2013

ECE 595Z Project Report

SAT Solver

RICKARD EWETZ
PRASHANT LALWANI

PURDUE UNIVERISTY | Spring 2013

Table of Contents

Introduction

Satisfiability is a key concept in the EDA industry, given a Boolean formula finding a variable assignment such that the formula evaluates to True or proving no such assignment exists. As the design sizes are increasing with the shrink in transistor technology every 18 months it has become impossible to naively try all possible inputs.

In this report we discuss a SAT solver implemented for ECE 595Z, Digital System Design Automation. The SAT solver was implemented in C++ using DPLL as the basic framework and several improvements were done in order to improve its performance, the improvements being: *watched variables, conflict driven learning, non-chronological backtracking and branching heuristics.* The results were pretty substantial and a maximum speed up of 91X, minimum of 3.5X was observed while an average speed up of 24X was observed as compared to basic DPLL which itself is much faster than testing all possible set of inputs. In the proceeding sections, elements of SAT solver, a flowchart of the algorithm, data-structures used, usage methodology, benchmarking methodology and results will be discussed.

SAT solver key components

Basic DPLL: The Davis-Putnam-Logemann-Loveland algorithm is a backtracking based search algorithm introduced in 1962, capable of determining the Satisfiability of a CNF formula. Even after 50+ years it still remains the basic framework for many efficient SAT solvers. It is the first NP-complete proved problem. The algorithm starts by assigning True value to a variable and making the assignment to all the clauses and continues the variable assignments until a solution is obtained, If not it backtracks and assigns the other value(False) to the variables. It can also be viewed as a Boolean tree.

Watched Variables: An implication occurs when all variables in a clause but one are False, in such cases it is essential for the last variable to take a True value in order for the CNF formula to be satisfied. Since such a case will only happen when there is only one variable left in the clause so two literals are picked from each clause as watched variables, whenever a variable assignment is done the watched variables are updated. If any of the clause has only one watched variable at any stage then it is a unit clause and need to be forced a True or False depending on whether it appears in complemented or non-complemented form.

Conflict driven learning: Basic DPLL backtracks to the previous node in the binary tree whenever it comes across a conflict and tries the other assignment to the variable just to find out that both assignments lead to a conflict. Such cases can be avoided by learning from the conflicts and keeping a track of the previous assignments that lead to a conflict in order to avoid making same assignment again and reaching a conflict. Finding conflict clauses early in the tree and saving them in memory can drastically improve performance by avoiding traversing many parts of the tree that would lead to a conflict thus improving performance.

Non-Chronological backtracking: Conflict driven learning can even further increase performance by backtracking non-chronologically. Basic DPLL backtracks to the previous node whenever it comes across a conflict and tries the other assignment that may not always prove helpful as the conflict may be caused by a variable assignment up the tree. Learning such cases from the conflict clauses it will prove much more beneficial to keep backtracking until one arrives at a node that belongs to the conflict clause and then trying other assignment at that node thus avoiding all the assignments that would have taken place if DPLL were to backtrack one node at a time.

Chaff Variable State Independent Decaying Sum (VSIDS): Variable assignment priority should always be given to unit clause implications whenever one comes across such cases but when an independent decision needs to made then it can either be made randomly or by using any branching heuristic in order to improve performance such as starting with a literal with the highest occurrence in all clauses as it would increase the probability of satisfying more clauses faster as compared to random decisions for all Independent node assignments.

SAT solver flow

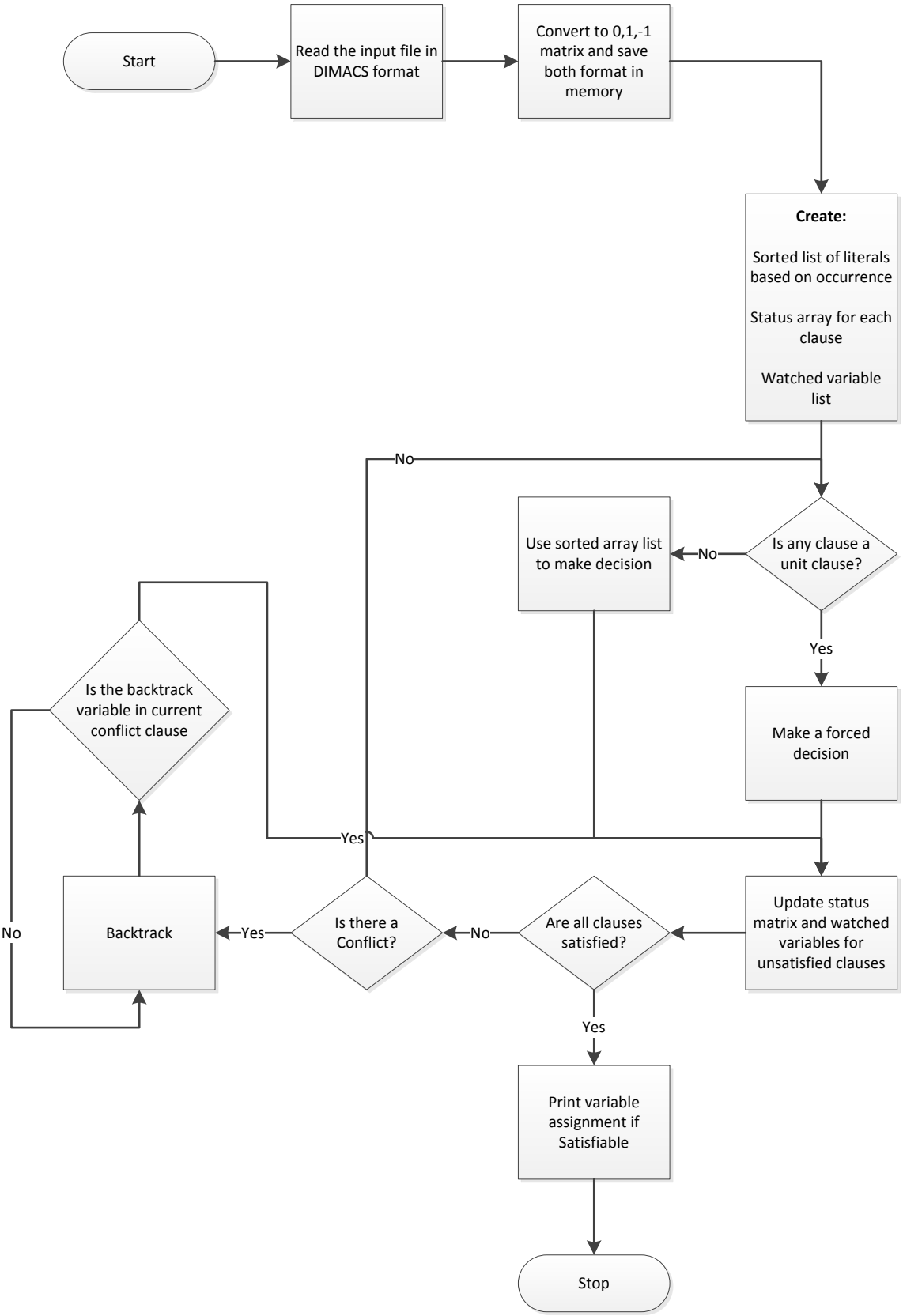

Figure 1: SAT solver flow

- The input parser starts by reading the DIMACS input file.
- The CNF formula is then saved as a 0, 1,-1 matrix where -1 and 1 specify the variable occurrence as complemented or non-complemented respectively.
- A status vector is used in order to maintain the status of all clauses as Satisfied, Unsatisfied or Conflicted.
- Two watch variables are saved for each clause.
- If a unit clause is found then a forced decision is made and clause status as well as watched variables are updated. A unit clause always gets first priority on variable assignments.
- If no unit clauses are found then a forced decision is made based on a variable with highest occurrence in the formula and the value that it takes the most.
- Watched variables and clause status are updated after every decision.
- After every assignment, conflict clauses are added (if any) to the initial formula up to a maximum of N where N is the number of initial clauses and then the older conflict clauses are replaced with newer ones.
- In case of conflicts the algorithm keeps backtracking until it reaches a node with a variable that exist in the current conflict clause.
- The entire process is repeated until the CNF formula is proved to be Satisfied or Unsatisfied.

Data Structures used

- Vectors were used in order to save the clauses in memory as they are easy to deal with when it comes to memory management. The size was kept constant to 2N where N is the number of clauses, with the lower half being used for conflict clauses.

- A 0, 1,-1 matrix was used to represent the CNF formula such that the rows represented clauses and each column represented a variable X_i making it really easy for variable assignment as only the i^{th} column was required to be assigned a value rather than searching for the variable in the entire matrix.

- At each level of the node, a list of all clauses containing that variable were stored making backtracking easy and only updating watched variables for those clauses.

Benchmarking methodology and results

Several benchmarks downloaded from [1] were ran on the SAT solver and the runtime for each case was recorded in order to analyze the speedup. Table 1 includes the run-time (hh:mm:ss) for a few of the benchmarks (arranged in descending order of runtime) that were used for testing.
Note: The Max speedup case being marked in blue.

Benchmark	Status	Basic DPLL	DPLL with Conflict learning	DPLL with Watch_Var	DPLL with Watch_Var & Conflict learning
uuf225-01.cnf	UNSAT	2:02:30	12:05.1	1:32:09	10:03.7
uuf200-02.cnf	UNSAT	1:01:51	02:55.0	47:12.5	02:38.0
uuf225-02.cnf	UNSAT	51:32.9	06:24.4	38:58.6	05:54.2
uf250-01.cnf	SAT	39:15.2	01:00.6	29:19.9	00:45.7
uuf225-03.cnf	UNSAT	34:53.5	06:41.2	26:24.8	05:29.0
uuf200-03.cnf	UNSAT	33:58.6	00:24.0	25:25.1	01:34.8
uf225-03.cnf	SAT	17:46.0	00:28.2	13:20.5	00:20.7
uuf200-01.cnf	UNSAT	14:51.0	02:33.6	11:18.3	02:19.8
uf225-01.cnf	SAT	09:31.5	00:17.3	07:13.2	00:11.5
uuf175-02.cnf	UNSAT	09:14.1	01:47.3	06:54.4	00:06.1
uuf175-03.cnf	UNSAT	05:06.1	00:31.0	03:50.8	00:27.3
uf200-01.cnf	SAT	03:37.7	00:34.7	02:45.6	00:16.8
uf150-03.cnf	SAT	02:08.6	00:22.8	01:37.3	00:15.3
uf200-03.cnf	SAT	02:08.0	00:46.3	01:36.9	00:29.3
uuf175-01.cnf	UNSAT	01:47.7	00:30.8	01:21.7	00:30.2

Figure 2: Run-time comparison of speedup techniques

From the table above it is clearly evident that using watched variables and conflict driven learning, the run-time for all the benchmarks drastically reduced. As expected the runtime trend is somewhat DPLL > DPLL with watch variables > DPLL with conflict driven learning > DPLL with watch variables and conflict driven learning. watched variables would improve the runtime significantly when there are lot many unit clauses or if fewer variable assignments lead to more unit clauses and in cases where there are lot of conflicts it is clearly evident

from benchmarks like uuf175-03 and uuf175-01 that there is a minimal difference between the runtime of Conflict driven learning with and without watched variables.

The above results clearly indicate that watched variables and conflict driven learning would both improve the performance but if used together can lead to speed up as high as 91X as evident from uuf175-02 benchmark.

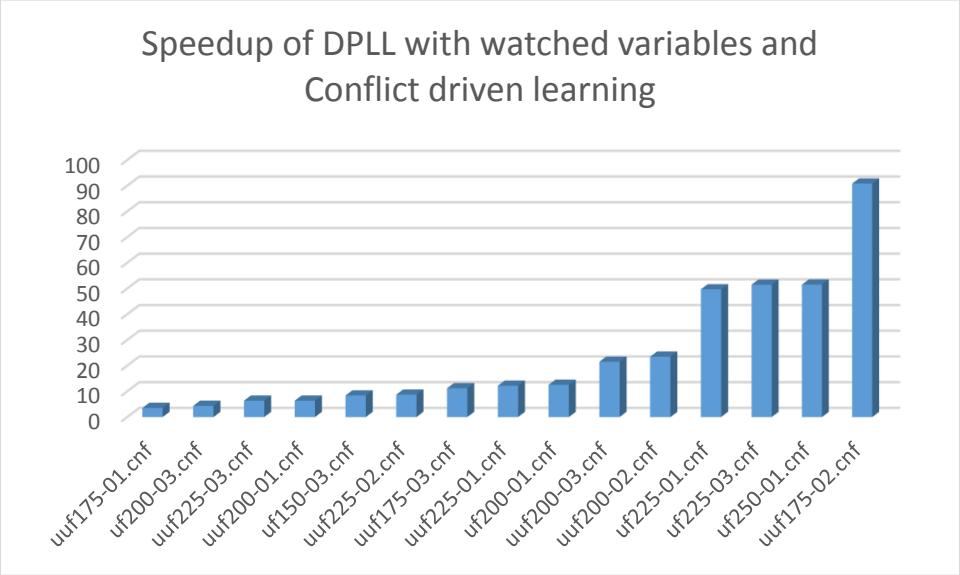

Figure 3: Runtime Speedup

Figure 3 shows a graph of speedup observed with conflict driven learning and watch variables when used together for several benchmarks. With a wide range of speed up varying from 3X to 91X, an average speed up of 24X was observed for the top 15 benchmarks out of 30+ that were tested on the SAT solver. A complete list of results is also included as s separate excel file (Results.xlsx).

Usage Methodology

The SAT solver can be compiled by using the included make file.

To run the SAT solver the following command can be executed:

./myrun input_file output_file X Y

Where X and Y are binary toggles used to turn On/Off certain functionality. The functionality being:

X -> Watched Variables
Y -> Conflict driven learning

e.g.

./myrun uf100.cnf uf100.out 1 0

Will run the SAT solver with watched variables and without conflict driven learning on uf100.cnf benchmark and will save the results in uf100.out.

Looping over Benchmarks:

- A Perl script is also included which is capable of running the SAT solver over several *.cnf benchmarks with all possible On/Off combinations of Watched variables and conflict driven learning.
- To run the script, include all your *.cnf files in benchmark folder and run the run_benchmarks.pl script.
- The script will loop over all benchmarks and will save the runtime as well as Status (SAT/UNSAT) for each benchmark in Results.csv.
- All output files will be available in output_files.tar in the benchmark folder and the runtime files in time_files.tar.

References

1. Holger H. Hoos and Thomas Stützle: *SATLIB: An Online Resource for Research on SAT.* In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000, pp.283-292, IOS Press, 2000 retrieved from http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html