Look-Ahead vs. Look-Back Techniques in a Modern SAT Solver

Enrico Giunchiglia, Marco Maratea, Armando Tacchella

{enrico, marco, tac}@dist.unige.it

MRG-LAB DIST, Università di Genova

DLL algorithm

function DLL-Solve()

  do

    $r \leftarrow$ *LookAhead*()

    if $r =$ T **then**

      $r \leftarrow$ *ChooseLiteral*()

    else

      $r \leftarrow$ *LookBack*()

  while $r =$ U

  return r

The concept of modern SAT solver

The key aspects of modern SAT solvers (as zChaff, Berkmin, Limmat, Simo) are:

- efficient data structures (e.g. watched literals)
- an innovative heuristic, based on the information extracted from the look-back phase
- an innovative look-back method (e.g. UIP-based learning)
- low-level optimizations of the code

Failed literal detection

- Was introduced in Posit
- Is used in Satz and Relsat, in combination with the heuristic
- It reached good results in different domains, in particular on random problems
- A similar technique is used in 2clseq (HypBinRes)

Failed literal detection: the question

Is failed literal detection an effective look-ahead technique on modern SAT solver?

In order to answer this question, we introduce 2 versions of Simo:

- **Simo-Up**, i.e., Simo in its default configuration
- **Simo-Fp**, i.e., Simo enhanced with failed literal detection in the look-ahead phase

Industrials benchmarks: Simo-Up vs. Simo-Fp

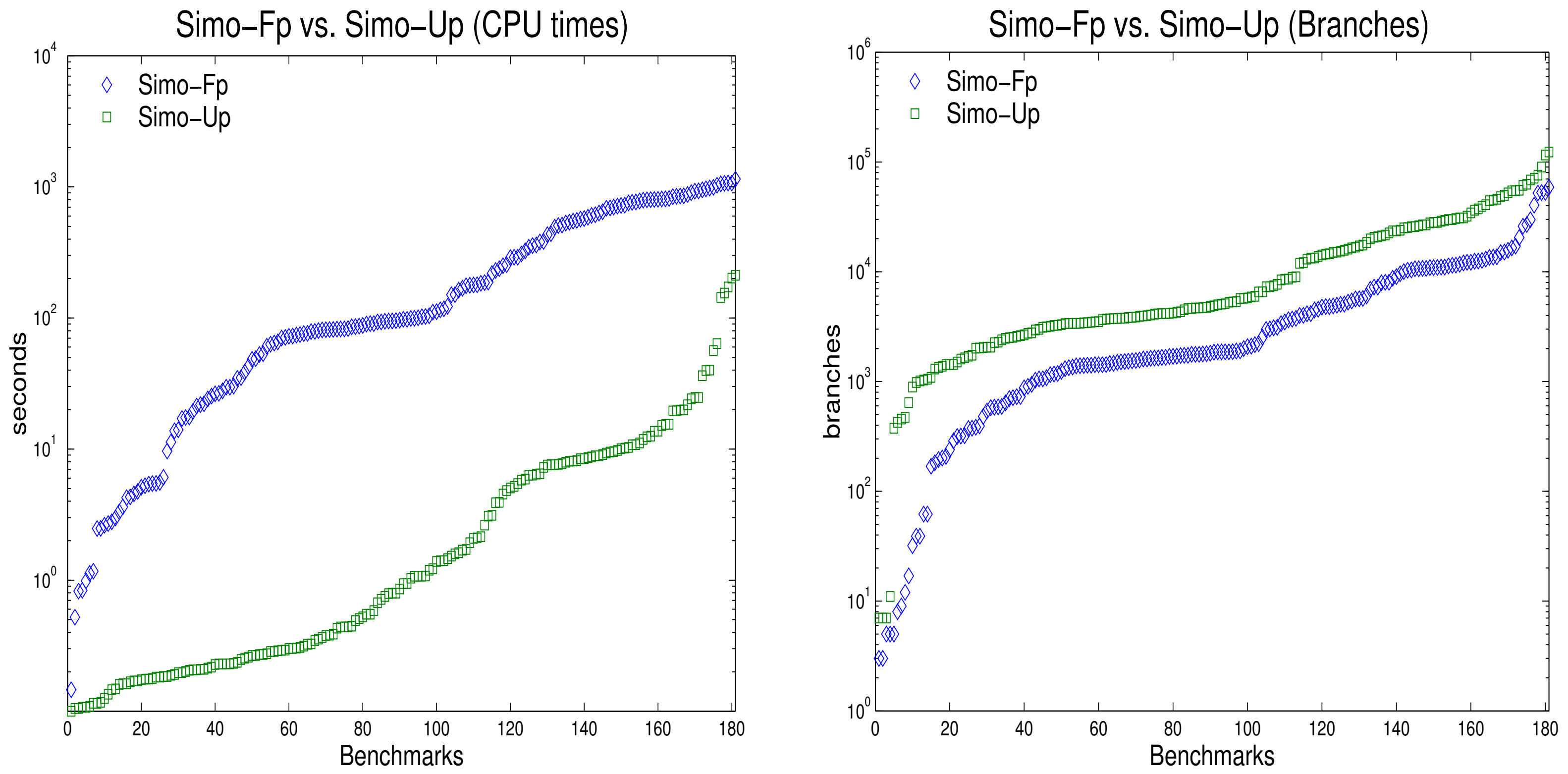

Figura 1: 483 industrials benchmarks.

Failed literal detection: two more detailed questions

From the experimental analysis showed, two questions follow:

1. Does there exist a way to make it effective?

2. Are the results due to specific implementation inefficiencies?

We restrict our attention to “complete” failed literal detection algorithms

Failed literal detection: two more detailed questions

From the experimental analysis showed, two questions follow:

1. Does there exist a way to make it effective?
2. Are the results due to specific implementation inefficiencies?

We restrict our attention to “complete” failed literal detection algorithms

We will see that the answer to both questions in NO

Agenda

- The oracles
- Answering question 1.
- Answering question 2.
- Conclusions and future work

Answering the questions: Introducing oracles in Simo-Fp

In order to answer the questions above, we introduce three oracles-based versions of Simo-Fp, and we assume to have:

- in Simo-Fp(TO), an oracle testing whether a literal will fail in Simo-Fp, thus saving the time necessary to try the literals which will not be failed
- in Simo-Fp(FO), an oracle returning the sequence of literals which will fail in Simo-Fp, thus saving also the time necessary to scan the list of open literals
- in Simo-Fp(FRO), an oracle returning the sequence of the literals which will fail in Simo-Fp and their reasons, thus saving also the time necessary to calculate the reasons of the failed literals

Answering question 1.

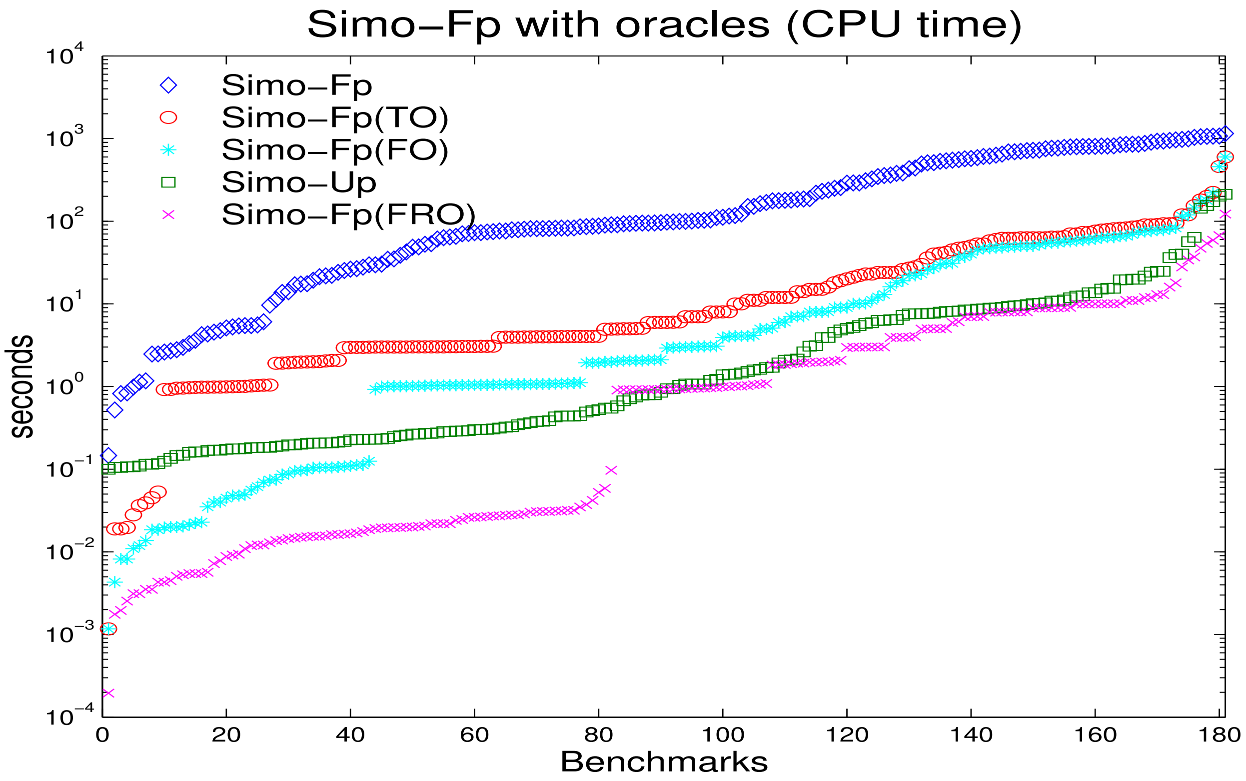

Figura 2: Simo-Up, Simo-Fp and oracles-based versions of Simo-Fp, considering CPU time.

Answering the questions: Introducing Tries as measure

In the following, we use tries as CPU independent performance measure, instead of branches.

A trie happen each time a literal is assigned a value for whatever reason (choice, unit literal, failed literal, tentative failed literal in the failed literal detection phase).

Moreover:

- Clearly, the number of branches is always less than (or equal to) the number of tries
- Most of the overall run time of the solver is spent on assigning literals
- (Number of) tries is a better measure of the dimension of the search tree explored

Answering question 2.

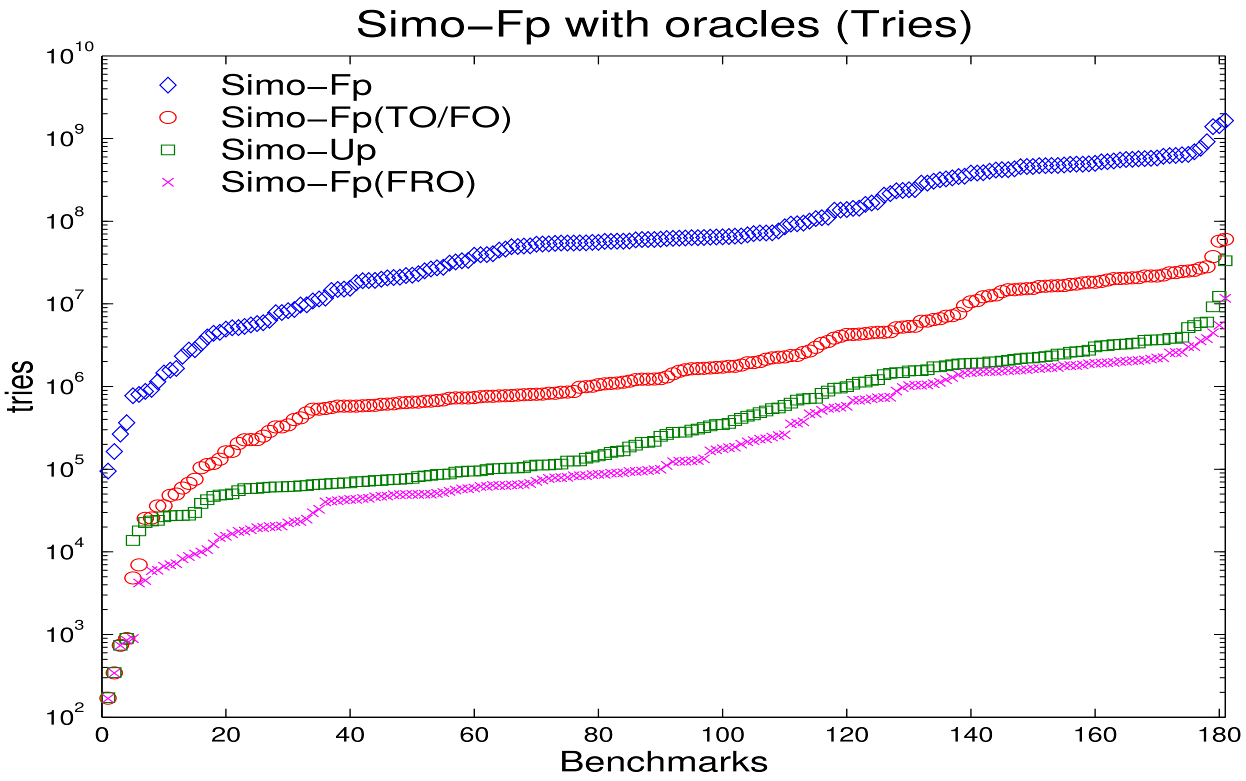

Figura 3: Simo-Up, Simo-Fp and oracles-based versions of Simo-Fp, considering Tries.

No compensation effects

Benchmarks					Simo Tries (x1000)			
Family	Sat	Tot	At#	Cl#	Up	Fp	Fp(FRO)	Fp(TO)
Beijing-1996	8	8	8,226	53,390	1,151	4,475,504	113	5,141
bmc	14	30	10,466	52,995	95,253	5,006,227	33,669	290,623
des	7	7	3,285	20,539	3,073	85,967	786	13,554
fev	0	3	1,324	3,819	1,636	168,740	786	2,814
fpga	10	30	32,612	194,786	10,326	960,441	9,232	48,375
fvp-unsat.2.0	0	5	1,468	15,206	8,370	1,047,241	5,438	52,900
mediator	2	2	561.50	12,086	3,689	22,472	1,289	12,833
miters	3	12	2,261	6,119	27,398	2,505,136	23,478	72,405
sss-sat.1.0	79	79	5,022	51,043	75,227	17,553,074	56,333	601,287
vliw-sat.1.1	5	5	20,780	284,509	242	2,657,969	127	8,374

Tabella 1: Tries arranged by benchmark family, note that Fp(FO) = Fp(TO).

Conclusions

- We have presented strong empirical evidence that enhanced look-ahead based on failed literal detection does not pay off in a modern SAT solver
- We have shown that this result is independent of the specific implementation of failed literal detection
- Preliminary results seem to point out that look-back techniques and look-back based heuristics account for a great part of the effectiveness of modern SAT solvers

Future work

- confirm our results with additional tests
- evaluate the (in)effectiveness of other look-ahead techniques (recursive failed literal detection, dilemma) in a modern SAT solver
- evaluate the impact of look-ahead based heuristics (using informations that arise from the look-ahead step) instead of look-back based

References

Learn more about Simo at the STAR project homepage available from:

http://www.mrg.dist.unige.it/star

Failed literal detection: the algorithm

function FailedPropagate()

  for each open atom a

    $r \leftarrow$ *UnitPropagate*(a)

    LookBack()

    if $r =$ F **then**

      $r \leftarrow$ *UnitPropagate*($\neg\, a$)

      if $r =$ F **then return** F

    else

      $r \leftarrow$ *UnitPropagate*($\neg\, a$)

      LookBack()

      if $r =$ F

        UnitPropagate(a)

  return T

Adding failed literal detection to unit propagate

The *LookAhead*() step, composed only by unit propagation in modern SAT solvers, is changed in the following way:

function LookAhead()
  $r \leftarrow$ *UnitPropagate*()
  if $r =$ F **then**
    return F
  else
    return *FailedPropagate*()

Introducing timers and counters in Simo-Fp

- Total time (resp. tries), T_f (resp. N_f), is the sum of the run times (resp. tries) of each call to *FailedLiteral*
- Time (resp. Tries) spent on failed, T_s (resp. N_s), is the sum of the run times (resp. tries) spent to perform literal propagations when the literals are failed
- Time (resp. Tries) wasted on failed, T_w (resp. N_w), the same as above, but when the literals are not failed
- Time (resp. Tries) spent on reason, T_r (resp. N_r), is the sum of the run times (resp. tries) spent to calculate the reason of each failed literal when the literal is failed

Calculating performances of oracles-based versions

The performances of oracles-based versions can be calculated as follow.

In terms of CPU time:

$$T(TO) = T - T_w$$

$$T(FO) = T - T_f + T_s$$

$$T(FRO) = T - T_f + T_s - T_r = T(FO) - T_r$$

Instead, in terms of tries:

$$N(TO) = N(FO) = N - N_w$$

$$N(FRO) = N - N_f + N_s - N_r$$

Simo-Fp(TO): the algorithm

function FailedPropagate()

  for each open atom a

    $r \leftarrow$ *UnitPropagate*(a)

    LookBack()

    if $r =$ F **then**

      $r \leftarrow$ *UnitPropagate*($\neg\, a$)

      if $r = F$ **then return** F

    else

      $r \leftarrow$ *UnitPropagate*($\neg\, a$)

      LookBack()

      if $r =$ F

        UnitPropagate(a)

  return T

Simo-Fp(FO): the algorithm

function FailedPropagate()

    for each open atom a
        $r \leftarrow$ *UnitPropagate*(a)
        LookBack()
        if $r =$ F **then**
            $r \leftarrow$ *UnitPropagate*$(\neg\, a)$
            if $r = F$ **then return** F
        else
            $r \leftarrow$ *UnitPropagate*$(\neg\, a)$
            LookBack()
            if $r =$ F
                UnitPropagate(a)
    return T

Simo-Fp(FRO): the algorithm

function FailedPropagate()

  for each open atom a

    $r \leftarrow$ *UnitPropagate*(a)

    LookBack()

    if $r =$ F **then**

      $r \leftarrow$ *UnitPropagate*($\neg\, a$)

      if $r = F$ **then return** F

    else

      $r \leftarrow$ *UnitPropagate*($\neg\, a$)

      LookBack()

      if $r =$ F

        UnitPropagate(a)

  return T