Solvers with a Bit-Encoding Phase Selection Policy and a Decision-Depth-Sensitive Restart Policy

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn

***Abstract*—To develop more efficient SAT solvers, recently we proposed two new solving policies. One is called bit-encoding phase selection policy, which aims at selecting more exactly the polarity of a decision variable. Another is called a decision-depth-sensitive restart policy, which determines when a restart begins, depending on search depths. Based on the two new policies, we developed a number of new SAT solvers, which are named glue_bit, minisat_bit, gluebit_lgl and gluebit_clasp. This paper describes briefly them.**

I. Introduction

Most of modern solvers are based on Conflict Driven Clause Learning (CDCL), which is a variant of DPLL procedure. In general, CDCL-type solvers contain some important ingredients such as variable selection, phase ((also called polarity) selection, restart, BCP (Boolean Constraint Propagation), conflict analysis, clause learning and its database maintenance etc. Changing any ingredient has an impact on the whole performance of solvers. This paper focuses on how to improve the following two ingredients: restart and phase selection. Recently, we proposed two new methods for optimizing a few ingredients. One is called decision-depth-sensitive restart [1], which is used to optimize the restart policy. Another is called bit-encoding phase selection [2], which is used to improve the quality of a polarity selection. Based on the two new policies, we developed a few SAT solvers. Below we describe briefly these new SAT solvers.

II. a Bit-Encoding Phase Selection Policy

In [2], we introduced a new phase selection policy called bit-encoding. The basic idea of this new policy is to let the phase at each decision level correspond to a bit value of the binary representation of an counter. Let n denote the value of a counter, and the binary representation of n be

$$n = b_k 2^k + b_{k-1} 2^{k-1} + \cdots + b_1 2 + b_0.$$

This phase selection policy stipulates that during the m-th search period, the phase of a variable at the k-th decision level is equal to b_k. Every time a restart begins, the counter n increases by one. Based on our experimental observation, it is better to apply bit-encoding scheme on only the first 6 levels. In the detailed implementation, we use only the first 4 bits of the counter n, and let the phase of a variable at the k-th decision level correspond to the (k modulo 4)-th bit of n, where $k < 6$. When $k \geq 6$, we use the phase selection policy of Glucose [3]. Here is the C code of this phase selection.

```
// assume current decision level is k
  if(k < 6 ) polarity[var]=(n >>(k %4))&1;
  else polarity[var]=previous[var];
```

where previous[var] is used to save the previous phase and is initially set to false. It is easy to see that the phase refresh period of our policy is 16, while that of the other existing policies are actually 1. The phase refresh period can be considered as a metric to measure the diversity of a search procedure. If the phase refresh period of a solver is two or more, it is said to be diverse. Otherwise, it is said to be non-diverse or uniform. So far, all the known phase selection policies are uniform, whereas this new phase selection policy is diverse.

III. a Decision-Depth-Sensitive Restart Policy

Here we introduece a new notion called DDD(decision depth decreasing). It is related to the Longest Decreasing Subsequence (LDS). LDS may be defined as follows. Given a sequence S, LDS(S) is the longest decreasing subsequence with the following property: (1) it contains the first term of S; (2) each term is strictly smaller than the one preceding it. For example, assuming S={7, 11, 10, 9, 5, 6, 2}, then LDS(S)={7, 5, 2}. In our solver, S is seen as a sequence of conflict decision levels. The DDD of S is defined as the number of terms in LDS(S), that is, DDD(S)=|LDS(S)|. For the above example, DDD(S)=3. The larger the DDD value is, the closer the goal is likely achieved. However, in many cases, DDD=1. To get the larger DDD, we need to produce many more conflicts. This is harmful to UNSAT instances. Hence, for the restarts that are not postponed by Glucose blocking strategy, we do not apply the DDD blocking strategy. For the restart postponed by Glucose, if DDD<2, even if the restart triggering condition is true, we continue to postpone that restart. That is, our postponing interval is not smaller than that of Glucose 2.1.

Another measure related to our new blocking strategy is the average of maximal depths (denoted by AveMax_D), which may be defined as follows.

$$\text{AveMax\_D} = \frac{1}{8}\sum_{i=2}^{9} \max\{\text{conflict depths in } i\text{-th restart interval}\}$$

the reason why the above formula removes the first maximum is because the first maximum has a greater deviation from its subsequent ones in many instances. In general, on the instances with small AveMax_D, say AveMax_D < 250, we do not apply any blocking strategy. On the instances with large AveMax_D, say AveMax_D > 1500, we remove the DDD blocking strategy.

In addition to the restart triggering condition of Glucose 2.1, we embed such additional conditions as the AveMax_D test and the DDD blocking test. Here is the C++ code of the new restart triggering strategy.

```
K=AveMax_D < 250 && freeVars > 2500 ? 0.82 : 0.8;
assume learnt clause is to c;
sumLBD+= c.lbd(); conflicts++;
queueLBD.push(c.lbd());
if(queueLBD.isFull() && queueLBD.avg()*K > sumLBD/conflicts)
   if(AveMax_D < 250 || AveMax_D > 1500 || !blocked || DDD > 1) {
        queueLBD.clear();
        restart();
  }
```

To remove the postponing strategy on some instances, we add the parameter AveMax_D to our postponing algorithm. Here is C++ code for the new postponing algorithm.

```
R=AveMax_D ≥ 250 && AveMax_D ≤ 900 && conflicts < 1500000 ?
1.38 : 1.4;
if (AveMax_D ≥ 250 && freeVars > 5000 ){
    queueTrail.push(trail.size());
    if(queueLBD.isFull() && queueTrail.isFull() &&
      trail.size() > R*queueTrail.avg()) {
        queueLBD.clear();
        blocked=true;
    }
}
```

IV. System Description of SAT Solvers

Using two new technologies mentioned above, we developed a few new SAT solvers. Below we describe briefly them.

A. glue_bit

glue_bit is built on top of Glucose 2.1, but incorporates two new policies given in previous two sections. It is a sequential single-engine CDCL SAT solver, which runs SatElite as a preprocessor. In glue_bit, the bit-encoding phase selection policy is used to enhance the ability of solving UNSAT instances, whereas the decision-depth-sensitive restart policy is used to enhance the ability of solving SAT instances. For big instances, glue_bit uses still the same as the solving strategies of Glucose 2.1. This solver is submitted to the sequential, application SAT+UNSAT and SAT track of the SAT Competition 2013.

B. minisat_bit{_u}

Minisat_bit{_u} is a hack version of MiniSAT [5]. Except for the pickBranchLit procedure, Minisat_bit is the same as MiniSAT 2.2.0. In the pickBranchLit procedure, Minisat_bit{_u} adds the bit-encoding phase selection policy mentioned above. The phase selection policy of Minisat_bit{_u} is a little bit different from that of glue_bit. In glue_bit, the decision level applying the bit-encoding scheme is limited to 6, while in minisat_bit, the decision level applying the bit-encoding scheme is limited to 12. Furthermore, minisat_bit{_u} has a bit-encoding sub-scheme. Every 4 levels corresponds to a bit-encoding sub-scheme. When the decision level is greater than 12, We use the same phase selection policy as MiniSAT to select a polarity of decision variables. Minisat_bit is submitted to the sequential, MiniSAT hack-track and application of the SAT Competition 2013. Minisat_bit_u is submitted to certified UNSAT track.

C. gluebit_lgl

gluebit_lgl can be regarded as a hybrid solver or an interacting solver using two SAT solving engines. It combines glue_bit and Lingeling 587 [4] that participated in SAT Competition 2011. Its main framework is based on glue_bit. For big instances, this solver switches to glue_bit to solve them. For other instances, glue_bit and Lingeling run specified search steps in turn, and exchanges intermediate results each other. If a solver performs better than another solver, the subsequent solving will be done by that solver with the better performance. This solver is submitted to the sequential, application SAT+UNSAT and SAT track of the SAT Competition 2013.

D. gluebit_clasp

gluebit_clasp is a hybrid solver combining glue_bit and clasp 2.0-R4092 (Gold Non-portfolio in SAT Competition 2011 crafted track). At the initial stage, we use glue_bit to solve an instance. This is similar to the role of a preprocessor. Once glue_bit has found the instance not suitable for it, it aborts to solve and is switched to clasp. In some cases, clasp is switched also to glue_bit. This solver is submitted to sequential, hard-combinatorial SAT+UNSAT and SAT track of the SAT Competition 2013.

V. Conclusion

All the SAT solvers given in this paper are based on two new policies. For glue_bit, we conducted sufficient experiments, while for the other new solvers, we did only a few experiments. The results from the experiment on glue_bit show that the performance of glue_bit was surprisingly good. We believe that the other solvers containing glue_bit will perform well.

References

[1] Chen, J.C.: Decision-Depth-Sensitive Restart Policies for SAT Solvers, submitted for publication, 2013.

[2] Chen, J.C.: A Bit-Encoding Phase Selection Strategy for Satisfiability Solvers, submitted for publication, 2013.

[3] Audemard, G., Simon, L.: Refining Restarts Strategies for SAT and UNSAT, 18th International Conference on Principles and Practice of Constraint Programming (CP'12),pp. 118C-126 (2012)

[4] Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010,

[5] Sörensson, N., Biere, A.: Minimizing Learned Clauses, SAT 2009, LNCS 5584, pp. 237–243 (2009)