.

2.2 A Variant of the Davis-Putnam Algorithm

The method used in this work to solve SAT is based on the classical Davis-Putnam algorithm [39]. This resolution-based, backtracking procedure is described by the pseudo-code of Figure 4. The procedure uses a two-dimensional vector termed the Candidate that uses the 3-state encoding scheme discussed above. A stack (last-in-first-out data structure) is also required for backtracking purposes whenever erroneous assignments are made. Lines 4 – 9 represent the initialization section. At line 4, the Candidate is initialized as all its variable positions are assigned "11" to represent the fact that they are free. Lines 5 and 6 show that the stack is

```
 1: Procedure DP-Variant
 2: Input: A set of clauses Γ, a set of variables Ψ
 3: Output: SAT if all clauses of Γ evaluate to true, UNSAT otherwise

 4: Candidate ← all variables free;
 5: Stack ← empty;
 6: TOS ← -1; // top of the stack
 7: υ ← -1;
 8: Candidate(0) ← '0';
 9: Stack.push(0);

10: while(true)
11:   if SAT(Γ) then
12:     return SAT;
13:   else if CONTRADICTION(Γ) then
14:     if Stack.empty then
15:       return UNSAT;
16:     else
17:       TOS ← Stack.pop;
18:       RESET_ASSIGNS(Candidate, TOS);
19:       Candidate(TOS) ← '1';
20:   else if ONE_IMPLICATION(Candidate) then
21:     ASSIGN_IMPLICATION(Candidate);
22:   else
23:     υ ← NEXT_FREE_VAR(Candidate, Ψ);
24:     Candidate(υ) ← '0';
25:     Stack.push(υ);
26:   end if;
27: end while;
```

Figure 4: Pseudo-code of a DP-like algorithm

initially empty and the top of the stack (TOS) is initialized to -1. At line 7, the temporary variable index υ also receives -1 as staring value. Lines 8 and 9 indicate that the procedure starts by first deciding to assign '0' to x_0. Inside the procedure's only while-loop, there are four main sections.

The first section spans lines 22 – 25. Here new decisions are made regarding the assignment of the current next available free variable (obtained from the auxiliary procedure $\mathrm{NEXT\_FREE\_VAR(Candidate,}\ \Psi)$). These variables are termed decision variables and are different from implication variables (described shortly). As soon as the index of the next free variable is determined, it is used as an index into the $\mathrm{Candidate}$ vector where '0' is assigned as a first (arbitrary) try. This index is pushed onto the stack so that the procedure can keep track of the decisions it has made.

The second section of the loop is formed by lines 20 and 21. Here the implications of the decisions made in the first section above are determined. As the procedure makes decisions, more and more literals receive assignments in clauses. At a certain point, a clause can become a unit clause when it has only one literal that is yet unassigned in the Candidate and all its other literals are false. For example, Candidate $("11")("11")("11")("01")("01")("01")("01")$ would cause γ_0, γ_1 and γ_2 to become unit as all three clauses have only one remaining literal, namely x_4, $\overline{x_4}$ and $x_5$respectively and all their other literals are assigned false. Focusing only on γ_0, the implication of having previously assigned x_3 to be false is that variable x_4 in the Candidate must be assigned '1' so that the clause is satisfied. This assignment is carried out by the auxiliary function $\mathrm{ASSIGN\_IMPLICATION(Candidate)}$. This section implements the unit clause rule (also called BCP) which is an important feature that helps to augment the resolution of partial solutions so that the search space is pruned. Moreover, it is important to note that index 4 is not pushed onto the stack as x_4 is an implied variable and not a decision variable.

The next section spans lines 13 – 19. Contradictions are found by the $\mathrm{CONTRADICTION}(\Gamma)$ auxiliary procedure. These arise when the function has found one clause to be false as all its literals evaluate to false. Continuing with the partial Candidate solution $("11")("11")("10")("01")("01")("01")("01")$ from above after implying variable x_4

to '1' we find that γ_1 is now false. Inside this third section, at line 17 the stack is popped and the TOS receives the index of the last decision variable which in this case is 3. All assignments beyond this point (including the offending assignment on x_4) are reset by RESET_ASSIGNS(Candidate, TOS). Since '0' did not work as a decision on x_3 '1' is now tried. Index 3 is not pushed onto the stack this time because both possibilities have been exhausted for variable x_3. Line 14 determines if the problem is unsatisfiable. As the DP-Variant procedure proceeds through the problem's search space, it will examine both possibilities for all decision variables. As both these possibilities are unsatisfactory, the procedure backtracks to earlier and earlier decisions. If all decision variables have been tried in both polarities unsuccessfully, the stack will be empty. This means that no matter what decisions are made on any decision variable, it is impossible to satisfy all clauses and the problem is unsatisfiable.

The last section includes lines 11 and 12 where the SAT(Γ) auxiliary procedure is invoked to determine if all clauses have been satisfied. If all clauses are true or satisfied, the DP-Variant procedure terminates with satisfiable (SAT) as answer. If at least one clause is not yet satisfied SAT(Γ)returns false. A clause can be unsatisfied in two cases. In the first, it can be neither false nor true as not all of its variables have yet received an assignment (considering the example of Figure 2, this situation would arise for clause γ_0 after only x_0 and x_1 would receive "01" or false as assignments; the clause still requires more assignments to be resolved). In the second case, the clause is false because all of its variables evaluate to false. For example, Candidate ("11")("01")("11")("11")("01")("11")("11") would render clause γ_2 false because x_5 and x_2 are both false.

2.3 Enhancements over the Basic DP-Variant

The basic procedure described in the previous section leaves room for improvement by means of several techniques and heuristics that have been proposed by researchers over the years. Some of the more important of these features are described in this section.

2.3.1 Pure Literal Assignment Rule

The pure literal assignment rule consists of examining all literals of clauses that are not yet satisfied and determining if there are any that occur in only one polarity (i.e. either true or false) in all the problem's clauses. The variable that is associated with this literal can thus be assigned '1' if all literals in the remaining unsatisfied clauses are true or '0' if they are false. This method, paired with the unit clause rule described above, are termed reduction methods as they increase the resolution of the partial solution and permit the simplification of the Boolean formula (by satisfying all clauses in which the variable is present) [40].

2.3.2 Non-Chronological Backtracking and Dynamic Clause Addition

The backtracking that is described higher for the DP-Variant procedure is of a chronological nature. In other words, if a contradiction arises, the procedure pops the stack and uses that index to try a different solution. As long as the contradiction is present, the stack is popped and eventually the offending decision is found and the contradiction is resolved. However, the contradiction may not have been caused by the last decision made. Introduced by Marques-Silva and Sakallah in [32], GRASP is a search algorithm that features a more sophisticated method of backtracking and pruning the search space. An analysis is performed on the clauses involved in contradictions so that the level of the actual decision variable responsible for the erroneous situation is identified. In addition, it is also possible to construct one or more conflict clauses (that are added to the initial Boolean formula) that prevent future repetition of the same conflict.

2.3.3 Dynamic Decision Variable Ordering

The DP-Variant procedure described in this chapter makes use of a simple static method of deciding which variable is picked for the next decision as the next free variable is always selected. However, it is also possible to determine during execution which variable should be chosen so that the maximum number of variable implications are generated in an effort to identify contradictions quickly. In [18], Suyama et al. use two heuristics to improve the execution time of their implementation of the Davis-Putnam procedure. The first is termed Maximum Occurrences in Clauses of Minimum Size (MOM). Simply put, this method attempts to find the variable that occurs in most clauses that have only two unassigned literals. The intuition is that choosing and assigning this variable will create many unit clauses. The second heuristic used to dynamically select decision variables is called Experimental Unit

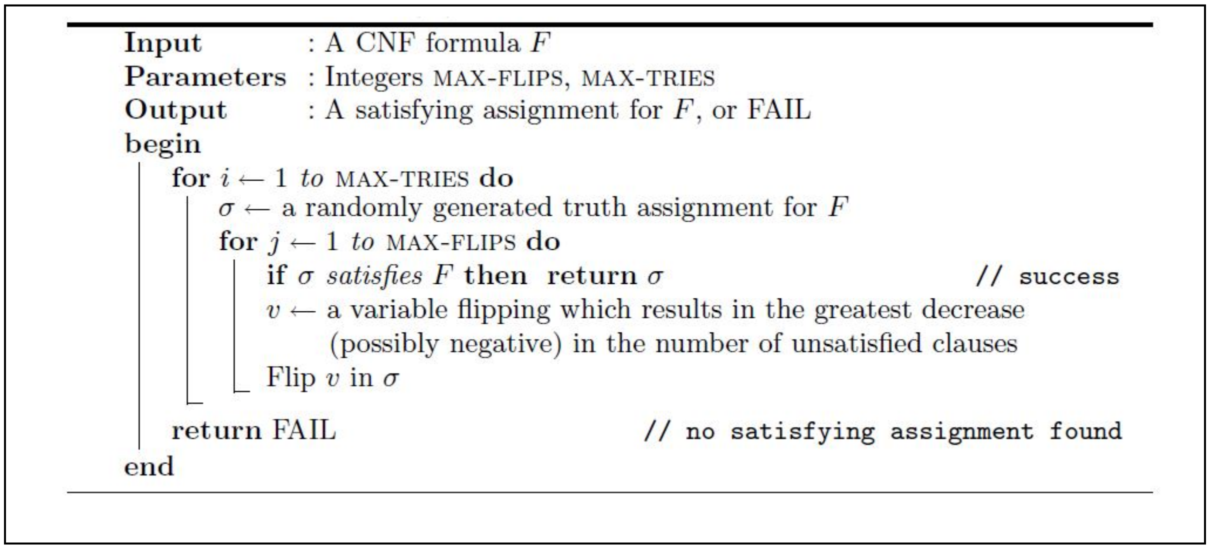
Propagation (EUP). This technique is computationally more intensive as for each unassigned variable both '0' and '1' are tried in parallel. The variable that gives rise to the most unit clauses is hence selected.

2.4 Incomplete Algorithms

The algorithm presented here is a complete method. In other words, given a SAT instance, the algorithm is always able to either find a solution or report that no solution exists.

SAT can also be solved by a different approach. An incomplete SAT solving algorithm is one that can find a satisfying assignment but can never declare the instance unsatisfiable. Unlike the DP-Variant procedure whose main approach is exhaustive branching (decision variables) and backtracking, incomplete algorithms most often use stochastic local search. These algorithms are greatly superior than DP-based approaches on some problem instances [1]. Two of the most important incomplete algorithms that have been highly successful in solving SAT by using local search are GSAT [41] and WSAT (or Walksat) [42].

GSAT, whose algorithm is given in Figure 5, initially begins with a random truth assignment for all variables. It then tries to greedily flip variable assignments that would create the greatest decline in the number of unsatisfied clauses. This is repeated either until a solution has been found or until MAX-FLIPS has been reached. The whole procedure is



```
Input       : A CNF formula $F$
Parameters  : Integers MAX-FLIPS, MAX-TRIES
Output      : A satisfying assignment for $F$, or FAIL
begin
    for $i \leftarrow 1$ to MAX-TRIES do
        $\sigma \leftarrow$ a randomly generated truth assignment for $F$
        for $j \leftarrow 1$ to MAX-FLIPS do
            if $\sigma$ satisfies $F$ then return $\sigma$              // success
            $v \leftarrow$ a variable flipping which results in the greatest decrease
                (possibly negative) in the number of unsatisfied clauses
            Flip $v$ in $\sigma$
    return FAIL                           // no satisfying assignment found
end
```

Figure 5: GSAT incomplete SAT algorithm [1]

repeated MAX-TRIES times [1].

WSAT, shown in Figure 6, further focuses the search by looking in unsatisfied clauses when selecting the variable to flip (the clause from which this variable is chosen is also selected randomly). The “freebie move” is performed when the algorithm finds a variable whose flipping does not cause any currently satisfied clauses to become unsatisfied. If no such variable exists, with a predetermined probability, a random literal is flipped in the randomly chosen unsatisfied clause and with the remaining probability a variable in this same clause is flipped in such a way that the “breakout count”, or the number of currently satisfied clauses that become unsatisfied, is minimized. The parameter p which controls the frequency of non-greedy moves has been empirically found, for various related problem instances, to sometimes have an optimal value. For example, for random 3-SAT formulas, this parameter should be set to 0.57.

```
Input      : A CNF formula $F$
Parameters : Integers MAX-FLIPS, MAX-TRIES; noise parameter $p \in [0,1]$
Output     : A satisfying assignment for $F$, or FAIL
begin
    for $i \leftarrow 1$ to MAX-TRIES do
        $\sigma \leftarrow$ a randomly generated truth assignment for $F$
        for $j \leftarrow 1$ to MAX-FLIPS do
            if $\sigma$ satisfies $F$ then return $\sigma$                // success
            $C \leftarrow$ an unsatisfied clause of $F$ chosen at random
            if $\exists$ variable $x \in C$ with break-count = 0 then
                $v \leftarrow x$                                         // freebie move
            else
                With probability $p$:                                  // random walk move
                    $v \leftarrow$ a variable in $C$ chosen at random
                With probability $1 - p$:                              // greedy move
                    $v \leftarrow$ a variable in $C$ with the smallest break-count
            Flip $v$ in $\sigma$
    return FAIL                              // no satisfying assignment found
end
```

Figure 6: WSAT incomplete SAT algorithm [1]

Chapter 3 – Hardware Architectures for Solving SAT

Generally, there have been two kinds of architectures that have been explored by researchers when implementing hardware SAT solvers, namely instance-specific and application-specific approaches. Instance-specific solvers have their circuits specialized for the problem at hand and thus need to be recompiled or reconfigured before every execution. Application-specific solvers feature a more general construction and can solve all SAT instances without needing to be recompiled. There are benefits and caveats to both these approaches such as long compilation times for the former and access to memory for the latter. A very comprehensive review focusing on detailed architectural aspects as well as on programming models is presented by Skliarova and Ferrari in their 2004 work [40]. There seems to have been a gradual shift towards instance specificity from application specificity. This chapter will therefore take a historical approach at presenting relevant previous works.

3.1 Some History – The Boolean Analyzer

Perhaps one of the earliest attempts at utilizing specialized hardware to treat Boolean equations and formulas was undertaken in 1968 by Antonin Svoboda as he proposed a Boolean Analyzer (BA) [3] capable of producing all the prime implicants of a Boolean formula given the Disjunctive Normal Form (DNF) of its complement. Svoboda's main goal in building the BA was to achieve accelerations over the then fastest general purpose computers which displayed execution times much longer than acceptable. The author identifies some the factors responsible for BA accelerations as being the parallel processing of a large number of Boolean terms and the reduction of Boolean information processing to a repetition of the same operation whose result is only a single bit of information. In order to parallelize the design and achieve accelerations considering the factors mentioned above, the author makes use of triadic and binary counters, a main memory where information for each term is stored, logic gate control and special processing registers that are able to evaluate Boolean terms in parallel. Figure 7 shows the logical organizations of the BA (left) and of the processing register (right). Essentially, the BA uses the triadic counter to generate all 3^n possible candidate solutions for a Boolean function $\overline{y}$ of n variables. This triadic counter is necessary to express the three possible states that a variable can find itself in in a given

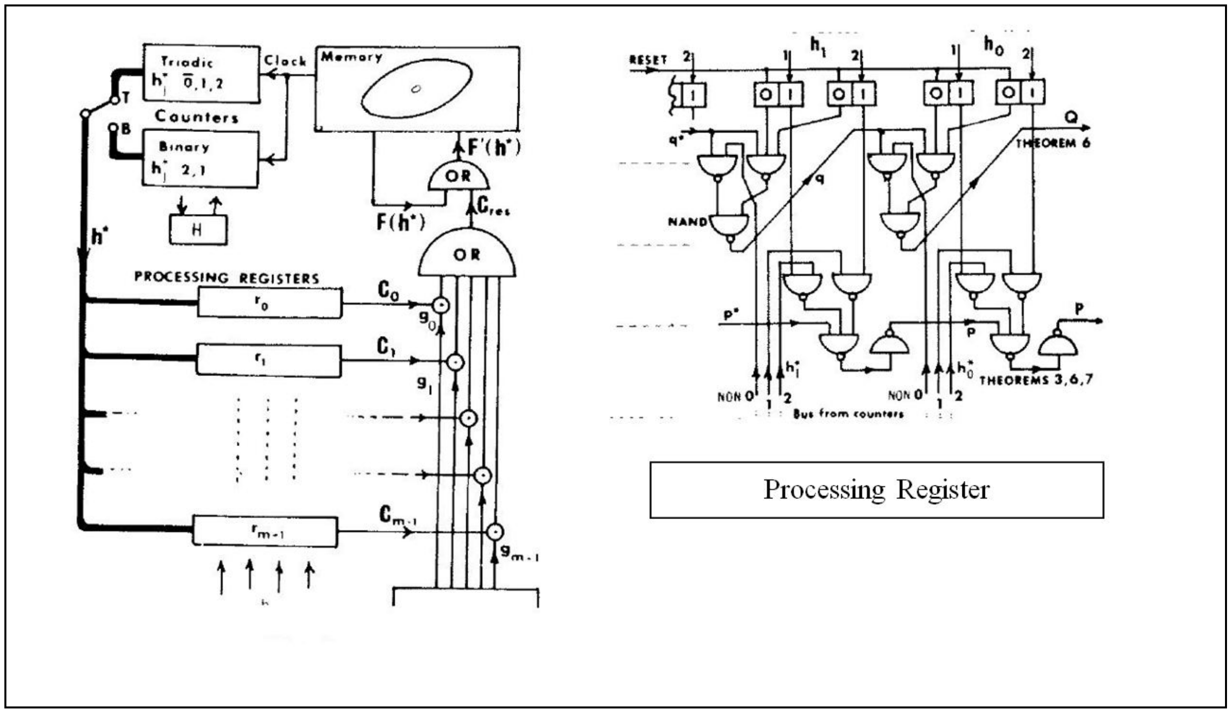


Figure 7: Logical designs of the BA and of its processing register [3]

minterm: assigned true, assigned false or not present. For example, consider the following 4-variable Boolean formula $\bar{y}$ taken from Svoboda's 1968 work [3]:

$$\bar{y} = x_2\, \bar{x}_1\, \bar{x}_0 + \bar{x}_2\, x_1$$

There are two minterms which are encoded using a ternary scheme as follows: 0122 and 0210. To represent the fact that the variable is not present (for example, x_3 does not appear anywhere in the formula) 0 is used. If the variable is present as itself (for example x_2 in the first minterm), 1 is used. Finally, if the variable is present in its negated form ($\bar{x}_0$ in the first minterm) 2 is used. Therefore, in order to be able to represent these 3 states, 2 bits are needed.

Candidates are checked against the terms of $\bar{y}$ to find disjunction points. If a candidate solution is disjoint from all terms of $\bar{y}$ it is not included in it but is included in y. The memory is used to store intermediate and final results for each candidate examined. The binary counter is used by the BA during its solving of Boolean equations (not shown here).

The BA is attempting to solve a very difficult problem which is harder than SAT and as such is unsuitable for tackling any SAT instances modeling real-life applications whose sizes sometimes exceed millions of variables. Because the BA is examining all possible terms of a Boolean formula as opposed to all possible minterms, its search space is composed of 3^n possibilities instead of 2^n. The universe of possible solutions is explored in a brute-force manner and therefore has a time complexity which is exponential. In addition, there is also an explosion in the amount of memory resources needed as they also grow in exponentially.

3.2 Instance Specificity – The First Generation of Solvers

By their nature, FPGAs offer great flexibility. As SAT instances can vary greatly in their size and complexity, it was natural for researchers to initially explore an instance-specific approach when creating SAT solvers whose logic circuit was uniquely tailored after the problem at hand. In this manner, the various variable and clause relationships were imbibed into a specialized circuit that was built each time to solve one instance.

Amid successful early efforts exploring the applicability of FPGAs to SAT solving, Suyama et al. suggested an instance specific approach [19], [9], [17], [18] that is capable of finding all or a fixed number of solutions to a SAT instance. The method is complete as the solver is capable of determining if at least one solution exists. The authors' design flow (Figure 8) entails initially having a C program examining a file containing the description of a SAT instance and generating a high-level HDL behavioral description of the problem at hand. The logic circuit thus described is subsequently analyzed, synthesized and mapped by a CAD tool. In order to increase the efficiency of their parallel algorithm, the authors make use of a

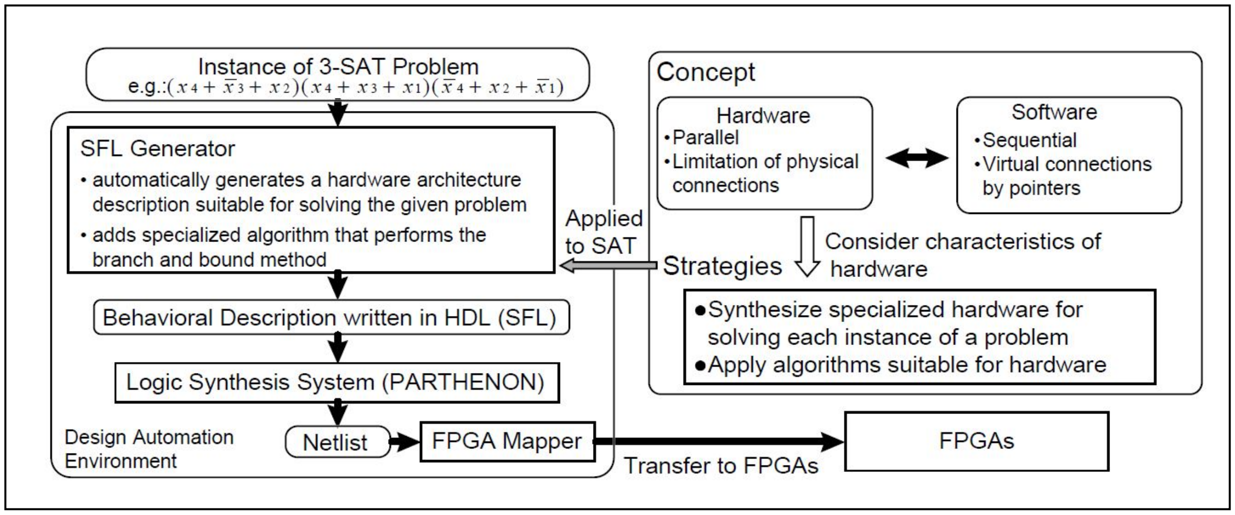

Instance of 3-SAT Problem
e.g.:$(x_4+\bar{x}_3+x_2)(x_4+x_3+x_1)(\bar{x}_4+x_2+\bar{x}_1)$
SFL Generator
• automatically generates a hardware architecture description suitable for solving the given problem
• adds specialized algorithm that performs the branch and bound method
Behavioral Description written in HDL (SFL)
Logic Synthesis System (PARTHENON)
Design Automation Environment
Netlist
FPGA Mapper
Transfer to FPGAs
FPGAs
Applied to SAT
Concept
Hardware
•Parallel
•Limitation of physical connections
Software
•Sequential
•Virtual connections by pointers
Strategies
Consider characteristics of hardware
●Synthesize specialized hardware for solving each instance of a problem
●Apply algorithms suitable for hardware

Figure 8: Suyama et al. flow of logic circuit synthesis [9]

static variable ordering technique in [19] and two dynamic variable ordering techniques in [9], [17] and [18]. The first of these ordering techniques is termed experimental unit propagation (EUP) and entails assigning both possible values to a variable and verifying these assignments concurrently. The second technique, named maximal occurrences in minimum length clauses (MOM), deals with selecting variables occurring in a maximal number of binary clauses [9]. In addition, the approach is characterized by the avoidance of using memory for backtracking purposes as each variable is assigned a register that records the depth of the search tree where its value was decided. The reason offered for this design choice is that the memory required for a stack can become quite large and sequential memory accesses can introduce undesired latencies and bottlenecks. The authors report an implementation status [18] of the mapping of a SAT instance featuring 200 variables and 320 clauses on 21 FLEX10K250 FPGA chips. However, even though the total logic utilization is about 13%, the resulting circuit requires a large amount of wiring resources and therefore necessitates all 21 FPGAs. The authors also report an FPGA implementation of a 128-variable, 256-clause circuit that was synthesized at 10 MHz.

Similarly, Zhong et al. were able to develop a series of instance-specific SAT solver architectures [20], [15], [21] (Figure 9) all based on the DP algorithm [39]. The authors identify implication and conflict checking [15] as the most computation intensive tasks (from a software solution's standpoint) and try to focus their efforts to generating hardware able to accelerate these areas of the basic DP algorithm. The architecture presented in [15] is based on having one finite-state machine (FSM) (Figure 9 top) for each variable keeping track of its state (assigned logical '0', '1' or free). All FSMs are connected in a serial chain and, at any one time, only one FSM is active. If the right-most FSM attempts to pass control to its right, a solution has been found. On the other hand, if the left-most FSM wishes to pass control to its left, the SAT instance has been found to be unsatisfiable. Like Suyama et al., the authors use a C program to generate a VHDL model from the problem's specification. The implication circuit is formula-specific whereas the FSM circuit, since it is identical for all problems, is designed from components that are reused in all circuits. The order of the FSMs on the serial chain is determined statically before execution and variables are sorted depending on the number of their occurrences in the SAT formula. This architecture implies that only one

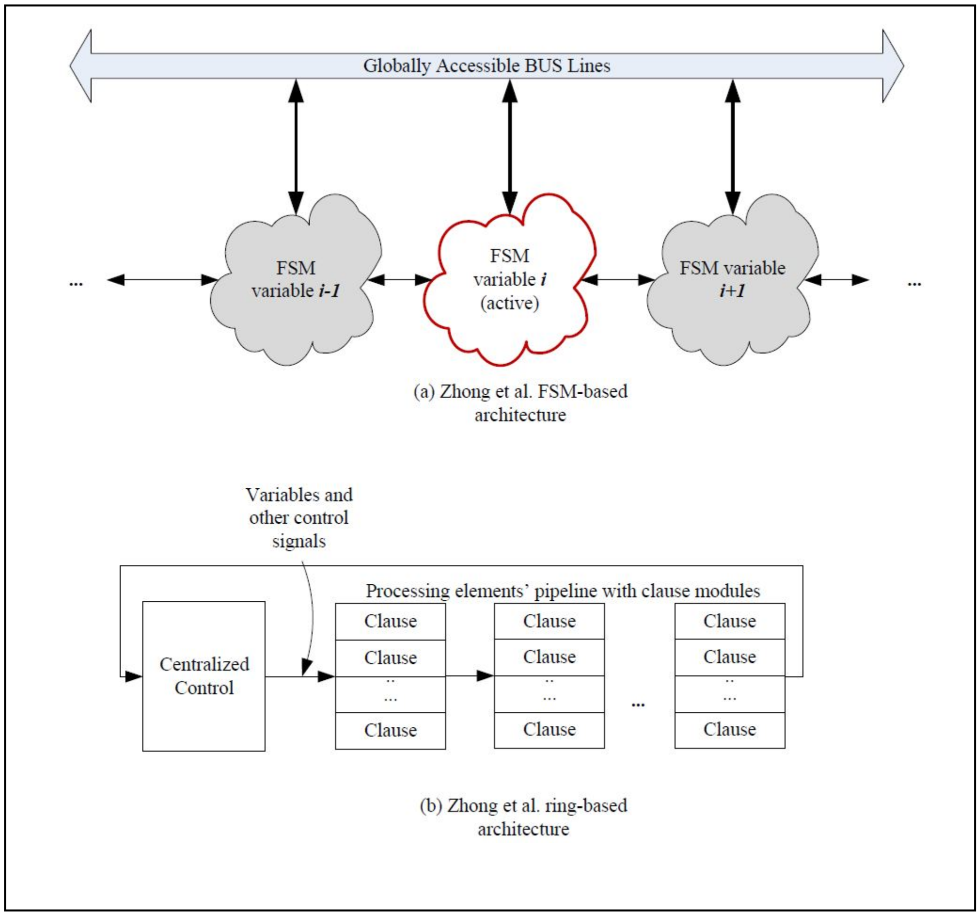

Globally Accessible BUS Lines
...
FSM variable i-1
FSM variable i (active)
FSM variable i+1
...
(a) Zhong et al. FSM-based architecture
Variables and other control signals
Processing elements' pipeline with clause modules
Centralized Control
Clause
Clause
Clause
(b) Zhong et al. ring-based architecture

Figure 9: Zhong et al. hardware SAT solver architectures [15], [21]

variable can receive an assignment at any one time, either due to backtracking, implication or decision. It should be noted that even though the design obviates the need for a backtracking stack, it introduces an unnecessary delay when backtracking as the solver backtracks not to the most recently decided variable but to the variable immediately to the left of the currently active FSM. The key drawbacks of the work presented in [15] are a low clock frequency of 0.7 – 2.0 MHz as well as very long compilation times of up to several hours on their Sun 5 machine featuring 110 MHz and 64 MB of RAM. In [20], non-chronological backtracking is used to further enhance the acceleration that is obtained over the software solution (GRASP solver). In order to resolve the low clock frequencies and high compilation times, Zhong et al. altered the architecture of their solver to a regular ring-based interconnect [21] with centralized control (Figure 9, bottom). This new architecture uses repeated clause modules on a pipelined bus which allows for higher clock speeds of about 30 MHz. However, at any one time, only a subset of the problem's clauses care evaluated in parallel. In addition, compilation times decreased drastically due to incremental synthesis and place-and-route (using Xilinx tools). To this effect, a reusable template for general components is first created. A Java program examines a SAT formula as well as a bitstream file describing this generic template of reusable modules. This is followed by a customization step to generate the appropriate logic function programs and routing connections. The result of this customization can then be downloaded onto the FPGA. The whole process is said to require only a few seconds.

In an approach similar to that proposed by Zhong et al. in [20], Platzner et al. [24], [8] also make use of an array of finite-state machines, each associated with one variable, that are connected in such a manner that an FSM can activate its top or bottom neighbors (Figure 10). A combinational datapath circuit takes variables states as input and computes information that is resupplied to the FSMs. There is a global controller that initiates computations and ensures proper I/O communication. The datapath portion of the circuit and the number of FSMs are

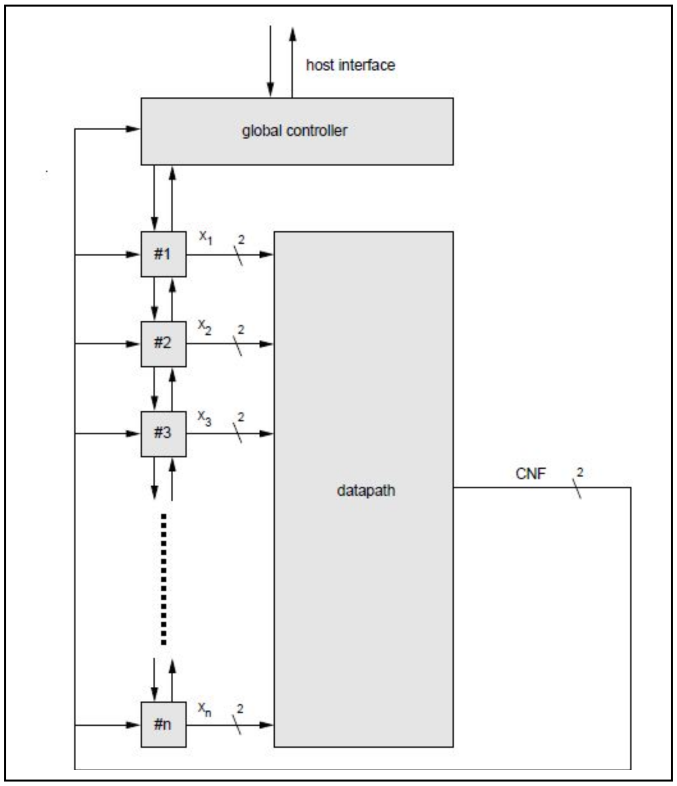


Figure 10: Platzner et al. FSM-based architecture [8]

instance-specific whereas the global controller and FSM structures do not change. At the outset, all variables are unassigned, which leads the datapath to compute its result (CNF line in Figure 10) as unassigned ('X'). In this case, following activation by the global controller, the first FSM (#1) assigns '0' to its variable and verifies the CNF line. If '1' is observed, the partial assignment satisfies the Boolean formula and a solution has been found. On the other hand, if '0' is observed, the variable assignment does not satisfy the Boolean formula and the FSM complements its value and reuses the datapath once again. If 'X' is observed, the partial assignment is neutral and neither satisfies nor unsatisifies the Boolean formula and the FSM activates the FSM below it. If both '0' and '1' assigned to a variable yield a CNF value of '0', control is passed (backtracking) to the FSM above. If the first FSM activates the global controller, the Boolean formula is found to be unsatisfiable. Once again, as was encountered by Zhong et al., the [24] architecture suffers from long compilation times (between 103 and 597 seconds) which dominate the time required to find a solution (e.g. it takes 0.005 seconds to find a solution to problem hole6 of the DIMACS set but 103 seconds to compile the circuit). It should be mentioned that the maximal circuit frequencies for [24] found were found to be between 27 and 65 MHz and therefore somewhat higher than those found by Zhong et al. (for all their architectures reviewed here). As there is no implication computation in [24] the authors use a C simulator program to evaluate enhancing the architecture with this and other features (such as the identification of unassigned variables in satisfied clauses, termed don't care variables) [8]. Simulation results for the architecture performing implication computation show several orders of magnitude speedup over the GRASP software solver although many of the problems tested would not fit onto the FPGA used (XC4020). Finally, physical synthesis results that take into account compilation times showed speedups between 0.03 and 6.54 on the problems from the hole benchmark class against the same GRASP software solver.

A radically different approach proposed by Abramovici et al. [2], [22] makes use of the PODEM algorithm [43] that is employed in the solving of test generation problems. The central concept of the approach is the objective which is the desired assignment of a value to a signal that is currently unknown. The main goal of the procedure is to assign the value '1' to the primary output of the combinational SAT circuit by finding the appropriate values for primary inputs (variables). For this purpose, a backtracing procedure is used to propagate an objective along a single path [2] to a primary input so that the objective may be achieved. An important feature of the approach is that primary inputs can be assigned in any desired order and pure literals are identified dynamically. For backtracking purposes, a hardware stack is used and a central control unit is in charge of performing the search process. The method allows for fine grain parallelism as all operations between successive division steps are done in one clock cycle [2]. Figure 11 (a) shows a high-level block diagram of the architecture proposed in [2]. However, the PODEM algorithm requires the propagation of objectives backward in the circuit whereas a hardware circuit always propagates values forward. To solve this issue, the authors use two distinct element models: every gate of the original circuit is mapped into a forward element and a backward element, each belonging to the forward and backward networks respectively. The authors conclude that the approach is hardware intensive as just the forward network alone is almost twice the size of the initial SAT circuit and the complete circuit is about ten times larger. In [22], a revised architecture is proposed (shown Figure 11 (b)). The Variable Logic block transforms (either implications or decisions) objectives into and maintains current variable assignments. These assignments are sent to the Literal Logic block whose job it is to distribute them to the Clause Logic block. The latter block determines clause values, the output of the overall formula and determines all literal objectives which are sent backwards to the Literal Logic block where they are merged with all other objectives (that are arriving for the same variable) into a single objective for a single

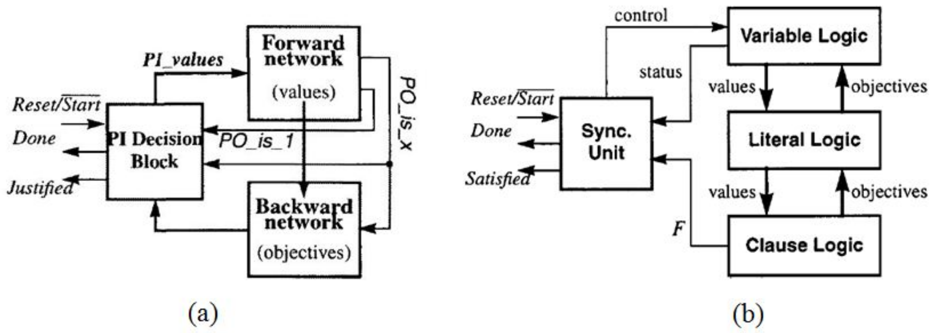


Figure 11: High-level view the architectures proposed by Abramovici et al. [2], [22]

variable. As an enhancement over the previous work where only one objective was propagated along a single path, backtracing of all objectives over all possible paths is now done in parallel and several variables are assigned concurrently. In addition, in [22] objectives are propagated with different priorities and unate variables (pure literals) are identified dynamically. It should also be mentioned that the authors propose a partitioning scheme as SAT problems are decomposed into independent sub-problems to be processed in parallel or sequentially [22]. Finally, it appears that this approach is also hardware demanding as a 13-variable, 29-clause, 69-literal circuit ran at 3.5 MHz. and occupied the whole area of a XC6264 FPGA.

3.3 Application Specificity – Software/Hardware Hybrid Solvers

As it became evident that instance-specific approaches resulted in resource-hungry solutions, researchers turned their attention to a different programming model, namely the application-specific paradigm which entails creating a circuit that is compiled once and that is able to treat different SAT instances. These solvers rely on memory modules to store a problem's clauses.

An interesting solver based on dynamic learning and FPGA reconfiguration is presented by Dandalis et al. [23], [14]. This approach is similar to that of Zhong et al. [21] in that it uses chains of pipelined clause modules (Figure 12). A given CNF formula is split into a fixed number of modules (groups of clauses) and thus implications and conflicts are deduced in parallel. Merge units are then used to combine information from the pipelines into a variable set that is consistent. These variables are then fed back to the pipelines until no more implications occur or a conflict has been detected. It is worth noting that only the implication process is executed in hardware with decisions and backtracking done in software. Each clause module has a memory unit that contains each variable's current value (which can be '1', '0', undecided or conflicting). A novel contribution of this work is the fact that the circuit evolves dynamically as it tries to find, by means of previously constructed FPGA template configurations, the minimal time needed to resolve implications or to raise conflicts (UPT). In order to do so, the solver relies on information computed during execution. A greedy heuristic executed by a host computer is used for this purpose. At the beginning, the template having a minimal number of clause modules per group (and thus the highest level of parallelism because more clause pipelines are created) is used to configure the FPGA. The UPT of this configuration is retained and the host computer decreases the level of parallelism of the next template by increasing the amount of modules per pipeline group. If the newly determined UPT is smaller than the previous minimum, it is retained and reused for the next round of implication and conflict determination. As the host machine reconfigures the FPGA with different templates, the hardware search process does not restart at the root of the search tree but continues from where the previous template had left off. Thus, the optimization process involves finding the level of parallelism that leads to a maximal speedup compared to the baseline solution which uses only one pipeline group for the complete SAT instance. Using a simulator, the authors report, on SAT instances from the DIMACS set, speedups ranging from 1.06 to 5.44 over the baseline solution.

Contrary to the complete nature of the solutions provided by the solvers examined so far, Leong et al. examined not only such a solution in [44] but also turned their attention to incomplete algorithms [4], [16] which can generate solutions but cannot pronounce a SAT

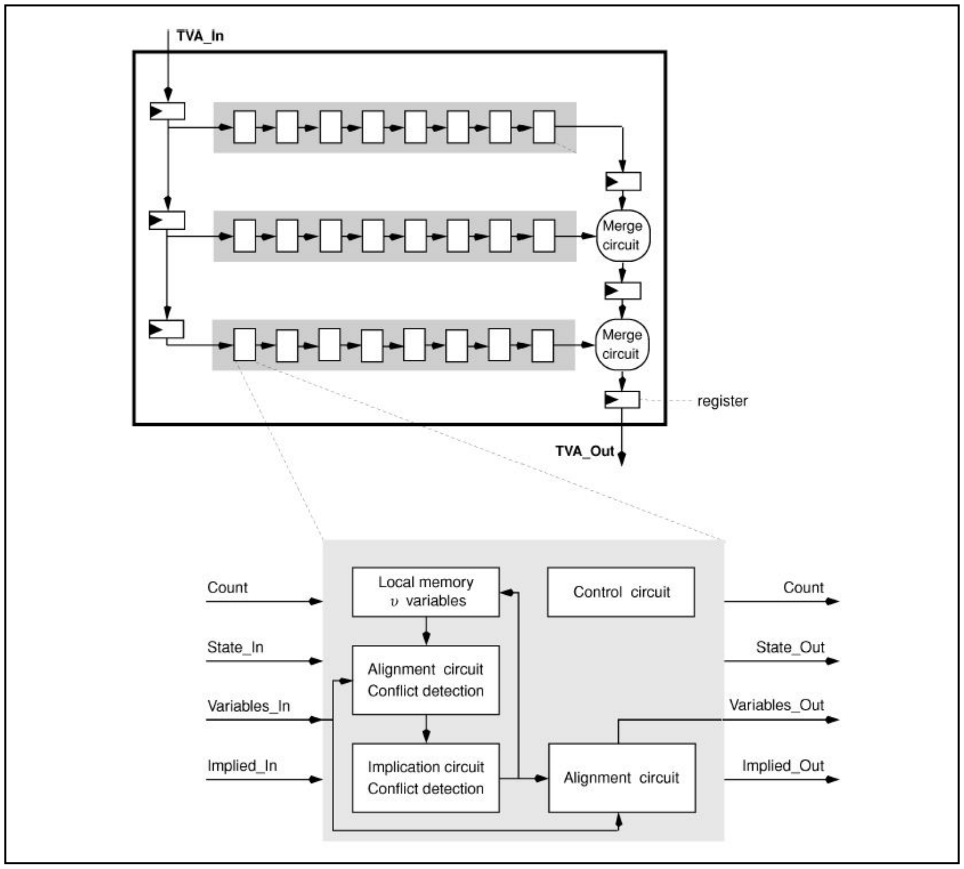


Figure 12: Dandalis et al. SAT deduction engine and clause module details (bottom) [14]

instance unsatisfiable. The work presented in [44] is based on a forward checking tree search method and will not be presented here.

In [4] the incomplete search heuristic GSAT (Figure 5, page 12) was implemented for 3-SAT problem instances (Figure 13). The approach deals with modifying the low-level FPGA configuration bitstream while taking into account the SAT specification. The clause checker in Figure 13 is problem dependent and is customized by a C program according to the SAT instance. Xilinx tools are then used to partially and dynamically reconfigure the FPGA bitstream. The rest of the circuit is generated from a VHDL description and is common to all instances. This eliminates the need for synthesis, mapping, placement and routing of the entire circuit. The inner for-loop of the GSAT algorithm is implemented in hardware whereas the outer for-loop is performed by software. The process starts with a new variable assignment which is downloaded onto the FPGA board. The hardware takes over and, by flipping variables, tries to find the assignment that would create the greatest number of satisfied clauses. If a solution is found, the algorithm terminates. If not, a new random variable assignment is made by software and the process restarts (for a predefined number of times). The solver was tested on a small problem from the aim suite of the DIMACS set but no accelerations were observed when compared to a software implementation of GSAT [4].

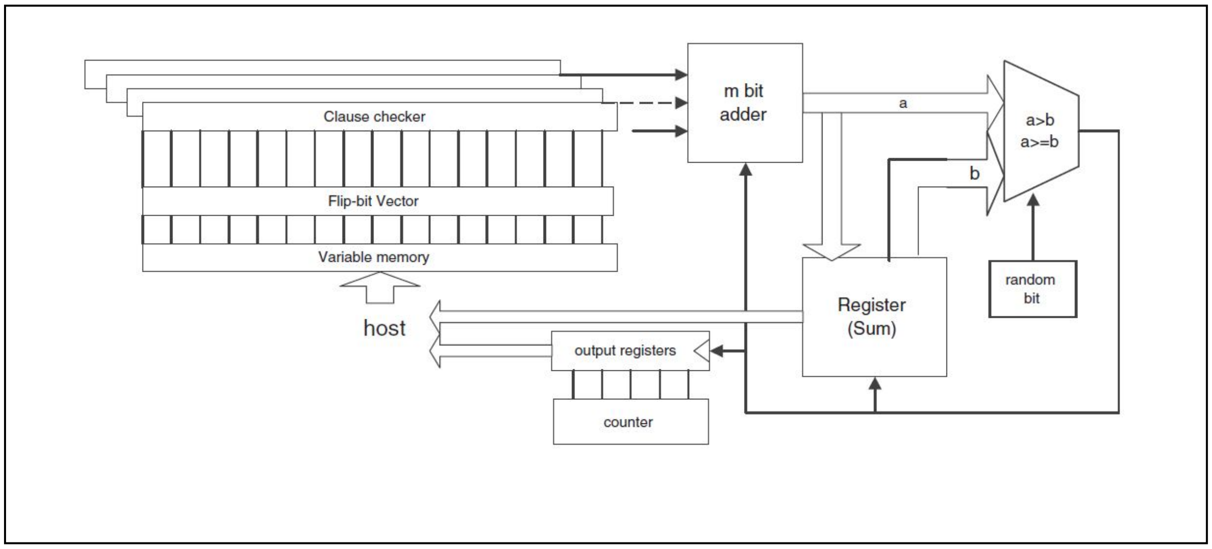


Figure 13: Leong et al. GSAT hardware implementation [4]

In [16] Leong et al. addressed the WSAT incomplete algorithm (Figure 6, page 13) and proposed a solver capable of accommodating 3-SAT problems of maximum 50 variables and 170 clauses (Figure 14). As was done for GSAT, the inner loop of WSAT was implemented in hardware and a software host was made to call the GSAT core a predefined number of times with random variable assignments. The clause checker portion of this core also requires configuration at runtime. This method directly manipulates bitstreams to generate FPGA configurations and the usual FPGA synthesis, place-and-route and mapping operations are avoided once again. The results obtained for this solver show accelerations (versus a software implementation of WSAT [16]) on problems from the DIMACS set of 0.1 to 3.3. The core was built to run at 33 MHz.

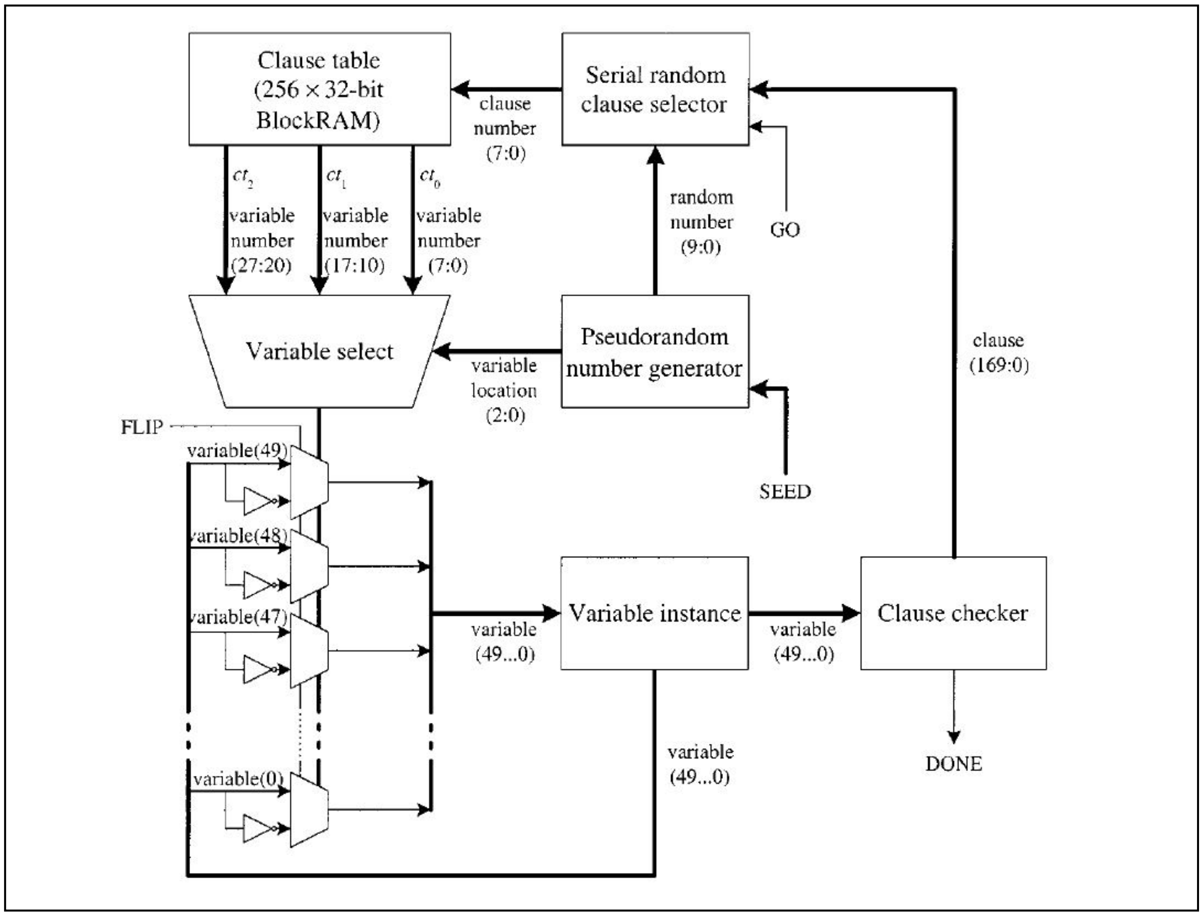


Figure 14: Datapath of Leong et al. WSAT core [16]

3.4 Application Specificity – Modern Solvers

With the work of de Sousa et al. [5], [25] it is possible to see that while hardware SAT solvers still present a hybrid hardware/software approach based on an application-specific programming model, partitioning of the SAT instance into sub-problems starts to become apparent.

Indeed, de Sousa et al. propose a configurware/software approach (Figure 15). Their SAT solver features dynamic conflict diagnosis and conflict clause identification which are done in software whereas implication computation and identification of decision variables (using heuristics) are done in hardware. When problem instances do not fit completely in hardware, the authors use context switching. They divide the original circuit in pages that are successively loaded with intermediate results from RAM modules. A high-level view of the SAT core is shown in Figure 15. The shaded portions are implemented in hardware. Of note is the use of a clause pipeline (Figure 16) that is architecturally similar to that used by Zhong et

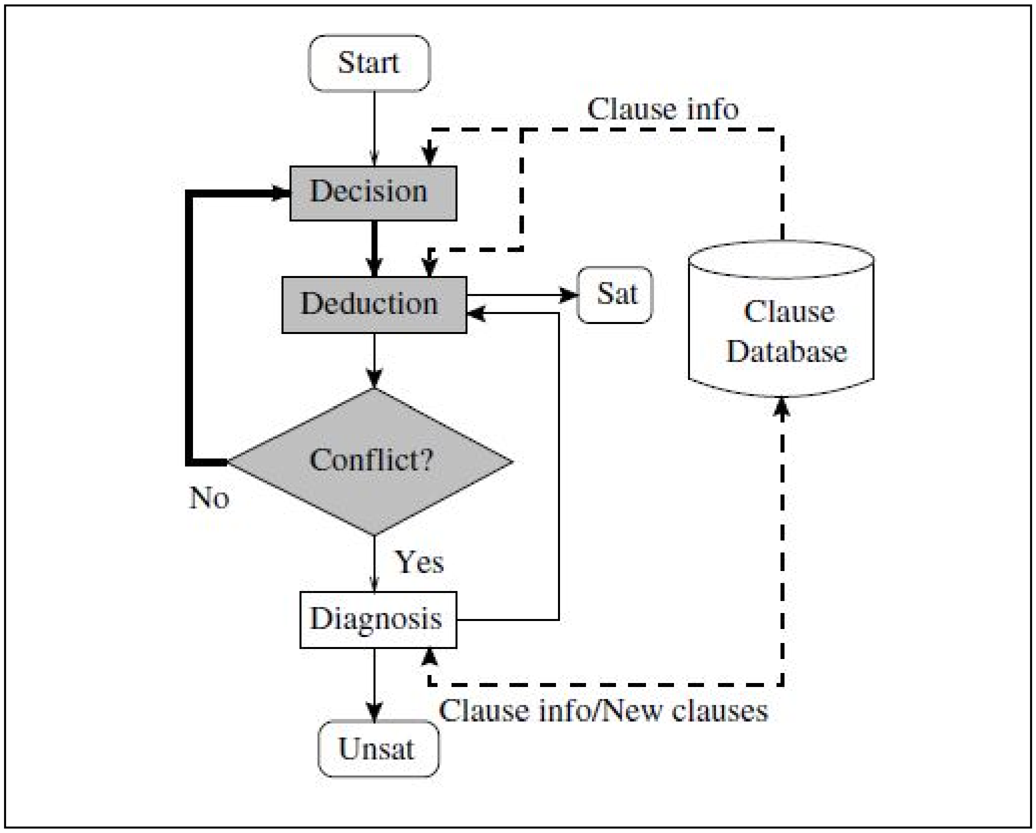


Figure 15: Configuware/Software SAT solver proposed by Sousa et al. [5]

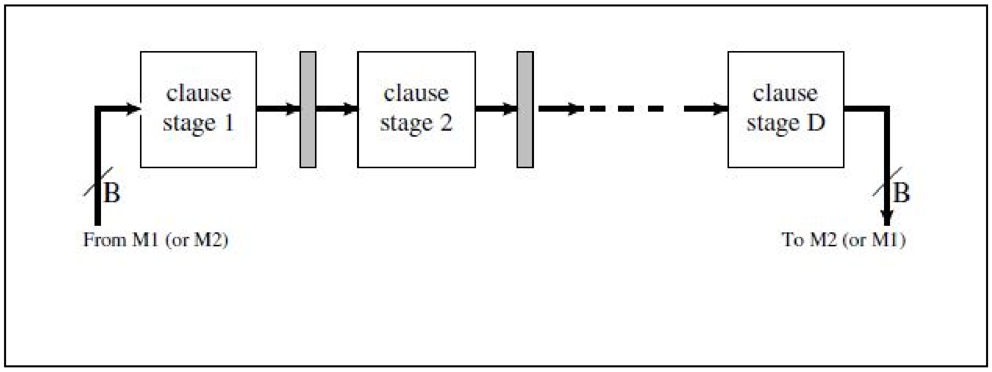


Figure 16: Sousa et al. clause pipeline [5]

al. in [21]. There are several engines that compose the SAT core. The first is the decision engine that, based on a predetermined heuristic, selects one variable for decision. The deduction engine's task is to determine all implications that arise due to a decision by using the unit clause rule and also signals conflicts. If conflicts have been generated, the diagnosis engine analyzes them and constructs a set of offending variables. The variable (belonging to this set) that was assigned most recently is complemented and all implications that arose from it, as well as all subsequent decisions are reset. New implications that were created by the complementation are found and the search continues. In [25], the authors use an XCV2000E FPGA board (97% resource usage) and report that SAT instances of up to 7680 variables and 214304 clauses can be processed. However, the software interface was not yet completed and thus no execution times were reported.

Another prominent example of researchers trying to avoid long hardware compilation times can be seen in the 2004 work of Skliarova and Ferrari [13]. This approach involves using functional units such as registers and arithmetic and logical units (ALUs) to construct a SAT co-processor that avoids being instance-specific (Figure 17). A hardware template circuit based on a ternary matrix that is able to accommodate a SAT sub-problem of a specific size was built. If the sub-problem (appearing at a specific level of the search tree) can fit onto the FPGA, it is downloaded and the aforementioned circuit, implemented on the basis of a DP-like algorithm, is used to solve it. If the sub-problem does not fit, software is used to simplify and reduce it so that it may be downloaded again. Hardware resources are used efficiently in this manner, although the success of instance partitioning between software and hardware is tied to

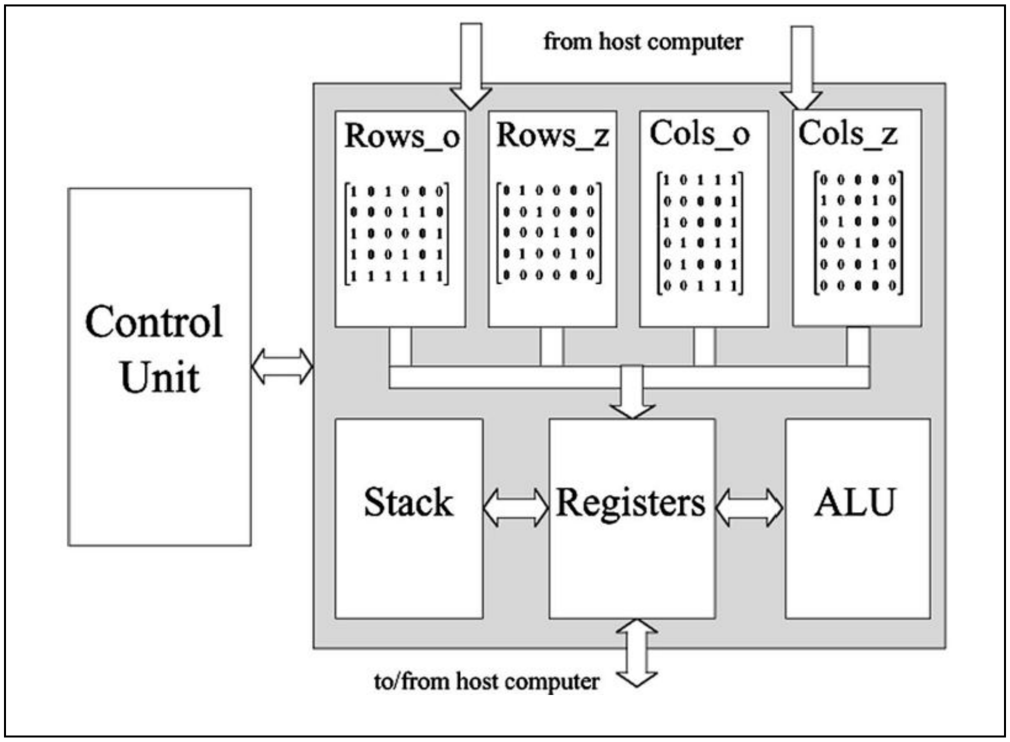


Figure 17: Skliarova and Ferrari's matrix based solver architecture [13]

the nature of the SAT problem at hand. The authors implemented three different circuit templates, all of different sizes. The largest circuit occupied 54% of the XCV812E FPGA and featured a maximal frequency of 30.516 MHz. Comparisons were made with the GRASP software solver on all instances of the hole suite from the DIMACS benchmark set. Speedups reportedly ranged from 0.289 to 111.245 (considering all three templates). Since it is difficult to correctly ascertain the efficiency of the solver on only one type of problem, other DIMACS instances were used to compare it to GRASP but no significant accelerations were observed. One limiting aspect of the approach is the communication between the software and hardware. As problems become larger and more difficult, more simplifications, reductions, and thus more downloads to the FPGA board are required, which may lead to a decrease in performance.

A slightly different SAT solver flavor is presented by Safar et al. [7], [6] as their solution is implemented completely in hardware and avoids instance specificity by storing SAT instance information in RAM modules. In [7], the presented approach is based on performing a depth first search which is paired up with non-chronological, conflict directed backtracking (Figure 18). In addition, the proposed methodology distinguishes itself by the method employed for clause evaluation as a shift register is used to encode when clauses are satisfied (right shift), unsatisfied (left shift) or there is no impact by the current variable assignment. The authors restrict the range of acceptable problems to 3-SAT and use the "0001000" vector in the shift register-based clause evaluator. A clause thus has two chances to be satisfied and if a '1' is present in the left-most bit, a conflict has been detected. For non-chronological backtracking purposes, a priority encoder is used to determine the return level. Variables are ordered statically and the maximal problem size that can by analyzed by this solver is 100 variables and 200 clauses. The circuit was able to runs at a maximal frequency of 65 MHz and occupies 85% of the XC2VP4 FPGA used.

Building on notions from their previously mentioned work, Safar et al. enhance their design [6] with a five stage pipeline (Figure 19). The first, called the variable decision stage (VD), is in charge of the overall control of the solver, performs static variable decision (there

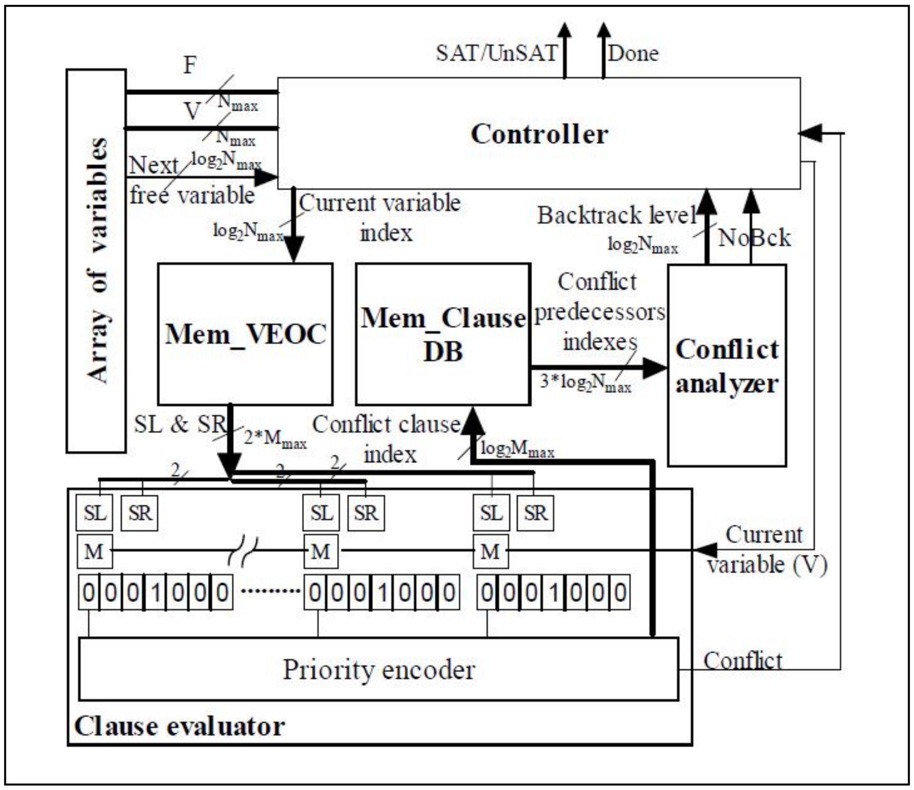


Figure 18: Safar et al. SAT solver architecture [7]

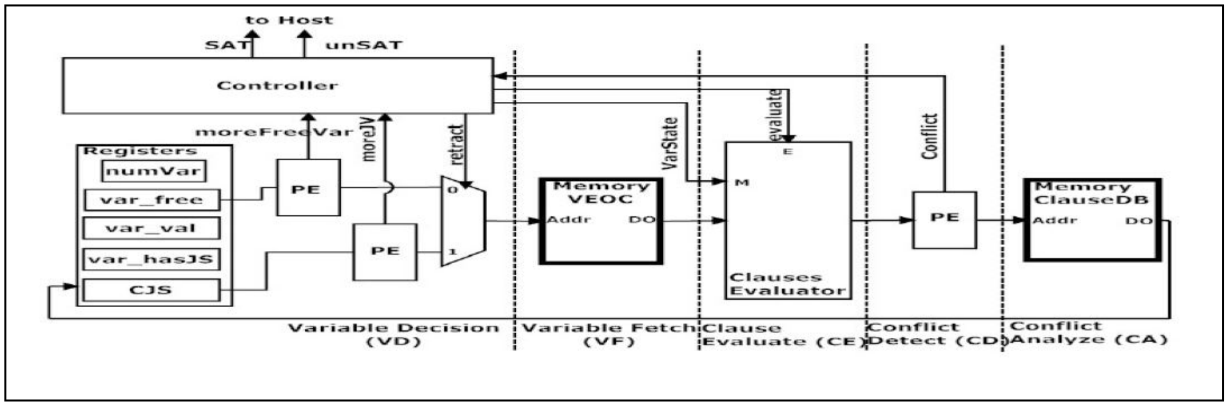


Figure 19: Safar et al. 5-stage pipelined solver [6]

is no implication computation) as well as conflict analysis. The second stage, termed the variable effect fetch stage (VF), uses a memory module to keep track of variable assignment effects on clauses. The principle behind clause evaluation from [7] is reused in the clause evaluate stage (CE) where 7-bit shift registers (one for each clause) are used to evaluate all clauses in parallel. The fourth stage (CD) deals with conflict detection (as before, a '1' in the left-most bit position of a clause evaluator register indicates a conflict) whereas the fifth and final stage (CA) analyzes conflicts. There are several advanced techniques employed in this solver. These include non-chronological backjumping, dynamic backtracking and learning without explicit implication graph traversal. As in their previous work [7], the authors use RAM modules to store SAT problem information thereby avoiding instance specificity. The maximal size of any one problem that can be accommodated by this solver is 511 variables and 511 clauses. This solver circuit operates at 120 MHz and occupies 82% of available LUTs in the XC2VP30-FF896 FPGA used as well as 47% of all available on-chip RAM. The authors report on a comparison made with the SATzilla2009_C software solver which revealed speedups and decelerations between 0.31 and 8.81.

Another example of partitioning SAT problems into smaller instances is provided by Gulati et al. in their work [11] based on their previous custom application-specific integrated circuit (ASIC) implementation [45]. The general architecture and the FSM of the decision unit are presented in Figure 20. The solver proposed here traverses the implication graph as well as generates conflict clauses in hardware in parallel. The BCP methods and non-chronological backtracking of the GRASP software SAT solver are implemented in hardware. The selection of decision variables is done statically and the number of variables and clauses (as well as clause width) is fixed.

Before solving a SAT problem, a preprocessing step is needed to heuristically partition it into instances that can fit onto the FPGA. For this purpose a 2-dimensional graph bandwidth minimization algorithm with greedy bin-packing is used. In order to find a solution to a problem, all sub-instance bins must be satisfied. Initially, all problem sub-instances are stored in off-chip RAM and are subsequently loaded into on-chip memory by a PowerPC core using the on-chip peripheral bus (OPB) and processor local bus (PLB) protocols from Xilinx. The authors use a XC2VP30 FPGA to implement their solution, which occupies about 70% of logic resources, and the maximal problem size is 8K variables and 14K clauses. Of note is the

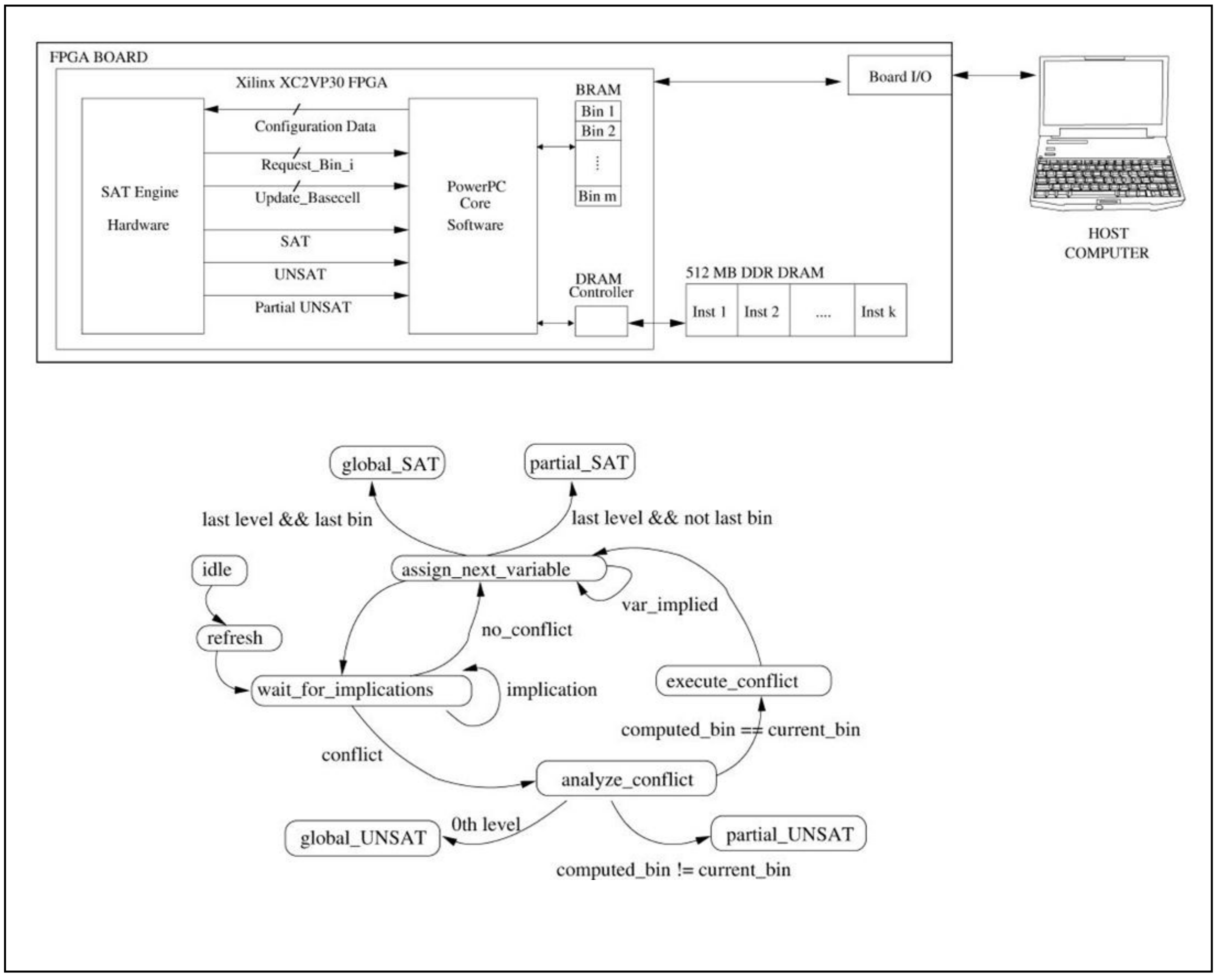


Figure 20: Gulati et al. solver architecture (top) and decision engine FSM (bottom) [11]

mathematical model developed by the authors to project their results to the larger XC4VFX140 FPGA. On this platform, the authors extrapolate their solver to accommodate 10K variables and 280K clauses. Moreover, comparisons of XC4VFX140 projected runtimes on various SAT instances with MiniSAT, a successful software solver, yielded a speedup of about 90.

Similar to the above-mentioned Leong et al. approach, Kanazawa and Maruyama's solver (Figure 21 shows a high level view) [10] explores the use of an incomplete SAT algorithm in solving very large 3-SAT problem instances on FPGA using a variant of the WSAT stochastic local search algorithm. The authors justify their selection of this incomplete SAT method with the fact that the resulting circuit does not need complex control structures as

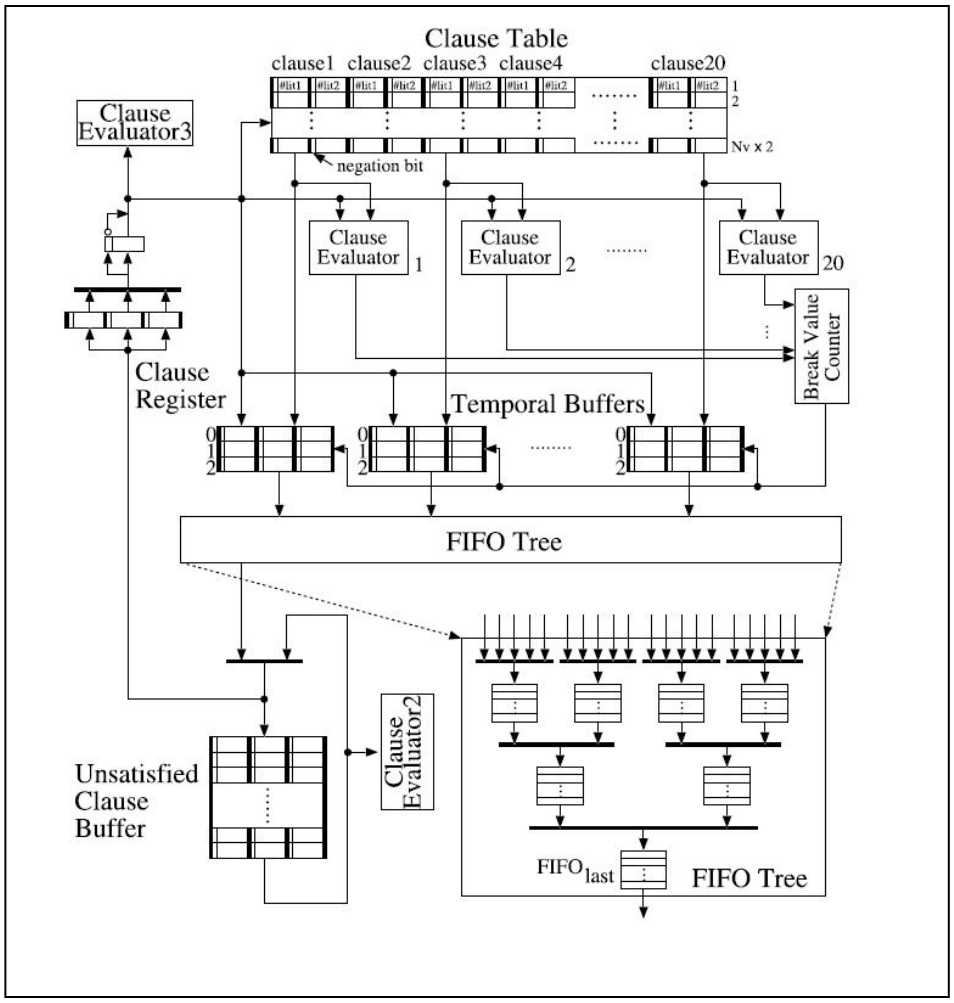


Figure 21: Kanazawa and Maruyama WSAT based solver [10]

well as with the idea that WSAT has very good inherent parallelism that can be used. In order to reduce circuit size, Kanazawa and Maruyama's solver evaluates, in parallel, only clauses that have the possibility of being unsatisfied by the flipping of a variable. Furthermore, this approach makes use of multi-threaded execution as a way to increase performance as many independent tries are executed in parallel. On-chip memory is heavily used for different tables and buffers as well as for storing variables which can be flipped in 1 clock cycle. An important aspect of this use of memories is the fact that the size of both on and off-chip memories as well as off-chip memory bandwidth affect performance. The authors present two implementations. The first does not use off-chip memory as the problem is completely handled by the FPGA and its memory resources. This circuit has a maximal frequency of 85.2 MHz, claims 51% of the XCV6000 FPGA slices and 90% of on-chip RAM. The maximal problem size is 2048 variables with the number of variables depending on the instance analyzed (the authors have verified up to 8500 clauses). A comparison was made with the Walksat[5] software solver which revealed speedups between 3.4 and 50.8 on problems from the SATLIB benchmark suites[6]. The second implementation makes use of 8 off-chip memory banks, runs at 67.2 MHz, and requires 88% of the same FPGA's slices and 97% of its on-chip memory blocks. The largest problem accommodated by this circuit has 32K variables and 128K clauses. A second comparison made with the same Walksat solver revealed accelerations between 13.8 and 37.0 on problems from the SAT Live[7] benchmark suite.

3.5 Summary and Analysis of Hardware SAT Solvers

To provide a compact view of all works presented heretofore, Table 1 lists characteristics and attributes of the hardware solvers presented in this chapter. A large hurdle that a hardware-based SAT solver has to face is the potentially enormous instance representation as some problems have thousands to millions of clauses and variables. Even though FPGA capacities can, up to a certain extent, accommodate these problems, there is still a barrier that hinders the use of these powerful devices to solving SAT. Indeed, as can be seen in Table 1, the approaches presented in this chapter have mostly focused on trying to fit problems on their

[5] Solver site: http://www.cs.rochester.edu/u/kautz/walksat/
[6] Site: http://www.satlib.org/
[7] Site: http://www.satlive.org/

FPGA platform in a timely and efficient manner rather than on enhancing solvers with sophisticated heuristics and techniques, as is done for their software counterparts [34]. In addition, with larger problem sizes come immense, complicated circuits that require long times to generate. Since the principal goal of hardware SAT solvers is to provide accelerations over software solutions, long circuit build times significantly decrease or even nullify the raw speedup generated by the hardware. Thus, from instance specificity, a seemingly natural way to map SAT to hardware harvesting great amounts of FPGA parallelism, SAT solvers started evolving towards pipelined, sometimes software aided, problem-partitioning natures that rely on memory modules to store instance clauses. By doing so, solver authors have avoided having to bulk up hardware with clause-variable relationships. A tradeoff was thence made between the parallelism obtained from instance-specific circuits requiring long times for compilation and the more coarse parallelism provided by pipelined application-specific circuits with no compilation overhead. On the other hand, it is our intuition that instance specificity is a necessary avenue for providing fine grained FPGA parallelism that is required to tackle the NP-hardness of SAT. Application-specific solvers use memory modules where problem clauses are stored. Nevertheless, different problems arise from the use for memory such as latencies and other limitations (e.g. can only access a fixed width word at a time). Solvers built in this manner have, in our opinion, generally mitigated these barriers by using heuristics such as dynamic variable ordering and non-chronological backtracking. However, for real-world applications with immense problem sizes, memory transactions are unacceptable. Consequently, the rest of this work will focus on assessing the influence of memory on hardware SAT solving and on highlighting the fact that instance specificity parallelism is required to solve the Boolean satisfiability problem. To this effect, a testing platform was implemented that simulates an on-chip memory with as many read ports as there are clauses in the problem instance. The fundamental unit that was used in building this memory is the 1-bit register (grouped in arrays) residing in each logic element of the Altera DE2-70 FPGA board. As such, a focus of this investigation was to develop 2 versions of a SAT solver model based on the complete DP method that has access to all problem clauses at once. One version is a software simulator that is able to provide accurate clock cycle counts and the second is a VHDL description of the hardware necessary to implement the model on an FPGA board.

Table 1: Summary of hardware SAT solvers

Solver	Year	Specificity	Algorithm	HW/SW Execution	Maximal Problem Size (approx.)	Comments
Svoboda [3]	1968	Application	Exhaustive search	-All execution in HW.	N/A	-Highly parallel design: one register for each minterm. -Explosion of time and memory needed due to exhaustive search.
Suyama [9, 17-19]	1996-2001	Instance	DP-based with MOM and EUP dynamic variable selection	-All execution in HW.	~200 variables, ~300 clauses	-Use of registers for backtracking. -Large amount of resources due to wirring requirements.
Zhong [15, 20, 21]	1998-2000	Instance	DP-based with non-chronological backtracking and dynamic clause addition	-All execution in HW.	~200 variables, ~1000 clauses	-Only a subset of total problem clauses evaluated at any one time -FSM and Ring architectures -Incremental synthesis decreases compilation time
Platzner [8, 24]	1998-1999	Instance	DP-based	-All execution in HW.	~100 variables ~500 clauses	-High circuit frequencies (27 – 65 MHz). -FSM architecture.
Abramovici [2, 22]	1997-2000	Instance	PODEM with pure literal rule	-All execution in HW.	~10 variables ~30 clauses	-Dynamic pure literal identification. -Hardware intensive. -High parallelism: e.g. several variables can be assigned concurrently. -Partitioning scheme proposed

Table 1 cont'd : Summary of hardware solvers

Solver	Year	Specificity	Algorithm	HW/SW Execution	Maximal Problem Size (approx.)	Comments
Dandalis [14, 23]	2000-2002	Application	DP-based	-Implication computation in HW. -Variable decisions and backtracking in SW.	N/A	-Dynamic learinng to reconfigure FPGA. -Focuses on minimizing time to resolve implications or raise conflicts -Pipeline processes groups of clauses.
Leong [4, 16]	1999-2001	Application	GSAT, WSAT	-Inner algorithm loop in HW. -Outer algorithm loop in SW	50 variables, 170 clauses	-Problem specific FPGA runtime configuration -Circuit speed moderately high(33 MHz)
de Sousa [5, 25]	2001-2002	Application	DP-based	-Implication computation and selecting decision variable in HW. -Conflict analysis, backtracking, clause addition in SW.	~7000 variables[a], ~200000 clauses[a]	-Use context switching for instances that do not fit onto FPGA. -Use of clause pipeline to evaluate clauses.
Skliarova and Ferrari [13]	2004	Application	DP-based	-SW splits and reduces problem until can fit onto HW.	~100 variables, ~850 clauses	-Communication with host processor is not negligeable.
Safar [6, 7]	2007-2011	Application	DP-based with dynamic backtracking and clause learning; static variable selection.	-All execution in HW	511 variables, 511 clauses	-Store information in memory. -5-stage pipeline. - No implication computation.

[a] Interface with software was not implemented and no real execution times were available. Circuit was synthesized for XCV2000E board.

Table 1 cont'd: Summary of hardware solvers

Solver	Year	Specificity	Algorithm	HW/SW Execution	Maximal Problem Size (approx.)	Comments
Gulati [11]	2009	Application	Based on GRASP software solver	-SW partitions problem. -HW performs non-chronological backtracking	~8000 variables[b], ~14000 clauses[b]	-Heavy use of memory. -Use of general purpose PC and busses to transfer clause information.
Kanazawa and Maruyama [10]	2010	Application	WSAT	-All execution in HW	~32000 variables, ~128000 clauses	-Heavy use of memory. -Memory bandwithd limitations. -High circuit frequencies (85.2 MHz and 67.2 MHz).

[b] Authors developed a mathematical model that provides projections toward a larger FPGA where the maximal problem size is about 10000 variables and 280000 clauses. No actual problems of this size were tested. From projected runtime comparisons with MiniSAT, the largest problem had 3301 variables and 10092 clauses.

Chapter 4 – The Evaluation Platform

This chapter will describe in detail the platform that was built. Before proceeding with details regarding the functioning of the DP-like solver, consider the clauses γ_i (this time expressed in DNF) of the 5-clause, 7-variable formula $\overline{\varphi}$[8] that are shown in Figure 22. The CNF form of its inverse φ was introduced in Chapter 2.

$$\left\{\begin{array}{lcl} \gamma_0 & = & (\overline{x_0} \wedge \overline{x_1} \wedge \overline{x_4} \wedge \overline{x_3}) \\ \gamma_1 & = & (\overline{x_0} \wedge \overline{x_1} \wedge x_4 \wedge \overline{x_3}) \\ \gamma_2 & = & (\overline{x_2} \wedge \overline{x_5}) \\ \gamma_3 & = & (x_3 \wedge \overline{x_6} \wedge \overline{x_0}) \\ \gamma_4 & = & (x_3 \wedge x_6 \wedge \overline{x_1}) \end{array}\right\}$$

$$\overline{\varphi}(x_0, x_1, x_2, x_3, x_4, x_5, x_6) = \gamma_0 \vee \gamma_1 \vee \gamma_2 \vee \gamma_3 \vee \gamma_4$$

Figure 22: DNF of formula $\overline{\varphi}$

As an initial step in our exploration, we attempted an implementation of Svoboda's Boolean analyzer [3]. Since this solver makes use of the DNF, it was deemed natural to continue with this notation when considering SAT which is a simpler but nearly identical problem (Svoboda attempts to find all solutions to a Boolean formula while SAT attempts to find one such solution). This form is easily obtained by applying De Morgan's law of duality[9]. From a SAT standpoint, finding a solution to the Boolean formula φ starting with its negation $\overline{\varphi}$ now involves finding a variable assignment that renders the entire formula false. In other words, the satisfying assignment must have at least one disjunction point (one variable assigned differently) with all clauses belonging to formula $\overline{\varphi}$. The significance of this disjunction is that if a solution is not included in the set of all solutions to the inverse formula $\overline{\varphi}$ then surely it must be a solution of φ.

[8] Example borrowed and adapted from: http://www.cs.cmu.edu/~mtschant/15414-f07/lectures/grasp-ex.pdf
[9] http://en.wikipedia.org/wiki/De_Morgan's_laws

4.1 Introducing a DP-Based Solver

Figure 23 below shows a high-level view of the DP-like solver that was constructed. The components shaded in grey represent a generic decision machine that uses information from registers to analyze a Boolean formula in the attempt to finding a solution. The portions of Figure 23 not shaded in grey represent the different memories used by the solver. The nature of the solver may be therefore thought of as a hybrid between instance and application specificity. Indeed, the model is in part instance-specific as the size of the clause register array (both the number of registers and also their size in the number of bits), the current partial solution register (Candidate in Figure 23) as well as the size of the stack memory need to be specified at compile time. The application-specific flavor of the solver comes from the fact

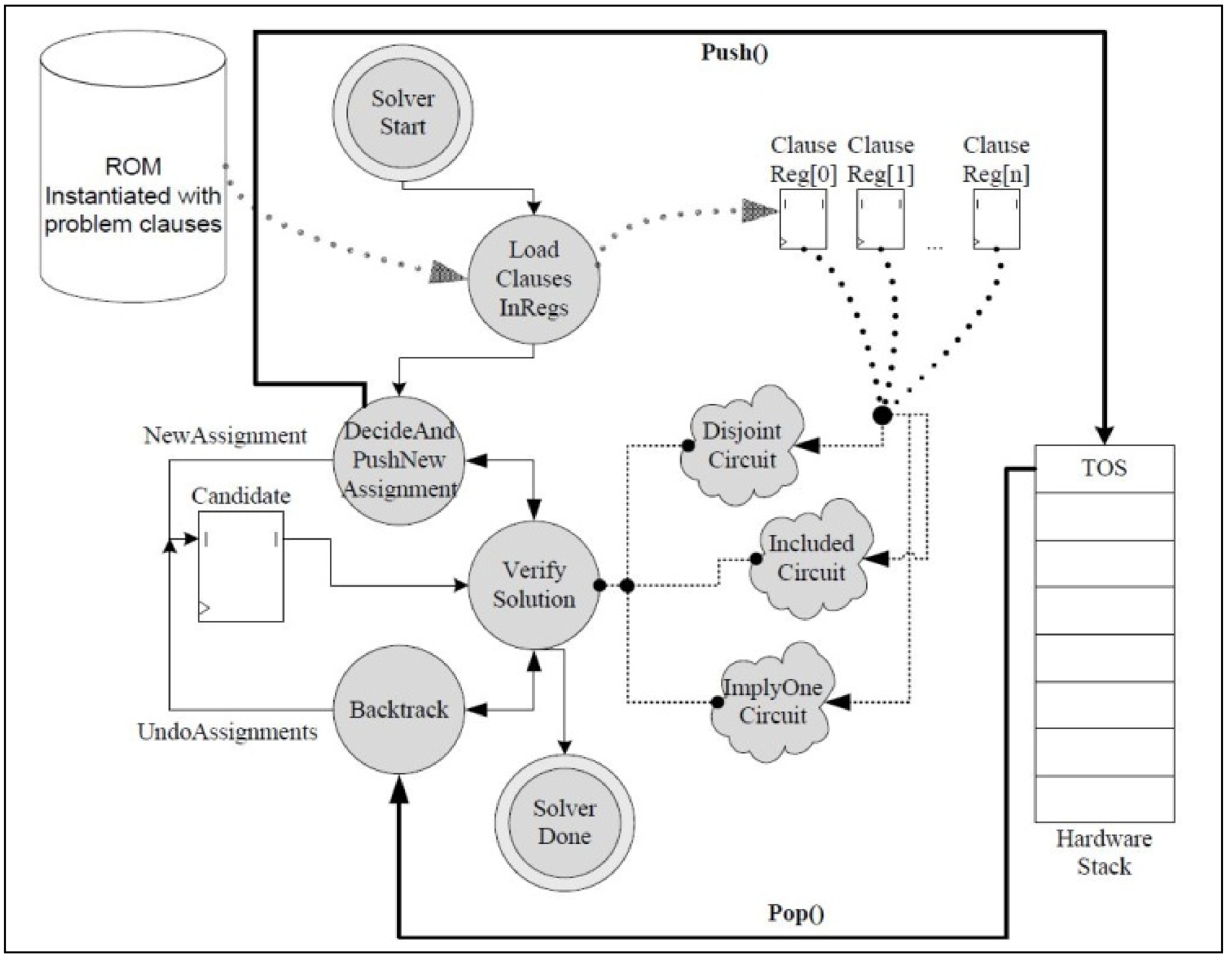


Figure 23: High-level model of a DP-based SAT solver

that it uses a memory, a read-only memory (ROM) in this case, to hold problem information (initially only). With some adjustments, the solver can bend towards application specificity. For example, adding a few extra configuration registers would allow for programming the size of the problem. The only component then needing reconfiguration would be the ROM. This would, however, enforce a theoretical maximal limit (regarding problem size) on the solver but recompilation would only be needed when the problem size would exceed this limit.

4.2 Encoding a SAT Problem Instance

The first step that the solver performs before solving the SAT instance is transferring SAT clauses from the ROM to an array of clause registers. The ROM is generated automatically (not shown) by a Java program from a CNF file specification[10]. A variable may be in any of 3 states (therefore needing only 2 encoding bits): "10" or true, "01" meaning false and "11" or unassigned; the state "00" is illegal and is treated as a don't care value. Each ROM address represents a clause index and the width of the memory word is thus equal to the number of variables of the problem at hand. The ROM module can be thought of as storing 2 dimensional quantities as at each address a word is as long as there are variables in the problem and each variable state is encoded with two bits. The size in bits of the ROM needed for problem representation is therefore twice the number of clauses multiplied by the number of variables. The Candidate register is also 2 bits wide and uses the same encoding scheme to represent variable states. An index into this register represents a variable index. The stack memory used in backtracking is as deep as there are variables in a problem and as wide as $\log_2 \mathrm{NumVars}$ where NumVars is the number of variables of the problem. Indeed, a secondary job of the Java program is to determine, from the CNF file, the sizes needed for the different registers and memories previously described. Finally, the register array (ClauseReg in Figure 23) also uses the state encoding scheme mentioned above and holds the information transferred from the ROM module. This register array is used to simulate the memory whose words are all accessible at once and whose latency is only of 1 clock cycle. Of course, a ressource penalty must be paid as each register is located inside a logic element (LE) of the DE2-70 FPGA.

[10] For CNF file specification see: http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

4.3 Disjunction, Inclusion and Implication – Solving SAT

In Figure 23, the logic-cloud shapes represent combinational circuits that, within 1 clock cycle, are responsible for computing various results that are used to solve SAT. To obtain these circuits, the following three Boolean formulas were used:

$$\mathrm{Disjoint} = \prod_{i=0}^{\Gamma-1}\sum_{j=0}^{N-1} \overline{\alpha_1[j]} * \overline{\Upsilon_0[i][j]} + \overline{\alpha_0[j]} * \overline{\Upsilon_1[i][j]}$$

$$\mathrm{Included} = \sum_{i=0}^{\Gamma-1}\prod_{j=0}^{N-1} \overline{\alpha_0[j]} * \overline{\Upsilon_0[i][j]} + \overline{\alpha_1[j]} * \overline{\Upsilon_1[i][j]}$$

$$\mathrm{ImplyOne} = \sum_{i=0}^{\Gamma-1}\sum_{j=0}^{N-1} \mathrm{Unit}(\gamma_i) * \overline{\mathrm{Disjoint}(\gamma_i)} * \overline{\Upsilon_1[i][j] * \Upsilon_0[i][j]} * \alpha_1[j] * \alpha_0[j]$$

The $'*'$ and $'+'$ operators implement the usual Boolean logical-and and logical-or operations. Symbols Γ, N and represent the total number of clauses and the total number of variables, respectively. The $\alpha\_$ symbol is the appropriate dimension of the current variable assignments vector (Candidate solution), $\Upsilon\_$ denotes one of the dimensions of the clause register array Υ while γ_i means the ith clause belonging to it. For example, $\Upsilon_1[0][2]$ stands for the 1st state bit of clause 0, variable 2 whereas $\alpha_0[8]$ can be taken to mean the 0th state bit of variable 9. The functions $\mathrm{Unit}(\gamma_i)$ and $\mathrm{Disjoint}(\gamma_i)$ are implemented as auxiliary circuits that are used to find out if a particular clause is a unit clause, meaning it has only one unassigned variable, and if this same clause has at least one variable that creates a disjunction point from the current assignment; these are necessary conditions for implication computation.

The Disjoint formula describes the circuit used to determine if a solution has been found to the SAT instance as it calculates the disjunction of the candidate solution from all problem clauses. The formula examines, for a given clause, the assignment value of each variable. If at least one (logical-summation Σ) variable value has been found to be of different polarity from the current solution (assigned "10" in the clause and "01" in the Candidate), then the clause is disjoint. The logical-product Π large operator dictates that all clauses must be disjoint in order for the formula result to be equal to '1'.

The Included formula calculates whether a given partial (or full) solution is included in at least one of the clauses of (and therefore provides a solution to) the Boolean formula $\overline{\varphi}$. If such a situation occurs, the algorithm backtracks as a contradiction has occurred since a solution to $\overline{\varphi}$ is necessarily not a solution to φ. This second formula also examines all clauses of the problem. This time, logical-product Π operator is used inside the logical-summation Σ operator since for inclusion it is necessary that all literals inside at least one clause be of same value as the variable assignments of the Candidate (there is no point of disjunction, or the corresponding clause of φ is false). For example, looking at the example presented earlier, the partial solution ("11")("11")("10")("01")("01")("01")("01") (in other words "--10000") is included in clause $\gamma_1 = (\overline{x_0} \wedge \overline{x_1} \wedge x_4 \wedge \overline{x_3})$ and leads to a contradiction because $\overline{\varphi}$ is satisfied and φ is not.

Finally, the ImplyOne formula is at the origin of the implications' finding circuit. Four conditions must be met to allow a variable to be implied to a value in the Candidate. Firstly, the variable must be specified in a given clause. This is determined by the $\overline{\Upsilon_1[i][j] * \Upsilon_0[i][j]}$ portion of the formula. Secondly, the examined clause must necessarily have only one unassigned variable as well as no other disjunction points from the current assignment vector. These computations are performed by $\mathrm{Unit}(\gamma_i)$ and $\mathrm{Disjoint}(\gamma_i)$ as already discussed. Lastly, it is necessary to check that the variable has not been assigned in the current Candidate solution, a verification which is done by the $\alpha_1[j] * \alpha_0[j]$ portion of the formula. If all these conditions are met, the variable is simply assigned the opposite value of that which it has in the clause.

To illustrate how these formulas are used consider the following (partial) Candidate solution (to the formula presented at the beginning of this chapter) ("11")("11")("10")("01")("01")("01")("01"), remembering that the left-most bracket contains the current assignment for variable 6 and the right-most for variable 0. The Disjoint circuit starts with clause $\gamma_0 = (\overline{x_0} \wedge \overline{x_1} \wedge \overline{x_4} \wedge \overline{x_3})$ and looks at every variable position in the clause. Variable 0 is represented by its negative literal. As the Candidate also has assigned false to variable 0 (the right-most bracket contains "01"), no disjunction point is found here. Next in line is variable 1 where the same situation repeats itself. When x_2 is examined, the solver determines that this variable is not present in the clause and thus does not affect the outcome of the Disjoint operation. When x_3 is considered, the Candidate features "01" as the variable is assigned false which prevents this variable from forming a disjunction point from the clause. As x_4 is analyzed, a disjunction point is found as the variable is assigned false in the clause but not in the Candidate. At this point, the first clause has been determined to be disjoint, and thus unsatisfied. The solver will continue on to clause γ_1. If all clauses have at least one disjunction point, a solution has been found.

Examining the same partial assignment, the Included circuit is tasked with determining if any contradictions have occurred. This situation happens when all the variable assignments in the Candidate have the same polarity as they do in the clause. As Disjoint had done, clauses are examined in turn and their variables are analyzed one by one. It is possible to see that all variables present in clause γ_0 are assigned as they are in the Candidate with the exception of x_4, which is true in the partial solution but false in the clause. Therefore, the Candidate is not included in this clause. However, the contradiction arises with the next clause $\gamma_1 = (\overline{x_0} \wedge \overline{x_1} \wedge x_4 \wedge \overline{x_3})$ as all variables feature the same polarity in both the Candidate and the clause.

Finally, to illustrate the ImplyOne circuit, consider the following partial assignment: ("11")("11")("11")("01")("01")("01")("01") which is almost identical to the one used for the first two circuits with the exception that x_4 is free. The ImplyOne formula dictates that we look at each variable present in every clause. Starting with γ_0, the formula asks if the clause is a unit clause, which is the case as variables 0,1 and 3 are all assigned in the Candidate. The next step involves determining if the clause has a disjunction point with the Candidate, which it does not as x_0, x_1 and x_3 have identical polarity in both the clause and the Candidate register; the second portion of the formula is satisfied. Next, with $\overline{\Upsilon_1[i][j] * \Upsilon_0[i][j]}$ the formula stipulates that in the clause register array, the variable must be present (therefore cannot exist as "11" as this means the variable is free or absent; e.g. in clause γ_0 variable x_2 is free). This is the case for x_4. Finally, the last portion of the formula makes sure that the variable is actually not assigned in the Candidate. Indeed, x_4 is not assigned. In this case, an implication has arisen and x_4 is assigned true, the opposite of the polarity found in the clause (which is false in this case).

4.4 Solution Space Exploration

The method used in exploring the space of solutions to a Boolean function is based on the famous DP SAT algorithm. The circles of Figure 23 represent a FSM that implements a variant of this algorithm.

The solution search space is organized as a tree and the main goal is to find a solution that unsatisfies all clauses. At the root, no assignments have been made and all variables are said to be free. The Candidate displays "11" at all its positions to indicate this fact. In order to move forward, the solver uses a simple circuit (not shown here) that detects the next unassigned free variable by examining the Candidate vector in a sequential manner (as a sidebar, it is worth mentioning that several interesting heuristics used in selecting this variable do exist [13] and can be used to increase the performance of the solver, although this would come at the price of increased FPGA area that the solver would require). This variable is termed the decision variable as the solver decides it to be false and pushes its index onto the stack. Considering our earlier example, Figure 24 (a) shows the first three decision steps of the algorithm in which the solver decides to assign '0' to variables x_0, x_1 and x_2. The choice of '0' and is purely arbitrary. The indexes of these variables are pushed onto the stack in order. The incomplete solution that results from these decisions does not yet constitute an unsatisfying assignment (although this is possible as some variables can be deemed superfluous) but does not raise any contradictions either. Clauses γ_0 and γ_1 contain x_0 and x_1 and are not yet unsatisfied. However, clause γ_2 has only two literals, one of which is x_2. The decision on this variable has rendered γ_2 a unit clause. This is significant as it is no longer necessary to make a decision on x_5 later in the search process. All sub-trees resulting from decisions on x_5 do not need to be explored and the search space is pruned in a significant manner. Now, the decision to make x_2 '0' has as an implication the fact that x_5 must be assigned '1' so that γ_2 can be unsatisfied and therefore disjoint from the solution (green arrow in Figure 24 (a)).

As it was previously mentioned, it is possible for contradictions to arise. This situation occurs when the same variable is implied to different values by at least two unit clauses. In our

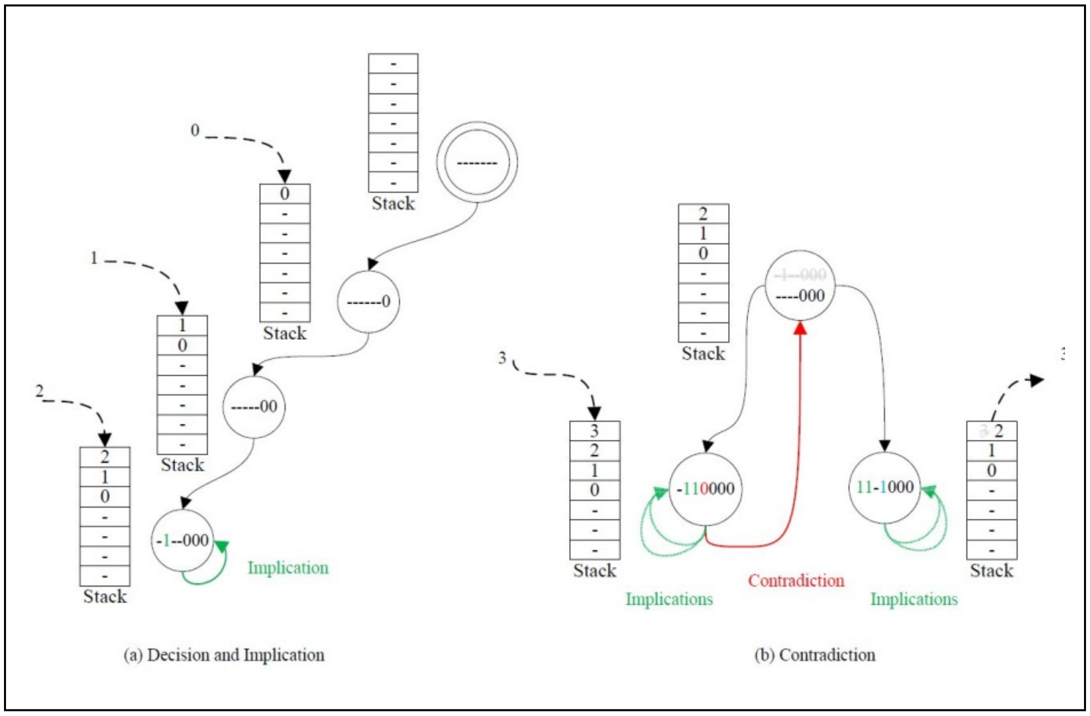


Figure 24: Operation of DP-like solver

current situation, after assigning '0' to the first three variables and implying the 6th one to '1' the solver proceeds (as the implication did not cause a contradiction but did not generate a solution either because, with the exception of γ_2 all clauses are still not disjoint from the assignment) by choosing the next available free variable, x_3, which, as was done before, is decided to be '0'. The solver checks for and detects 2 new unit clauses, γ_0 and γ_1. The only variable left to assign in both these clauses is x_4. Sequentially, the solver starts to check for implications by looking at γ_0 and implies the value of x_4 to '1' so that the clause is unsatisfied. However, during the next cycle, the current variable assignment of "-110000" renders γ_1 true and thus creates a contradiction. Figure 24 (b) shows the backtracking mechanism employed by the solver to correct this situation. Index 3 had been pushed onto the stack as its variable had been decided. However, since a contradiction was raised by this decision, this index is popped off the stack, all assignments starting with index 3 are reset and the value of its associated variable (x_3) is complemented. At this point, both possibilities have been tried for variable x_3. Variable x_5 is once again implied as before. In addition, γ_3 has emerged as new unit clause. This event allows the solver to set the variable x_6 to '1' so that the clause is

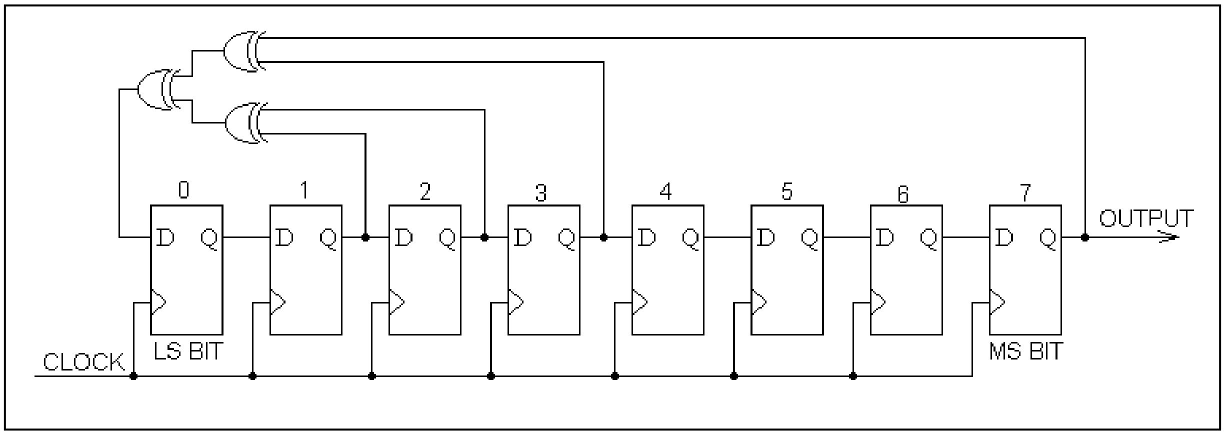


Figure 25: Example of an 8-bit LFSR

unsatisfied. This will again lead to a contradiction as the last clause of the problem is now true because x_1 evaluates to true as its assignment is '0' while x_3 and x_6 are also true as their assignments are '1'. After resolving all contradictions resulting from decisions and implications, the solver proceeds with new decisions. The search terminates when the circuit described by Disjoint evaluates to true or all values of all variables have been tried.

4.5 LFSR and Pseudo-randomness in Variable Decision

In the previous section, the manner in which variables get their value is described as arbitrarily starting with '0', an assignment which is inverted with subsequent conflicts. In addition to this predefined choice of always starting with false, it is also possible to configure the DP-like solver to use a linear feedback shift register (LFSR) to pseudo-randomly select a value during execution by selecting a bit from the LFSR output. Figure 25[11] shows an example of an 8-bit LFSR.

As the LFSR's seed is randomly generated at compile time (by the same Java program that generates the problem ROMs) it allows the introduction of almost randomness into solver operation so that it can take random paths to solutions. Thus, for a given seed and a given problem instance, the LFSR method of deciding on a variable value will always yield the same

[11] Figure taken from : http://www.markharvey.info/fpga/lfsr/lfsrfig4.gif

solution. However, if the circuit is recompiled with a different seed, it is possible (and highly likely) that a new solution will be found in a different time period.

4.6 Some Remarks

While this DP-based search algorithm does solve SAT instances in a manner that is significantly more efficient than, say, a brute-force approach where all solutions are enumerated and verified, there are many improvements that were suggested in the other solvers presented in this thesis that could be used to further accelerate its execution. Examples include dynamic variable ordering [9], [17], [18] and non-chronological backtracking as presented in [21] whereby conflicts are analyzed so that the solver backtracks not to the last decision variable (chronological backtracking) but to the level responsible for the contradiction. Moreover, implication generation is one of the most important aspects of SAT solving. As more and more decisions are made, a greater number of unit clauses appear and the solver is capable of verifying quickly if its decisions are correct. Thus, a measure of SAT solver efficiency may be thought of as the number of implications that it can raise in a small amount of time

These enhancements do come with a price in terms of the hardware resources needed to implement them. For example, the Disjoint circuit for the example used in this chapter (as compiled with the Quartus II version 12.0 CAD software, which was also used during the experiments portion of this work) requires 47 logic elements, the Included circuit needs 55 whereas the ImplyOne circuit claims 204 logic elements.

Chapter 5 – Experimental Results

The solver model presented in this work is very hardware intensive and as such does not offer a viable solution for a successful, efficient SAT solver from a practical point of view. Rather, the solver presented here forms an exploration platform that paves the way for similarly constituted solvers relying heavily on FPGA fine-grained parallelism for processing power. As such, the main goal of the experimental section of this work will be to assess exactly how quick a DP-based solver (relying on no heuristics or other enhancements except perhaps pseudo-random variable assignment and computing all implications in 1 clock cycle) is when compared to other solutions, be they hardware or software in nature, if memory limitations are eliminated by using a large, fast memory module capable of producing clause information in 1 clock cycle.

To simulate the multi-port memory (in order to have as many read ports as there are clauses in a problem instance) so as to benefit from extensive parallelism when evaluating instance clauses, registers located in each logic element of the DE2-70 Cyclone II FPGA (Figure 26[12]) used are necessary to hold state information bits for each variable present in each every clause. The Cyclone II DE2-70 FPGA board has 68416 logic elements (1 logic element ≈ 1 LUT, Figure 26). This number constitutes an upper limit on the size of the problem that can be considered via our approach. In addition, registers are also needed for the candidate solution, although the space that this component requires is minimal compared to problem clause information. Furthermore, some logic elements are used for computing tasks and cannot be used for the aforementioned purpose. Compilation techniques are employed by the Quartus II CAD tool to optimize the design in a number of ways. One such technique is called register packing[12] and can be used to reduce the total number of device resources used by utilizing the LUT and the register of a logic element for functions that are not related to one another. However, very small problems (from a real-world application size standpoint) claim large amounts of resources. For example, one problem of 30 variables and 99 clauses necessitated 25966 logic elements or about 38% of the total number of logic elements of the FPGA.

[12] Obtained from the Cyclone II device handbook.

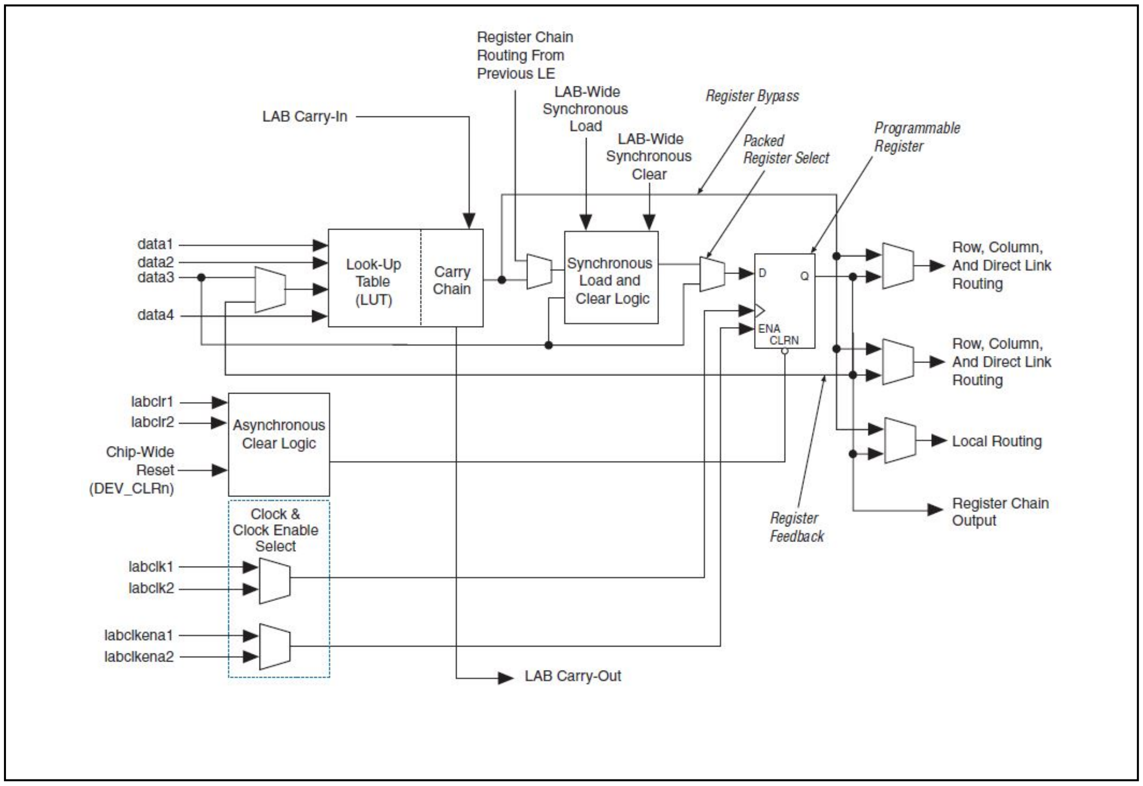


Figure 26: Cyclone II logic element

The heavy hardware requirements pose a significant barrier as not many problems can be implemented and run directly on FPGA. Thus it was decided that the software simulator version of our solver would be used to obtain clock cycle counts for the different problems examined in this work. As gate-level HDL simulators such as ModelSim perform simulations at a very low level evaluating all signals of an HDL specification, their runtimes can sometimes be exceedingly long (hours and even days of simulation time). Thus, a custom simulator written in the Java programming language was developed that is faithful in every way to the specification of the VHDL model of the DP-based solver. This simulator was then used to obtain clock cycle counts for problems analyzed by other solvers for comparison purposes. Furthermore, to estimate probable circuit frequencies regression curves were used as the VHDL version of the solver was synthesized for small problems randomly generated by the makewff utility that can be obtained with the Walksat software SAT solver[13].

[13] Solver site: http://www.cs.rochester.edu/u/kautz/walksat/

5.1 Regression Analysis – Rules of Thumb

The problems used in the regression analysis to approximate various circuit parameters are shown in Table 2. Note that what is sought here are not exact relationships between various parameters but rather rules of thumb that can be used to approximate and characterize the suitability of using FPGAs in solving SAT. In total, 20 random 3-SAT problems were synthesized and their circuit build times, maximal circuit frequencies and required amount of memory bits were used in constructing graphs illustrating relationships between these metrics and an instance's size.

Indeed, researchers have tried for many decades to develop sophisticated algorithms to solve hard optimization problems [46]. Experiments were carried out to see which algorithms

Table 2: Characteristics of circuits used in the regression analysis

Variable Number	Clause Number	Build Time(sec)	Maximal Frequency	Memory Bits
10	33	74	62.1	1280
11	36	75	59.41	1408
12	39	94	55.61	1536
13	43	104	50.7	1664
14	46	112	43.19	1792
15	49	125	39.3	1920
16	53	138	38.02	2048
17	56	167	42.34	2261
18	59	174	33.79	2394
19	63	204	41.4	2527
20	66	226	31.31	5220
21	69	262	39.34	5481
22	72	262	35.8	5742
23	76	297	36.97	6003
24	79	303	27.19	6264
25	82	368	32.92	6525
26	86	398	33.12	6786
27	89	450	32.68	7047
28	92	476	30.7	7308
29	95	492	30.2	7569
30	99	577	29.1	7830

perform best on benchmark instances that are publicly available but provided no useful conclusions. In addition, the no-free-lunch theorem [47] states that there is no algorithm that can indicate which algorithm would perform better than all other algorithms on all instances of a given problem [46]. It is therefore difficult to establish exactly what metric (e.g. problem size, clauses-to-variables ratios, number of literals present in one polarity or another, etc.) best characterizes a SAT instance as it is solved by the DP algorithm. Various combinations of these metrics could be used to create easier or harder problems. However, the approach of exploring various metric combinations is neither useful nor particularly feasible. Instead, inspiration was drawn from a famous result by Selman et al. [48], which states that a clauses-to-variables ratio of 4.26 in random 3-SAT instances makes for very difficult problems. Consequently, the 3.29 average clauses-to-variables ratio of all problems used in comparing our solver to others was used to generate random 3-SAT problems using the makewff utility. In this manner, some of the SAT instances' structure is reflected in the smaller problems used in the regression analysis.

Before proceeding with the description of the data obtained it is worthwhile to note that the general purpose PC used in this work features an octa-core Intel i7 processor running at 2.80 GHz with 8 GB of RAM. The operating system used was the 64-bit version of the Ubuntu operating system (12.04 LTS).

The first graph generated is shown in Figure 27 and it depicts the size of a circuit expressed as the number of logic elements claimed on the Cyclone II FPGA as influenced by the number of clauses present in the SAT instance. The curve that fits the experimental data is best expressed by power type regression. The formula representing the rule of thumb relating logic elements to the number of clauses present in a SAT instance was in this way approximately found to be:

$$\text{logic\_elements} = 2.8 * \text{num\_clauses}^2$$

It is now possible to infer that the maximal problem size that can be accommodated by our FPGA features about 156 clauses and, since we have assumed 3-SAT, 52 variables. Of course, this limit is imposed by the capacity of the FPGA board used. However, using larger

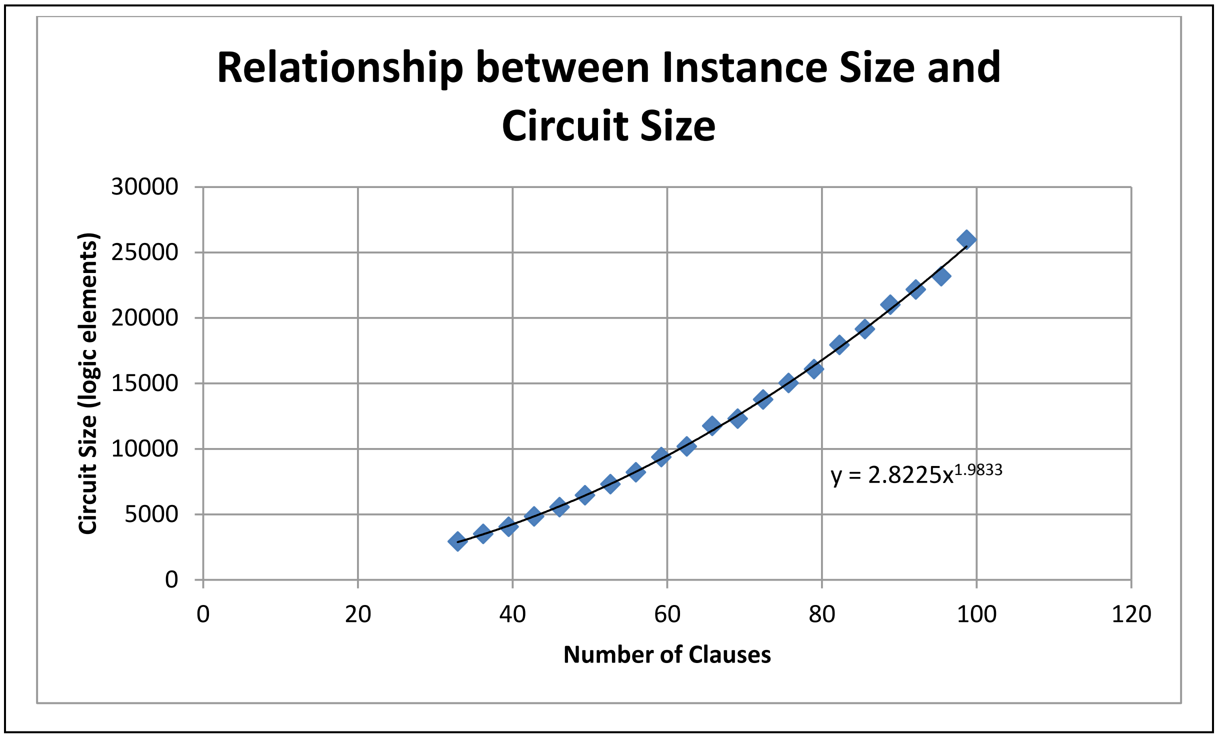


Figure 27: Relationship between an instance's size and its associated circuit size

FPGAs would not allow the solver to process significantly bigger problems. Solving this area issue constitutes one of our most important research priorities.

Even though the circuits generated by our approach are large, the relationship between the time to build a circuit and instance size is also best expressed by a power regression curve (as opposed to an exponential one). The graph showing this is displayed in Figure 28 and the relationship between the circuit build time and the size of the problem as expressed by the number of clauses is given below:

$$\text{circuit\_build\_time} = 0.08 * \text{num\_clauses}^2$$

Thus, once again, the problem that occupies the entire FPGA (whose parameters are given higher) is solved by a circuit that would require approximately 1950 seconds (about 32 minutes) of build time.

The most important approximation that was performed concerns circuit frequency as dictated by problem size. This parameter is employed later after the simulator version of our solver is used to obtain clock cycle counts for problems that do not fit onto our FPGA. Figure 29 shows the graph of circuit frequency versus circuit size. The relationship expressing the link between circuit frequency and the number of clauses of a problem is given below:

$$\text{frequency} = 552/\sqrt[3]{(\text{num\_clauses})^2}$$

The minimal circuit frequency featured by the largest problem fitting onto the DE2-70 Cyclone II FPGA is thus considered to be about 19 MHz. This result differs somewhat from those found in other works presented in this thesis although the problem sizes considered are small. However, a circuit spanning the entire FPGA is still able to function at a relatively high frequency.

Lastly, to get an idea of the amount of the amount of memory bits required by the ROM module to store the problem clauses, a graph illustrating the relationship between this quantity and the number of clauses of an instance is given in Figure 30. The stair-like effect on the graph is attributed to the fact that memory address width is always a power of 2. For example, the point on the graph just before the inflection represents a problem of 19 variables

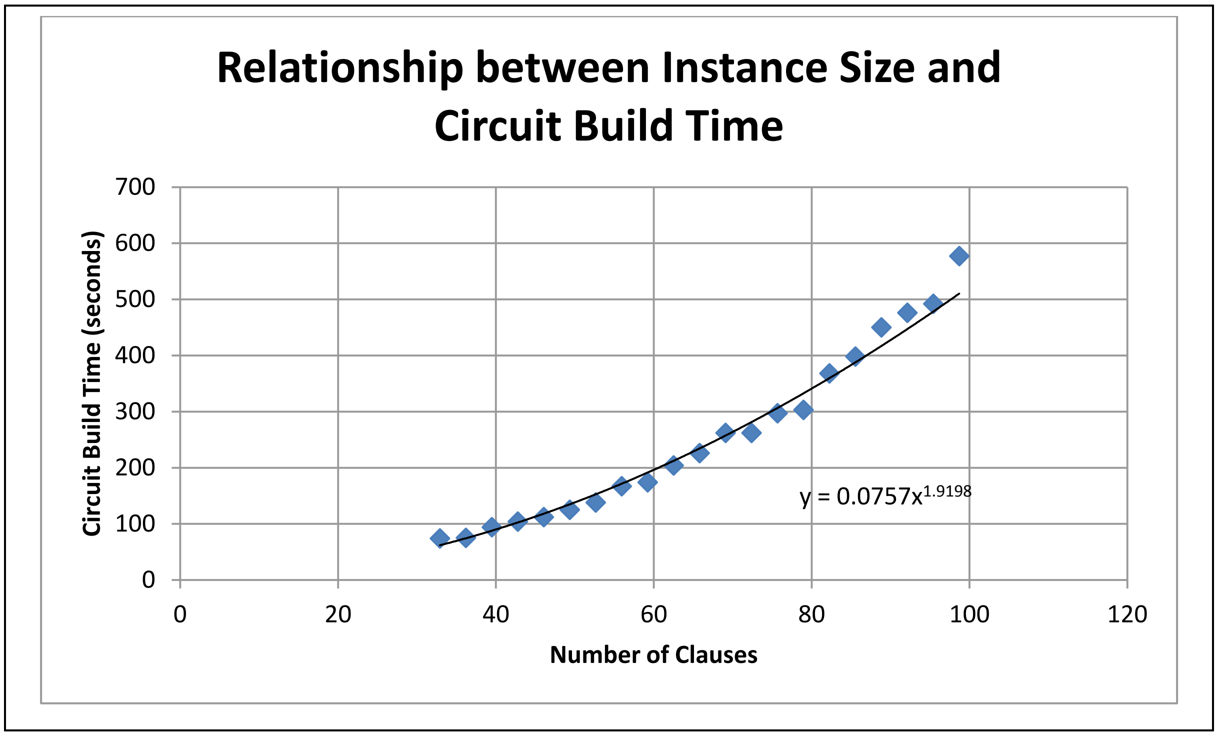


Figure 28: Relationship between an instance's size and its associated circuit build time

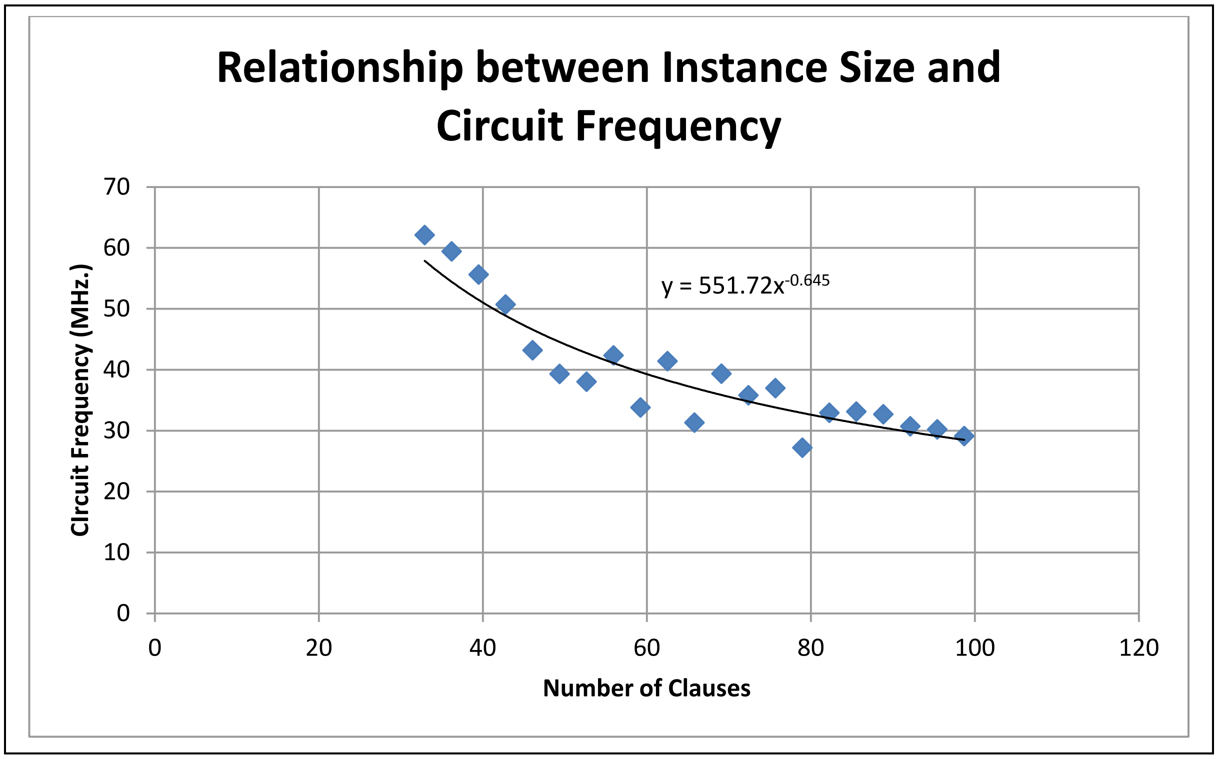


Figure 29: Relationship between an instance's size and its associated circuit frequency

and 63 clauses. As mentioned previously, in each clause we need 2 bits to represent variable states. As we have a word width of 19 (for the total number of variables), the number of memory bits required is 2^6*19 *2 = 2432 bits (where 2^6 = 64 are the total number of words in the memory; this is computed from CEILING (lg_2(num_clauses)) = 6). There is a discrepancy with the value reported in Table 2 due to the fact that the stack memory is also taken into account (19 variables would require 19 * 5 = 95 bits which brings the total up to the reported 2527 bits). On the other hand, if the stack memory is too small, as is the case for the first problems, the compiler selects logic elements to implement it rather than dedicated memory bits. Now, the next point on the line represents a problem of 20 variables and 66 clauses. The next power of 2 available is 7 (CEILING(lg_2(66)) = 7) and the total number of bits now jumps to 2^7 * 20 * 2 + 20 * 5 (for the stack) = 5220.

5.2 Comparison between Hardware and Software

The first execution time comparison was made between the 2 versions of our solver using problems from the DIMACS benchmark suite. The purpose of this comparison is to assess

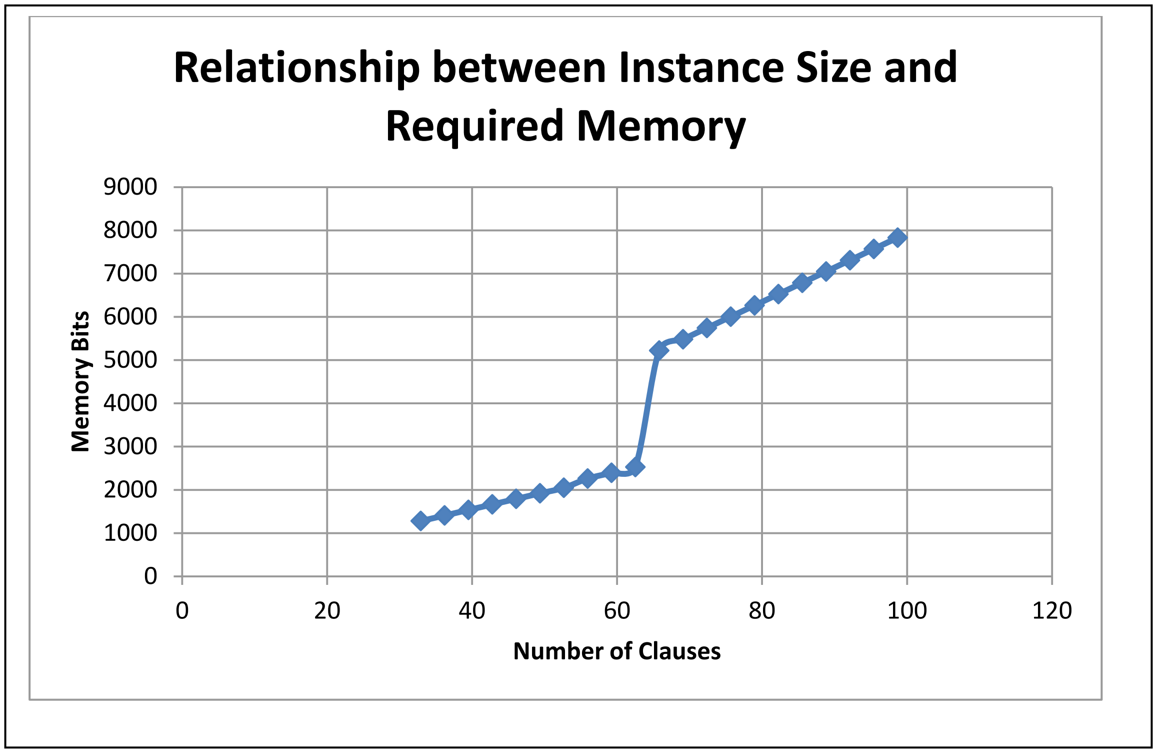


Figure 30: Memory requirements (ROM and stack) of the DP-like solver

exactly how much faster or slower the hardware version is than the software simulator. Paramount to this comparison is the fact that circuit build times were not taken into consideration when reporting speedups. As the instances used in the comparison do not fit onto the FPGA, their build times for larger platforms would necessarily take up more than the 32 minutes reported higher whereas the raw execution time for these problems is predominantly on the order of seconds. The goal here is to use run-times to characterize fine-grained parallelism as applied to solving SAT. Thusly instance specificity is pushed to the maximum (memory modules have been eliminated and all clauses are readily accessible by the solver in one clock cycle) to explore the limits of this parallelism.

Table 3 shows the run-times of the two solver flavors on various problems. The columns indicate, from left to right, the instance's name, how many clauses and variables it has, the number of simulated clock cycles (obtained from the software simulator), the estimated circuit frequency in Megahertz, the theoretical hardware execution time, the software simulation time and the resulting speedup. As can be seen from Table 3, most of the time, the hardware circuit is two orders of magnitude faster than the software simulator. On some problem instances three orders of magnitude were discovered. It is also important to note that the simulator is running on a general purpose CPU whose frequency is about two orders of magnitude faster than the estimated frequencies.

5.3 Comparison with other Solvers

Comparisons were also made with other hardware solutions that were presented in this work. An important aspect of this comparison is that all solvers used in the examination used a variant of the DP algorithm that was augmented with various sophisticated heuristics that can, in some cases, greatly help with solving instances faster. Techniques such as non-chronological backtracking whereby the solver backtracks not to the most recent variable

Table 3: Comparison between HW and SW versions of our DP-like solver

Problem Name	Clauses	Vars.	Clock Cycles	Freq. (MHz.)	Time (sec)	Simulation Time (s)	Speedup
dubois20	160	60	5.77E+07	18.73	3.08E+00	4.65E+02	1.51E+02
dubois21	168	63	1.21E+08	18.13	6.65E+00	1.00E+03	1.51E+02
aim-50-2_0-yes-1-2	100	50	1.69E+03	25.62	6.58E-05	6.80E-02	1.03E+03
aim-100-2_0-yes-1-4	200	100	4.66E+07	16.14	2.89E+00	5.38E+02	1.86E+02
aim-200-6_0-yes-1-1	1200	200	3.54E+05	4.89	7.24E-02	1.33E+01	1.84E+02
par8-1-c	254	64	1.39E+02	13.76	1.01E-05	3.20E-02	3.17E+03
par16-1-c	1264	317	1.76E+07	4.72	3.73E+00	1.03E+03	2.76E+02
pret60_40	160	60	6.49E+07	18.73	3.46E+00	7.36E+02	2.13E+02
hole9	415	90	2.11E+08	9.92	2.13E+01	2.71E+03	1.27E+02
hole8	297	72	1.37E+07	12.40	1.10E+00	2.50E+02	2.26E+02
hole7	204	56	9.77E+05	15.93	6.13E-02	1.82E+01	2.97E+02
hole6	133	42	7.80E+04	21.19	3.68E-03	2.37E+00	6.43E+02
uuf100-0457	430	100	3.09E+06	9.69	3.19E-01	6.03E+01	1.89E+02
uuf125-07	538	125	8.26E+06	8.34	9.90E-01	1.54E+02	1.55E+02
aim-100-1_6-yes1-1	160	100	9.00E+08	18.73	4.81E+01	4.80E+03	9.98E+01
aim-50-2_0-no-4	100	50	2.81E+05	25.62	1.10E-02	2.46E+00	2.24E+02
aim-50-1_6-no-1	80	50	7.31E+06	29.73	2.46E-01	4.45E+01	1.81E+02
aim-100-3_4-yes1-4	340	100	2.84E+05	11.33	2.50E-02	5.16E+00	2.06E+02
aim-50-2_0-no-1	100	50	3.61E+06	25.62	1.41E-01	2.39E+01	1.70E+02

decision but to the variable decision that has caused the contradiction can prune the search space even further by avoiding dead-ends in the search path. Heuristics such as these were not implemented for the presented solver because it was deemed that the basic DP algorithm was sufficient for the purposes of our investigation. A mature, powerful solver would, however, require some of these state-of-the art methods to increase its efficiency.

Table 4 and Table 6 report the run-times obtained for problem instances from the DIMACS benchmark set. Considering the older solvers presented in Table 4, we can see that some accelerations are achieved. On the other hand, the newer solvers presented in Table 6 are faster on most instances. Of note is the fact that execution times presented in [11] by Gulati et al. for their 2009 solver are projections to a large, industrial grade FPGA platform. In addition, it is worth remarking that the solver presented by Safar et al. in their 2011 paper [6] is capable of much faster execution on a specific set of benchmark problems, namely the aim set. This may be the result of Safar et al.'s implementation of heuristics that are suitable for this particular type of problem.

Finally, a very important result was obtained by comparing our simulation results of

Table 4: Comparison of HW version of our solver with other HW solvers

Problem Name	**Zhong et al. (2000)**		**Suyama et al. (2001)**		**Skliarova and Ferrari[a] (2004)**	
	Time (s)	***Speedup***	***Time (s)***	***Speedup***	***Time (s)***	***Speedup***
dubois20	8.44E+00	2.74E+00	2.07E+01	6.72E+00	N/A	
dubois21	N/A		4.26E+01	6.40E+00		
aim-50-2_0-yes-1-2	4.00E-04	6.08E+00	N/A			
aim-100-2_0-yes-1-4	9.70E+00	3.36E+00				
aim-200-6_0-yes-1-1	8.90E-01	1.23E+01				
par8-1-c	3.50E-05	3.47E+00				
par16-1-c	2.20E+00	5.90E-01				
pret60_40	9.00E+00	2.60E+00				
hole9	N/A				5.36E+00	2.51E-01
hole8					8.88E-01	8.05E-01
hole7					3.98E-01	6.49E+00
hole6					3.89E-01	1.06E+02

[a] Values reported for authors c256 circuit. The time used for acceleration computation is the reported t_{total} parameter.

Table 6: Comparison of HW version of our solver with other HW solvers cont'd

Problem Name	Gulati et al. (2009)		Safar et al. (2011)	
	Time (s)	*Speedup*	*Time (s)*	*Speedup*
aim-50-2_0-yes-1-2	1.79E-06	2.72E-02	5.20E-05	7.91E-01
par8-1-c	N/A		2.03E-05	2.01E+00
pret60_40			1.11E-02	3.20E-03
hole9			1.47E+00	6.90E-02
hole8	1.51E-01	1.37E-01	1.43E-01	1.30E-01
hole7	N/A		1.53E-02	2.49E-01
hole6	2.73E-03	7.42E-01	1.81E-03	4.92E-01
uuf100-0457	3.02E-02	9.46E-02	5.90E-01	1.85E+00
uuf125-07	1.04E+00	1.05E+00	N/A	
aim-100-1_6-yes1-1	N/A		1.42E-04	2.96E-06
aim-50-2_0-no-4			1.30E-03	1.18E-01
aim-50-1_6-no-1			2.50E-05	1.02E-04
aim-100-3_4-yes1-4			9.40E-02	3.76E+00
aim-50-2_0-no-1			7.29E-05	5.18E-04

Table 5: Run-time comparison of HW version of our solver with MiniSAT

Problem Name	MiniSAT	
	Time (s)	*Speedup*
hole9	8.68E+00	4.07E-01
hole8	4.32E-01	3.92E-01
hole7	6.40E-02	1.04E+00
hole6	4.00E-03	1.09E+00

run-times of problems from the difficult hole suite of the DIMACS set with those of MiniSAT, a very efficient software solver. These problems are unsatisifiable and are very laboriously solved by software. Table 5 relates MiniSAT's execution time as well as the resulting speedup. On the smaller of the 4 problems tested, hole6 and hole7, our solver is actually able to execute slightly faster whereas on the remaining two problems, though slower, it is still able to provide a solution within one order of magnitude. These results are important as our future goal is to develop a hardware-based FPGA solver that is able to outperform efficient software solvers on large, difficult, industrial-type problems.

5.4 Effect of Pseudo-Randomness on SAT Resolution

In our solver, the first decision when needing to assign a value to a variable is to always initially to try ‘0′. This is, arbitrary and may or may not help with finding a solution faster. There are methods available, such as the experimental unit propagation (EUP) that is used in [17], that attempt to assign variable values based on some heuristic to find solutions faster. The EUP technique assigns both values to a variable and attempts to verify these decisions in parallel. However, a resource price must be paid for this computation.

To avoid having to always pick false as an initial variable assignment we have decided to use an LFSR to pseudo-randomly decide variable values. The LFSR is given an initial seed obtained from the operating system as the Java program creates the ROM memories as described higher. For a given seed the same solution will be found. However, it is possible to obtain different solutions (with the HW version) if solver is reset (in the current implementation the LFSR is always running and is never reset with the rest of the solver).

To characterize the influence of the LFSR, several problems from the same DIMACS set were selected (Table 7, Figure 31). As before, these problems were not synthesized but rather the software simulator was used to obtain clock cycle counts. Unlike the previous

Table 7: Comparison between binary and LFSR decision modes (clock cycle counts)

Problem Name	Binary	LRSF (Avg. over 100)	Speedup
uf20-01	195	104	1.875
uf50-01	8735	8466	1.031774
uf50-02	1821	4462	0.408113
uf75-01	52625	36276	1.450684
uf100-01	114135	328309	0.347645
hole6	77954	77954	1
hole7	976947	976947	1
par8-1-c	139	423	0.328605
par8-1	6171	5260	1.173194
aim-50-2_0-no-4	281379	274427	1.025333
aim-50-2_0-yes1-2	1685	1861	0.905427
aim-100-3_4-yes1-4	283538	613125	0.462447
ii8a1	210	155	1.354839
ii8a2	568857	194352	2.926942
jnh1	1084	120029	0.009031
jnh2	52632	56391	0.93334

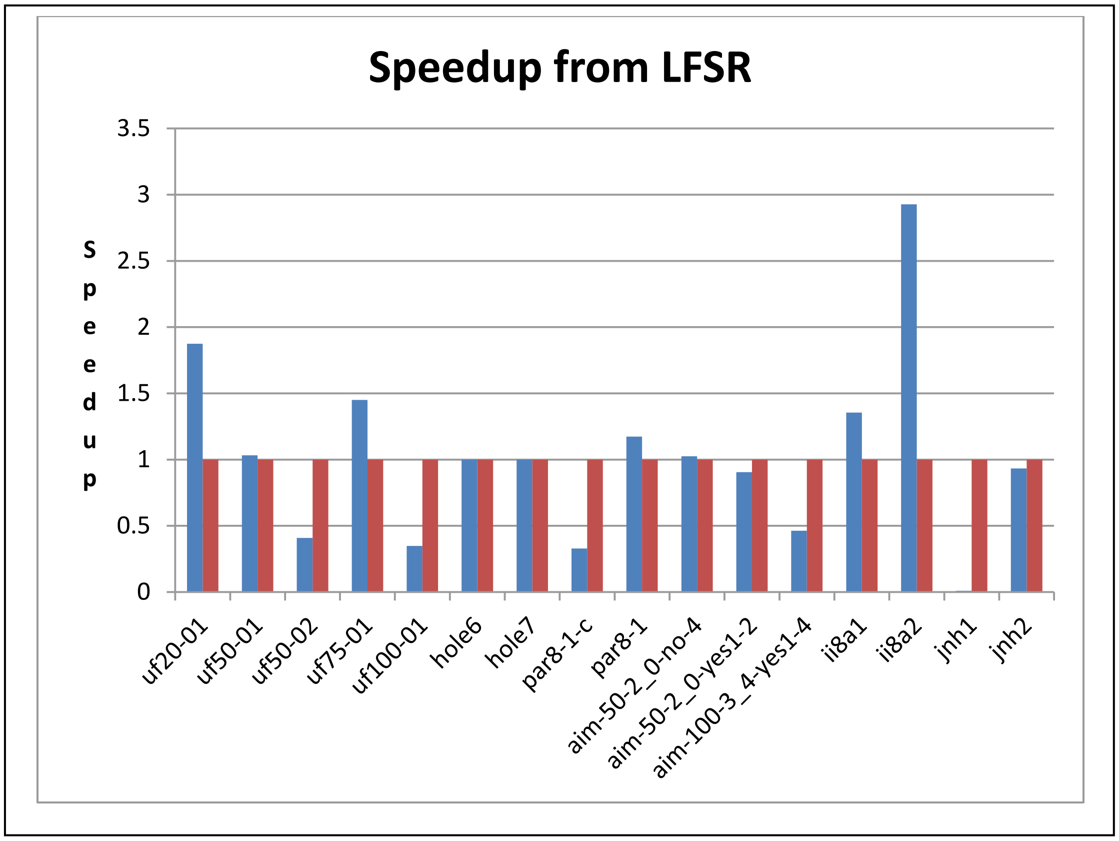


Figure 31: Speedup obtained from using LFSR in variable decision assignments

section, where the regression analysis was used to infer approximate circuit frequencies, only clock cycle counts were used for comparison. Initially, the problem is run with the simulator in 'binary assignment mode' whereby each variable gets '0' as a first assignment. Subsequently, the same problem is executed 100 times using random variable assignments from the LFSR after which an average is computed. It should be mentioned that in the simulator, the LFSR is modeled by the Random Java object. Table 7 shows, for the problems chosen, the number of clock cycles in binary assignment mode, the average number of clock cycles in LFSR assignment mode and the resulting speedup. Figure 31 graphically illustrates these results in the following manner. The horizontal axis indicates the problem name while the vertical axis indicates the speedup obtained over the binary assignment mode. This latter result (binary assignment speedup) is always shown as 1 on the graph.

Interestingly, on two of the hole problems the LFSR seems to have no influence whatsoever as the clock cycle counts are identical. In addition, other families of problems such as the uf class seem to exhibit accelerations in some cases and not in others. Moreover, the aim family of problems, with the exception of aim-100-3_4-yes1-4, do not seem to be as influenced by the LFSR, although not to the extent of the hole problems. Finally, other problem families such as par and jnh demonstrate rather unpredictable results.

In conclusion, it is difficult to characterize the effect of using a pseudo-random manner in deciding variable assignments. The average speedup conferred on all problems test is about 1.01 if the ii8a2 result is considered. However, removing this problem from the average speedup calculation yields an average speedup of about 0.89. Given that the LFSR requires FPGA resources, it would seem that always deciding false as an initial variable assignment is appropriate when compared to the pseudo-random LFSR variable decision method.

5.5 Computing All Implications in One Clock Cycle

The DP-like solver presented here has much room for improvement. Indeed, one example of just such an alteration that is conducive towards the improvement of the solver is the computation of all implications in one clock cycle.

To illustrate this, consider a hypothetical SAT instance with 500 variables and 1500 clauses. The solver starts deciding variables starting with index 0. In order, variables 0 to, say, 9 are decided without having any implications or contradictions raised. All of a sudden, after deciding variable 10 to 0, as is currently done in our solver, variables 450 to 499 become implication variables. In addition, consider that variable 499 causes a contradiction. The solver starts its implication phase, which lasts 50 clock cycles as there are 50 variables to imply. During cycles 0 – 48, no contradiction is found. As the solver gets to 49, the contradiction is raised, and the solver backtracks. As the last decision variable is 10, the solver pops the stack, inverts variable 10 to true and resets all previously computed implications. If the solver had been able to assign all 50 implications in the first cycle, 49 clock cycles would have been saved. Now, assume that perhaps it is variable 0 being assigned false that causes this contradiction. The solver must compute all 50 implications (with the 50th always causing a contradiction) 10 times (as it must backtrack all the way to the first variable) before variable 0 is flipped and the solver can assign the 50 implications without contradiction. This further exacerbates the number of clock cycles lost. To show just how much determining all implications in one clock cycle is beneficial, consider Table 8 and Figure 32. In Table 8, the "Clock Cycles" column is divided in two columns, one giving the number of clock cycles necessary to solve the respective SAT instance when 1 implication computation is performed per clock cycle and the other when all implications are done in one clock cycle. The chart shown in Figure 32 indicates that significant speedups are achieved for all problems tested (maximum of over 15 and minimum of about 3 times faster). Coming back to the comparisons with other hardware solvers we observe that our solver model is now faster than almost all others. For example, Skliarova and Ferrari's solver was faster on the hole9 and hole8 problems by one order of magnitude. Looking at Figure 32, these problems execute about 12 times and 10 times faster than before which allows us to make up this order of magnitude. Our solver is now about 2 and 8 times faster on these problems. The same can be said about most other solvers with the sole exception being Safar et al.'s solver which is still sometimes faster

Table 8: Improvement in performance by calculating all implications in 1 clock cycle

Problem Name	**Clock Cycles**		**Speedup**
	(1 implication)	***(all implications)***	
dubois20	5.77E+07	1.26E+07	4.58E+00
dubois21	1.21E+08	2.52E+07	4.79E+00
aim-50-2_0-yes-1-2	1.69E+03	5.65E+02	2.98E+00
aim-100-2_0-yes-1-4	4.66E+07	9.15E+06	5.10E+00
aim-200-6_0-yes-1-1	3.54E+05	3.73E+04	9.49E+00
par8-1-c	1.39E+02	1.30E+01	1.07E+01
par16-1-c	1.76E+07	1.13E+06	1.56E+01
pret60_40	6.49E+07	1.09E+07	5.93E+00
hole9	2.11E+08	1.69E+07	1.25E+01
hole8	1.37E+07	1.30E+06	1.05E+01
hole7	9.77E+05	1.13E+05	8.66E+00
hole6	7.80E+04	1.12E+04	6.98E+00
uuf100-0457	3.09E+06	3.73E+05	8.29E+00
uuf125-07	8.26E+06	8.93E+05	9.25E+00
aim-100-1_6-yes1-1	9.00E+08	2.42E+08	3.72E+00
aim-50-2_0-no-4	2.81E+05	8.54E+04	3.30E+00
aim-50-1_6-no-1	7.31E+06	2.51E+06	2.91E+00
aim-100-3_4-yes1-4	2.84E+05	5.56E+04	5.10E+00
aim-50-2_0-no-1	3.61E+06	9.80E+05	3.68E+00

(for example on the aim-100-1_6-yes1-1 problem we only gain speedup of about 4 which does not defeat the 6 orders of magnitude by which the Safar solver is faster).

In addition, when considering the comparison with MiniSAT, the gained speedup of computing all implications in one clock cycle now allows our solver to be faster on all problems tested. For example, hole7 is now executed almost one order of magnitude faster than before. This result indicates that it is possible for a bare (no heuristics) hardware implementation of the DP algorithm to outperform a very efficient software solver.

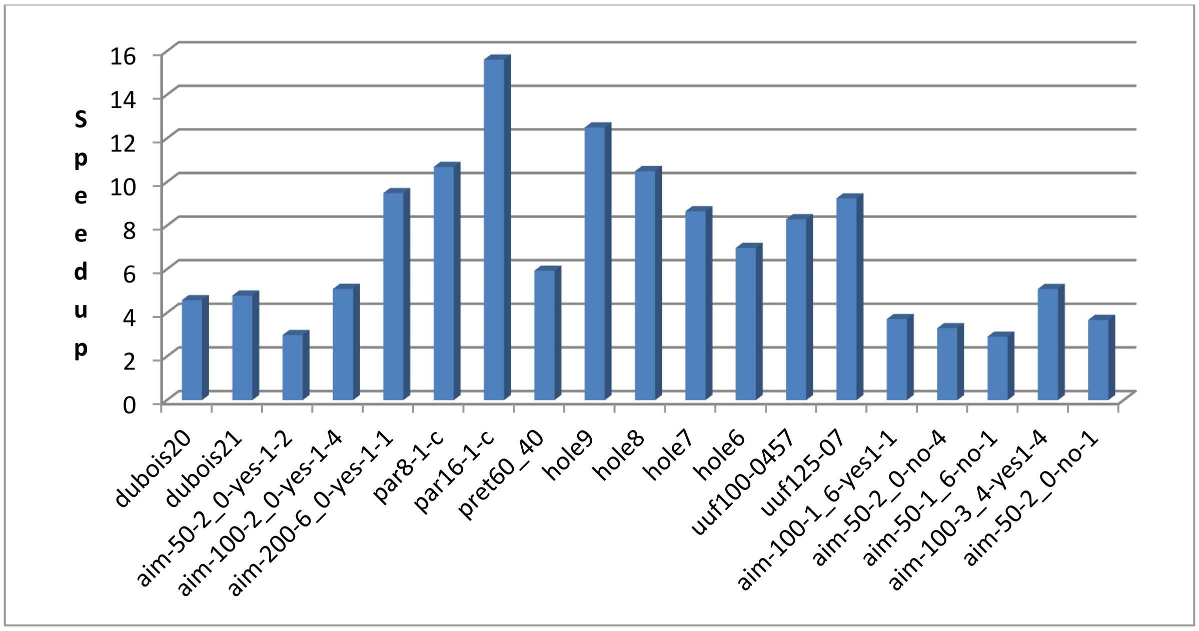


Figure 32: Speedup obtained from computing all implications in one clock cycle

Chapter 6 – Conclusion and Future Work

Though the Boolean satisfiability problem has seen dramatic improvement in its methods and techniques in the past two decades it remains hard to solve. In addition to its computational intractability, researchers have found it difficult to link SAT problem structure to the heuristics and techniques implemented in modern, state-of-the-art software CDCL solvers (like MiniSAT) that can reliably solve various problem instances that model real-life applications. As a starting point to further improve SAT solving performance, researchers propose to establish analytical/theoretical models that are able to predict the performance of these CDCL solvers on SAT instances [34]. Examples of CDCL attributes that can be considered are the symmetries of CNF formulas, the cut width of graph representations of CNF instances as well as the scale-free graph structure of industrial problems [34].

Although desirable in a mature solver implementation, there are no CDCL features present in the solver that is the subject of this work. Indeed, the approach presented here is one that focuses on exploiting to the utmost extent the strengths of an FPGA platform and characterizing their suitability to SAT solving. For example, an instance-specific approach was chosen despite the modern shift of hardware solvers towards application specificity because FPGAs are reprogrammable devices. A hugely important factor that has motivated this shift has been the extremely long compilation time of instance-specific circuits. This can also be seen the present work as a circuit that spans the entire Cyclone II FPGA would require over half an hour of compilation time. If a software solver is able to find a solution in less time there is no need to use hardware to accelerate SAT. On the other hand, application-specific solvers have their own limitations. For example, memory bandwidth may be considered as the greatest one. In a SAT solver that is based on the DP method, many contradictions are raised and clauses are continuously evaluated; accessing a memory module to read clause information greatly diminishes the acceleration that is obtained from the hardware.

We believe that a return toward instance specificity, with a focus on how a problem is represented and partitioned constitutes a viable option when solving SAT with reconfigurable hardware so as to obtain accelerations. As was seen with the work of Zhong et al. [21] techniques exist that are able to reduce compilation times essentially to zero. In addition, instance specificity allows for very fine granularity. The basic unit that is treated in our solver is the clause (compared to a pipeline of clause modules). In the same clock cycle, all clauses are presented to and are easily accessed by the decision machine of our solver and there is no need for general purpose memories. As was stated previously, the solver model that was implemented is not to be considered as a final product but rather an assessment platform, composed of a synthesizable VHDL model used for circuit frequency inference, as well as of a functionally identical software simulator homologue that we used to identify the fact that memory bottlenecks are not acceptable in a future version of our solver. In addition, we have seen that the solver is able to keep up (in raw hardware execution time as the circuits could not be synthesized for our board) with a very sophisticated software solver and even outperform it with the modification of computing all implications in 1 clock cycle.

6.1 A Review of Our Contributions

One of the most important aspects of this thesis is, in our opinion, a comprehensive exploration of the existing methodologies and techniques that have been employed in solving the Boolean Satisfiability problem on FPGA. Following this exploration, it was determined that an instance-specific approach that entails infusing problem clauses into a corresponding hardware circuit constitutes a very promising avenue to obtaining high-performing hardware SAT solvers. The largest obstacle to this approach is the method by which this information was injected into the circuit (by means of registers).

Useful in making the above claim is the exploratory testing platform that was constructed. The software simulator constitutes a useful tool for approximating and characterizing SAT instances without having to go through the (sometimes long) process of compiling and synthesizing a complete circuit. In addition, this simulator can also serve as an initial evaluator for new features that would be added to the hardware version. By first trying these heuristics in the software simulator, one may assess whether or not they would be useful to the hardware solver. This latter version of our solver thus constitutes a starting base for a more efficient, future solver implementation.

By using these two counterparts, we have determined (by measuring our solver on DIMACS benchmark problems) that the hefty memory constraint is considerable. Researchers, by wanting to avoid long compilation times and turning towards application specificity, have tried to alleviate this constraint by adding heuristics to their solvers. However, for large problems, memory latencies and physical port constraints will have an even greater impact on SAT solving that will dilute the effect of these heuristics. On the other hand, we envision targeting our future efficient implementation to large problems that are difficult to solve (or perhaps impossible in an adequate time frame) by the best current state-of-the-art software solvers. In this arena, a circuit's long build time is not as important and the large fine-grained FPGA parallelism coupled with key heuristics may provide faster solutions to these problems.

6.2 Future directions – Eliminating Memory Altogether and More

The current main focus of our work is to eliminate having to deal with memory altogether, whether it is on-chip, off-chip or even register arrays. In addition to tying up logic elements, registers also require a great deal of wiring resources on the FPGA. It is therefore paramount that we find a way to represent the problem instance other than by placing clause information in a ROM.

Additionally, in order to be successful, a full implementation of our solver requires several additions. The simplest feature, which was shown earlier as having a big impact, is to allow the solver to compute all implications generated by variable decisions in one clock cycle. In this manner, contradictions will be raised faster. A second technique to be incorporated is the implementation of some sort of dynamic variable ordering (when choosing decision variables). The MOM technique mentioned previously is a simple but powerful candidate that we will consider. Thirdly, in an effort to explore and instance's search space more efficiently so as to avoid dead ends, non-chronological backtracking is required. For example, a simple solution would be to have an array of 1-bit registers that would record whether a variable has been decided or implied. A second array of integers would record, when applicable, the index of the variables that have implied other variables. In this manner, when a contradiction is raised there is information available to backtrack to the appropriate level in the search tree. Of course, this is a rather hardware heavy method and alternatives of keeping track of this information can be examined. Finally, it would be possible to bestow upon our solver the ability to learn from its mistakes. Conflict driven clause learning, whereby new clauses are added to the original problem set to make sure that discovered contradictions do not recur, is to be considered. A way to implement this feature would be to have a small memory where extra clause information can be added dynamically.

Finally, as a longer term research avenue, it is worthwhile investigating building custom CAD tools that are tailored to our needs. As the decision machine of the solver is pretty much fixed and does not need to change with each problem instance, it could possibly be synthesized only once. All other circuits, such as the backtrack stack, can be synthesized for each problem. Having this custom tool would avoid generic CAD tools optimizations that are perhaps not necessary for our purposes.

Bibliography

1. Biere, A., *Handbook of satisfiability*. Vol. 185. 2009: IOS Press.
2. Abramovici, M. and D. Saab. *Satisfiability on reconfigurable hardware*. in *Field-Programmable Logic and Applications*. 1997. Springer.
3. Svoboda, A. *Boolean analyzer*. in *Proceedings of IFIP Conference*. 1968. Edinburgh, UK: North-Holland
4. Yung, W., et al. *A runtime reconfigurable implementation of the GSAT algorithm*. in *Field Programmable Logic and Applications*. 1999. Springer.
5. de Sousa, J., J. Da Silva, and M. Abramovici. *A configurable hardware/software approach to SAT solving*. in *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. 2001. IEEE.
6. Safar, M., et al. *A reconfigurable, pipelined, conflict directed jumping search SAT solver*. in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. 2011. IEEE.
7. Safar, M., et al. *A Shift Register based Clause Evaluator for Reconfigurable SAT Solver*. in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*. 2007. IEEE.
8. Platzner, M. and G. De Micheli, *Acceleration of satisfiability algorithms by reconfigurable hardware.* Field-Programmable Logic and Applications From FPGAs to Computing Paradigm, 1998: p. 69-78.
9. Suyama, T., M. Yokoo, and H. Sawada. *Solving satisfiability problems using logic synthesis and reconfigurable hardware*. in *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*. 1998. IEEE.
10. Kanazawa, K. and T. Maruyama, *An Approach for Solving Large SAT Problems on FPGA.* ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2010. **4**(1): p. 10.
11. Gulati, K., et al., *FPGA-based hardware acceleration for Boolean satisfiability.* ACM Trans. Design Autom. Electr. Syst, 2009. **14**(2).
12. Qasim, S.M., S.A. Abbasi, and B. Almashary. *A review of FPGA-based design methodology and optimization techniques for efficient hardware realization of computation intensive algorithms*. in *Multimedia, Signal Processing and Communication Technologies, 2009. IMPACT'09. International*. 2009. IEEE.
13. Skliarova, I. and A.d.B. Ferrari, *A software/reconfigurable hardware SAT solver.* Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2004. **12**(4): p. 408-419.
14. Dandalis, A. and V.K. Prasanna, *Run-time performance optimization of an FPGA-based deduction engine for SAT solvers.* ACM Transactions on Design Automation of Electronic Systems (TODAES), 2002. **7**(4): p. 547-562.
15. Zhong, P., et al., *Using configurable computing to accelerate Boolean satisfiability.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 1999. **18**(6): p. 861-868.

16. Leong, P.H., et al., *A bitstream reconfigurable FPGA implementation of the WSAT algorithm.* Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2001. **9**(1): p. 197-201.
17. Suyama, T., M. Yokoo, and A. Nagoya. *Solving satisfiability problems on FPGAs using experimental unit propagation*. in *Principles and Practice of Constraint Programming–CP'99*. 1999. Springer.
18. Suyama, T., et al., *Solving satisfiability problems using reconfigurable computing.* Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2001. **9**(1): p. 109-116.
19. Yokoo, M., T. Suyama, and H. Sawada. *Solving satisfiability problems using field programmable gate arrays: First results*. in *Principles and Practice of Constraint Programming—CP96*. 1996. Springer.
20. Zhong, P., et al. *Using reconfigurable computing techniques to accelerate problems in the CAD domain: a case study with Boolean satisfiability*. in *Proceedings of the 35th annual Design Automation Conference*. 1998. ACM.
21. Zhong, P., M. Martonosi, and P. Ashar. *FPGA-based SAT solver architecture with near-zero synthesis and layout overhead*. in *Computers and Digital Techniques, IEE Proceedings-*. 2000. IET.
22. Abramovici, M. and J.T. De Sousa, *A SAT solver using reconfigurable hardware and virtual logic.* Journal of Automated Reasoning, 2000. **24**(1): p. 5-36.
23. Redekopp, M. and A. Dandalis, *A Parallel Pipelined SAT Solver for FPGA's.* Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing, 2000: p. 462-468.
24. Mencer, O. and M. Platzner. *Dynamic circuit generation for boolean satisfiability in an object-oriented design environment*. in *System Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*. 1999. IEEE.
25. Reis, N. and J. de Sousa. *On implementing a configware/software SAT solver*. in *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*. 2002. IEEE.
26. Cook, S.A. *The complexity of theorem-proving procedures*. in *Proceedings of the third annual ACM symposium on Theory of computing*. 1971. ACM.
27. Krom, M.R., *The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary.* Mathematical Logic Quarterly, 1967. **13**(1-2): p. 15-20.
28. Karp, R.M., *Reducibility among combinatorial problems.* 50 Years of Integer Programming 1958-2008, 2010: p. 219-241.
29. Gu, J., et al., *Algorithms for the Satisfiability (SAT) Problem: A Survey.* Handbook of Combinatorial Optimization: Supplement, 1999: p. 379-510.
30. Marques-Silva, J. *Practical applications of Boolean satisfiability*. in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*. 2008. IEEE.
31. Järvisalo, M., et al., *The International SAT Solver Competitions.* AI Magazine, 2012. **33**(1): p. 89-92.
32. Marques-Silva, J.P. and K.A. Sakallah, *GRASP: A search algorithm for propositional satisfiability.* Computers, IEEE Transactions on, 1999. **48**(5): p. 506-521.
33. Moskewicz, M.W., et al. *Chaff: Engineering an efficient SAT solver*. in *Proceedings of the 38th annual Design Automation Conference*. 2001. ACM.

34. Katebi, H., K. Sakallah, and J. Marques-Silva, *Empirical study of the anatomy of modern sat solvers.* Theory and Applications of Satisfiability Testing-SAT 2011, 2011: p. 343-356.
35. Todman, T.J., et al. *Reconfigurable computing: architectures and design methods*. in *Computers and Digital Techniques, IEE Proceedings-*. 2005. IET.
36. Manohararajah, V., S.D. Brown, and Z.G. Vranesic, *Heuristics for area minimization in LUT-based FPGA technology mapping.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2006. **25**(11): p. 2331-2340.
37. Wu, J.-O., Y.-H. Fan, and S.-F. Wang. *FPGA placement methodology based on grey relational clustering*. in *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*. 2012. IEEE.
38. Porter, B., *Handbook of knowledge representation*. 2008, Elsevier Science Limited.
39. Davis, M., G. Logemann, and D. Loveland, *A machine program for theorem-proving.* Communications of the ACM, 1962. **5**(7): p. 394-397.
40. Skliarova, I. and A. de Brito Ferrari, *Reconfigurable hardware SAT solvers: A survey of systems.* Computers, IEEE Transactions on, 2004. **53**(11): p. 1449-1461.
41. Selman, B., H. Levesque, and D. Mitchell. *A new method for solving hard satisfiability problems*. in *Proceedings of the tenth national conference on Artificial intelligence*. 1992.
42. Selman, B., H. Kautz, and B. Cohen, *Local search strategies for satisfiability testing.* Cliques, coloring, and satisfiability: Second DIMACS implementation challenge, 1993. **26**: p. 521-532.
43. Goel, P., *An implicit enumeration algorithm to generate tests for combinational logic circuits.* Computers, IEEE Transactions on, 1981. **100**(3): p. 215-222.
44. Chung, C. and P. Leong. *An architecture for solving boolean satisfiability using runtime configurable hardware*. in *Parallel Processing, 1999. Proceedings. 1999 International Workshops on*. 1999. IEEE.
45. Gulati, K., et al., *Efficient, scalable hardware engine for Boolean satisfiability and unsatisfiable core extraction.* Computers & Digital Techniques, IET, 2008. **2**(3): p. 214-229.
46. Smith-Miles, K. and L. Lopes, *Measuring instance difficulty for combinatorial optimization problems.* Computers & Operations Research, 2012. **39**(5): p. 875-889.
47. Wolpert, D.H. and W.G. Macready, *No free lunch theorems for optimization.* Evolutionary Computation, IEEE Transactions on, 1997. **1**(1): p. 67-82.
48. Selman, B., D.G. Mitchell, and H.J. Levesque, *Generating hard satisfiability problems.* Artificial intelligence, 1996. **81**(1): p. 17-29.