CaDiCaL Modification – Watch Sat

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

***Abstract*—The solver CaDiCaL is different from other participants in SAT competitions in many aspects. Porting an algorithm detail from CaDiCaL to MergeSAT resulted in a performance degradation. Hence, this solver modification brings CaDiCaL's behavior closer to other solvers again: when watching a satisfied literal during unit propagation, the clause is moved to the watch list of that literal. Previously, CaDiCaL just updated the blocking literal of the clause and kept the clause in the current watch list. The solver CaDiCaL-watch-sat watches the satisfied literal.**

I. Unit Propagation Improvements

SAT solvers are used in many fields. Hence, some solvers are heavily tuned to perform well for the target application. Other research focusses on improving the overall solver performance in general. Many heuristic and algorithmic extensions to the core algorithm have been proposed [1]. The overall runtime distributions among the algorithm components still did not change significantly: unit propagation still takes a vast majority of the overall runtime [6], [3].

A. Watching Clauses in Propagation

This modification alters an implementation detail of unit propagation that is different in CaDiCaL when being compared to other SAT solvers that participate in competitive events. The two watched literals scheme has been implemented first in [7]. The next major improvement to skip processing clauses early was to move literals from the clause into the watch list data structure, so called *blocking literals*. Minisat 2.2 2.1 [2] started to use a blocking literal. When propagating a clause, first the current truth value blocking literal is checked. In case the blocking literal is satisfied, the related clause is known to be satisfied. Therefore, the clause does not have to be processed further. This technique helps to improve the performance of the SAT solver [6].

In Minisat 2.2, the blocking literal of a clause is typically the other watched literal. However, any other literal of the clause could be chosen.

B. How to Handle Satisfied Clauses

When a blocking literal is not satisfied, the clause has to be processed. During this process, each clause of the watch list for the current literal has to be iterated. For each clause, the truth value of all literals has to be checked, in case we find a *conflict clause* or *unit clauses* that force the extension of the current truth assignment. For satisfied clauses, we only need to process the literals until we find a satisfied clauses.

One difference between CaDiCaL and Minisat 2.2 based solvers is the way how they treat these satisfied clauses. Minisat 2.2 based solvers watch the satisfied literal. CaDiCaL implements further extensions, like memorizing the literal in a clause that was tested when last processing the clause [4].

a) Always Watching the Satisfied Literal: When a satisfied literal is detected in a clause during propagating a literal, the clause is removed from the current watch list. As a next step, solvers append the clauses to the watch list of the satisfied literal. Both operations are constant time, but require accessing the other watch list, which can lead to a cache miss [6] and TLB miss [3]. The watch list of the other literal can be higher in the search tree, so that the clause will be touched less frequent in the remainder of the search. Restart might reduce the saving, on the other hand solver today use *partial restarts* [9], *chronological backtracking* [8] as well as *trail saving* [5]. All these technique give this saving back partially.

This approach is implemented by Minisat 2.2 based solvers.

b) Just Update the Blocking Literal: As an alternative, CaDiCaL keep watching the current literal, which is now falsified, but updates the blocking literal to the satisfied literal. While this breaks the assumption that falsified literals are only watched for *conflict clauses* or *unit clauses*, we still know that the clause is satisfied. Hence, breaking this assumption does not have consequences. The positive effect is that the clause does not have to be removed from the current watch list. This results in no cache miss, nor a TLB miss. However, when the search progresses, after backtracking, the same clause might need to be processed again. In case the satisfied literal is still satisfied, only the blocking literal has to be processed. Otherwise, backtracking also removed the assignment for the blocking literal, so that the whole clause needs to be processed again.

c) Watching the Satisfied Literal in CaDiCaL*:* Preliminary testing with MergeSAT when just updating the blocking literal of a clause resulted in a performance degradation. Hence, removing this technique for CaDiCaL might result in a performance improvement. The solver CaDiCaL-watch-sat implements this modification.

Not processing a satisfied clause during propagation soon again can result in a different order of propagated literals, as well as different conflicts, and consequently in different heuristic updates and many different follow-up search steps of the solver. Hence, performance differences can not only be attributed to less or more compute resource utilization.

II. AVAILABILITY

The source of the modified CADICAL is publicly available at https://github.com/conp-solutions/cadical/tree/watch-sat. The used version of the tool is “rel-1.4.0-1-gc09aa31”.

REFERENCES

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. Amsterdam: IOS Press, 2009.

[2] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *SAT 2003*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Heidelberg: Springer, 2004, pp. 502–518.

[3] J. K. Fichte, N. Manthey, J. Stecklina, and A. Schidler, “Towards faster reasoners by using transparent huge pages,” in *Principles and Practice of Constraint Programming*, H. Simonis, Ed. Cham: Springer International Publishing, 2020, pp. 304–322.

[4] I. P. Gent, “Optimal implementation of watched literals and more general techniques,” *J. Artif. Intell. Res.*, vol. 48, pp. 231–251, 2013. [Online]. Available: https://doi.org/10.1613/jair.4016

[5] R. Hickey and F. Bacchus, “Trail saving on backtrack,” in *Theory and Applications of Satisfiability Testing – SAT 2020*, L. Pulina and M. Seidl, Eds. Cham: Springer International Publishing, 2020, pp. 46–61.

[6] S. Hölldobler, N. Manthey, and A. Saptawijaya, “Improving resource-unaware SAT solvers,” ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Heidelberg: Springer, 2010, pp. 519–534.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient SAT solver,” in *DAC 2001*. New York: ACM, 2001, pp. 530–535.

[8] A. Nadel and V. Ryvchin, “Chronological backtracking,” in *Theory and Applications of Satisfiability Testing – SAT 2018*, O. Beyersdorff and C. M. Wintersteiger, Eds. Cham: Springer International Publishing, 2018, pp. 111–121.

[9] P. van der Tak, A. Ramos, and M. Heule, “Reusing the assignment trail in cdcl solvers,” *JSAT*, vol. 7, no. 4, pp. 133–138, 2011.