PicoSAT Versions 535

Armin Biere

April 30, 2007

Abstract

Our SAT solver PicoSAT is an attempt to optimize low-level performance of BooleForce, which shares many of its key features with MiniSAT version 1.14. In this short note we describe the features of PicoSAT version 535, which is the version that was submitted to the SAT 2007 SAT Solver Competition.

1 Restarts and Phases

PicoSAT uses an aggressive *nested restart* scheme, inspired by but simpler than [2], in combination with a more sophisticated *strategy for picking the phase of decision variables*. The nested restart scheme triggers fast restarts with a high frequency. The period of fast restarts is increased by 10% after every restart until the end of the outer long period with a slow frequency. Then the long period of the outer restart interval is also increased by 10% and the fast restart interval is reset to its initial period of 100 conflicts. In addition, to avoid revisiting the same search space over and over again, the last learned clause before a restart is fixed and never deleted. Other learned clauses are garbage collected in the reduction phase as usual based on their activity [1].

The decision heuristic for the phase is as in RSAT [5]. It simply assigns the decision variable to the same value it was assigned before. Initially, as long a variable has never been assigned, PicoSAT prefers the phase with more occurrences among the original clauses and falls back to assign the variable to *false* as a tie breaker.

2 Proofs

PicoSAT's predecessor BooleForce has already been able to keep the resolution proof in memory, which greatly improves performance in applications, where clausal or variable cores or a proof trace have to be produced. This is up to an order of magnitude more efficient than writing the trace to disk and reading it back as it is necessary for proof logging versions of ZChaff and MiniSAT.

Since proofs of SAT solvers can grow very large, we employed two techniques to reduce space usage. First, clauses that become satisfied and never were used in deriving a conflict can safely be deleted. In principle, one could even go further and use reference counters for learned clauses. Clauses which are not referenced anymore can also be deleted. Another reduction is gained by sorting the clause indices of the antecedents of a learned clause, and then compress them by just storing the deltas, followed by a simple byte stream encoding. In this encoding the most significant bit of a byte flags the end of a delta, as in the binary AIGER format. In practice we obtain compression ratios close to one byte per antecedent.

3 Occurrence Lists

PicoSAT can be compiled to either use a stack based or list based occurrence list implementation. Our list based implementation of occurrence lists was developped independently but shares many features with the implementation of occurrence lists of the original Chaff solver [3]. Note that zChaff, for which source code became available earlier, and which inspired many state-of-the-art solvers uses a stack based implementation of occurrence lists.

PicoSAT can also use a compact representation of binary clauses as in [4]. The performance gain with our list based implementation is in the same order as the speed-up that can be obtained by treating binary clauses as in [4]. A detailed comparison of the run-time of various versions of PicoSAT will be reported elsewhere.

In the SAT Race 2006 the new version 535 of PicoSAT, as submitted to the SAT 2007 SAT Solver Competition, would have been the fastest solver. PicoSAT 535 is able to solve 78 instances while the winner of the SAT Race, MiniSAT version 2.0, only solved 73. PicoSAT 535 does not use any preprocessor yet. We expect preprocessing to improve performance even further.

4 Determinism

Considerable effort has been invested to make PicoSAT independent of platform and compiler. PicoSAT has its own simple floating point code for handling activity scores, an important innovation in MiniSAT 1.14. MiniSAT may produce different search trees on different platforms or with different compilers, because it relies on native floating point numbers.

Producing deterministic behavior when switching between stacks and lists was not hard to achieve. Initially, disabling or enabling special treatment of binary clauses, produced quite different search trees. The first necessary adjustment was to base the reduction schedule for garbage collection of learned clauses on the number of large clauses alone and ignore binary clauses. Delaying for instance reduction by one conflict alone can already change the search tree dramatically. We also had to make sure that during the analysis phase in backtracking the implication graph is traversed in exactly the same order.

5 Profiling

Most time is spent in BCP. But still a non negligible portion of the run time is spent in disconnecting satisfied or less active clauses.

Originally we implemented the same algorithm as in MiniSAT for disconnecting watched clauses which became satisfied or garbage in a reduction process. This algorithm is a linear search through the whole stack respectively list of clauses for a certain literal and removes individual clauses. It obviously has a quadratic accumulated worst case complexity in the number of clauses to be disconnected.

An improvement would be to use doubly linked lists. However, this only works for our list based implementation, and would require two more link fields in the clause header. The alternative, which we eventually implemented, simply delays disconnecting individual clauses as soon a clause becomes garbage. After all garbage clauses are marked, the collection phase is started, which goes through all occurrence lists respectively stacks of all literals only once and removes references to garbage clauses.

Still, as our experiments showed, flushing references to garbage clauses in the stack based implementation is much faster than traversing lists in our new implementation. We believe that this effect is due to the fact that in the list based implementation touching larger headers of clauses with two link fields is less cache friendly than just traversing the stack and skipping references to clauses marked garbage. In the latter case only one word of each clause needs to be read while in the former at least three, e.g. one containing the garbage flag, at least one literal to determine the correct link field, and the link.

6 Conclusion

The performance of PicoSAT not only benefits from high level features, such as rapid restarts and more intelligent phase assignment, but also from carefully engineered low-level data structures and algorithms, including a *real* list based occurrence list implementation and special treatment of binary clauses. In an extended version of this short note we will give more experimental evidence to these claims.

References

[1] E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust Sat-Solver. In *Proc. DATE'02*.

[2] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47, 1993.

[3] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC'01*.

[4] S. Pilarski and G. Hu. Speeding up SAT for EDA. In *Proc. DATE'02*.

[5] T. Pipatsrisawat and A. Darwiche. SAT solver description: RSat. SAT Race'06.