(In)Effectiveness of Look-Ahead Techniques in a Modern SAT Solver

Enrico Giunchiglia[1], Marco Maratea[1], and Armando Tacchella[1]

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
{enrico,marco,tac}@mrg.dist.unige.it

Abstract. In this paper we investigate the effect of adding a failed literal detection method to the traditional unit clause propagation method in the look-ahead component of a modern SAT solver. Our investigation is experimental, and leads to very interesting results. Perhaps the most interesting one is that, in all the SAT instances that we have tried, failed literal detection is bound to be ineffective, even assuming it has no overhead.

1 Introduction

In the last couple of years, we have seen a tremendous boost in the performances of SAT solvers, such boost mostly due to Chaff [1]. Chaff owes its efficiency to four components: (*i*) efficient data structures, fitted to the look-ahead and look-back methods implemented, (*ii*) an innovative look-back method, (*iii*) an innovative heuristic, based on the information extracted from the look-back phase, and (*iv*) low-level optimizations of the code. zChaff (Chaff latest incarnation) was the best among the complete SAT solvers on industrial and hand-made benchmarks in the last SAT competition [2]. Further, all the other top-performers in these two categories (with the exception of 2CLS+EQ [3]) were "Chaff-like" SAT solvers, as they incorporated some/most of the ideas behind Chaff. In this paper, when we speak of modern SAT solvers, we have in mind a "Chaff-like" engine. It is worth pointing out that the only look-ahead method implemented by most modern SAT solvers is still the simple, yet effective, unit clause propagation method, but they all feature far more sophisticated heuristics and look-back methods.

In this paper we investigate the effect of adding a failed literal detection method to the traditional unit clause propagation method in the look-ahead component of a modern SAT solver. Failed literal detection was first introduced in POSIT [4], used extensively, e.g., in SATZ [5] and RelSAT [6], and similar techniques led to positive results on real-world instances in [3]. Our analysis is mostly experimental and has been performed using our solver Simo modified to incorporate ideas (*i*) - (*iii*) above. With these modifications Simo is within a factor of 3 slower than zChaff, the remaining gap due to the absence of low-level code optimizations. We have run our experiments using several challenging real-world benchmarks. Such instances are either well-known in the literature or submitted to the last SAT competition. On the basis of the collected data, we conclude that:

– enhanced look-ahead based on failed literal detection does not pay off in modern SAT solvers;
– even assuming we had an oracle answering whether a literal will fail (thus saving the time necessary to try the literals which will not fail), enhanced look-ahead is not effective;
– even assuming we had an oracle giving us the list of literals which will fail (thus saving the time necessary to scan the list of literals), enhanced look-ahead is not effective.

We reached such conclusions by analyzing CPU times, but also, and perhaps most convincingly, on the basis of CPU independent data such as the total number of assignments performed by Simo. Therefore, enhancing the look-ahead method with powerful deduction techniques seems to be hardly successful, at least in the case of failed literal detection and real-world instances.

Excluding the Conclusions, the paper consists of just two more sections. In Sec. 2, we briefly describe the internals of Simo, the contents of our test set, and show how, at a first glance, failed literal detection might look as a promising technique. In Sec. 3, we introduce a family of oracles, and we describe how to simulate the execution of oracle-based algorithms by plugging performance counters into Simo. Finally, we analyze the results obtained by Simo in various configurations, to draw conclusions about the effectiveness of the various algorithms, both real and oracle-based. Throughout the paper, we will assume that the reader is familiar with the topics of Boolean satisfiability and satisfiability search algorithms for Boolean formulas in conjunctive normal form (for an extensive coverage of these topics see, e.g., [7]).

2 Experimenting with look-ahead techniques

In this section we introduce and compare two versions of Simo: Simo-Up, i.e., Simo in its default configuration, and Simo-Fp, i.e., Simo enhanced with failed literal detection. In Sub. 2.1 we briefly describe the implementation of Simo, the differences between Simo-Up and Simo-Fp, and the failed literal detection algorithm. In Sub. 2.2 we describe the benchmarks selected for our analysis and the computing platform on which the experiments have been carried out. Finally, in Sub. 2.3 we show experimental data indicating that the search space explored by Simo-Fp appears to be smaller than the one explored by Simo-Up, but Simo-Fp performance is much worse than Simo-Up.

2.1 Simo: a modern SAT solver

In this section we give an introductory overview of Simo. Particular emphasis is given to the look-ahead engine, since this is the only part in which Simo-Fp differs with respect to Simo-Up. The conventions we use to describe algorithms are those of [8] (pages 4,5).

In Fig. 1 we describe three basic components of Simo. The main search loop of Simo is DLL-Solve, Fig. 1 (top-left), an implementation of the DLL method [9]. Inside DLL-Solve:

```
DLL-Solve()
1 do
2   r ← Look-ahead()
3   if r = T then
4     r ← Heuristic()
5   else
6     r ← Look-back()
7 while r = U
8 return r

Look-ahead()
1 r ← Unit-propagate(nil)
2 if r = F then
3   return F
4 else
5   return Failed-propagate()
```

```
Failed-propagate()
1 for each open atom a
2   r ← Unit-propagate(a)
3   Look-back()
4   if r = F then
5     r ← Unit-propagate( ¬a)
6     if r = F then return F
7   else
8     r ← Unit-propagate( ¬a)
9     Look-back()
10    if r = F
11      Unit-propagate( a)
12 return T
```

Fig. 1. Overview of Simo.

- Look-ahead simplifies the input formula and returns F if an empty clause is deduced, T otherwise;
- Heuristic assigns an open literal according to some criterion and returns T in case no open literal can be found, U otherwise;
- Look-back backtracks to a previous open branch in the search tree, and returns F if such branch cannot be found, U otherwise.

Look-ahead, Fig. 1 (bottom-left), is where the distinction between Simo-Up and Simo-Fp comes into play: in Simo-Up, lines 2-5 of Look-ahead are replaced with the instruction "**return** r", i.e., Simo-Up simplifies the formula using unit clause propagation only; in Simo-Fp, the implementation of Look-ahead is exactly as detailed in Fig. 1. Failed-propagate, Fig. 1 (right), scans all the open atoms of the formula (line 1). For each such atom a, it tentatively assigns a to T and performs unit clause propagation (line 2), and then it retracts the tentative assignment (line 3). If Unit-propagate deduces an empty clause, i.e., the literal a is failed, then Failed-propagate tries to assign a to F (line 5): if even this fails, then it returns F i.e., it forces backtracking (line 6). If propagation of a succeeds, then Failed-propagate repeats the same process for $\neg a$ (lines 8-11). Notice that (line 11), in case $\neg a$ is failed, Failed-propagate already knows that a is not, and so there is no need to check the result of Unit-propagate. If there was no atom a such that both a and $\neg a$ failed, Failed-propagate returns T.

The functions Unit-propagate, Heuristic and Look-back called by the algorithms in Fig. 1 are implemented as described in [1] and [10]. In more details:

Unit-propagate is implemented as described in [1], Sec. 2, "Optimized BCP". The function is based on a data structure that, for each clause, keeps reference to two literals in the clause ("watched literals" in the terminology of [1]) with the proviso that they should not be both assigned to F. This data structure has several features: (*i*) it identifies all unit and empty clauses, (*ii*) it minimizes the number of accesses to clauses, and (*iii*) it does not need expensive updates while backtracking.

HEURISTIC shares many similarities with the heuristic described in [1]. Literal scores are updated only once each N calls to HEURISTIC (N is set by default in Simo to 255); once the scores have been updated, the literals are stored in a priority queue; at each branch, the highest open literal in the priority queue is selected. The rules to update the literal score are:
- each literal has a counter, initialized to the number of occurrences in the formula;
- when a clause is learned during the look-back process the counter associated with each literal in the clause is incremented;
- the score of a literal is updated by considering its old score (divided by a constant) plus the increment in the value of its counter since the last scoring occurred.

LOOK-BACK follows the learning method described in Sec. 2 of [10] as "1-UIP learning". The implementation of this look-back method in Simo involves some hairy details which are beyond the scope of this paper. Please see [10] for a detailed explanation of 1-UIP (and related) learning methods.

Considering the implementations of LOOK-AHEAD, HEURISTIC and LOOK-BACK described above, Simo duplicates very closely the behavior of Chaff: as we mentioned in Sec. 1, Simo is within a factor of 3 slower than Chaff while, considering the number of assignments performed, the two solvers are more than 80% correlated, and neither of the two dominates the other.

2.2 Experimental Setting

The test set that we used for our experimental analysis consists of 483 real world instances. The benchmarks have been selected considering classical SAT problems as well as instances submitted to the last SAT competition [2]. The benchmarks are divided into 10 families:

Beijing'96 16 instances from the Beijing'96 SAT competition, a mixture of scheduling [11], planning and circuit verification problems;

bmc 78 instances including
- 30 instances from CMU described in [12],
- 13 instances from IBM, described in [13], and
- 22 "dining philosophers" and 6 "counter" model checking instances described in [2];

des 28 instances of logic cryptanalysis for security protocols described in [14];

fev 8 instances, encoding the problem of comparing the output of different adder circuit implementations (see [2]);

fpga 102 instances of FPGA routing problems, including "homer" and "bart" instances described in [15];

fvp-unsat 2.0 22 instances of formal property verification problems;
mediator 4 instances of query optimization problems in complex mediator systems, described in [16];
miters 25 instances encoding problems of Combinational Equivalence Checking described, e.g., in [17];
sss-sat 1.0 100 instances related to formal property verification of superscalar microprocessors;
vliw-sat 1.0 100 instances related to formal property verification of VLIW microprocessor.

The three families fvp-unsat 2.0, sss-sat 1.0 and vliw-sat 1.0 are collectively described in [18]. The size of the problems varies considerably, from a minimum of 25 atoms and 60 clauses (an instance in the fev family), to a maximum of 315,312 atoms and 1,725,384 clauses (an instance in the fpga family), with an overall average of 11,520 atoms and 108,229 clauses. To confirm that the test set is indeed challenging for modern SAT solvers, we used zChaff [1], the winner of the latest SAT competition on industrial and hand made categories (see [2]). By running zChaff on the test set we obtained the following results:

- 438 problems can be solved within a time limit of 20 minutes;
- 303 instances are found satisfiable, 135 unsatisfiable, and the remaining timed out;
- the total run time (excluding timed out instances) is about 5 and a half hours, with an average run time of about 50 seconds.

All the experiments presented in this paper, including the above exploratory analysis, have been run on two identical PCs equipped with a Pentium IV 1.8 Ghz processor, and 512MB of RAM (DDR 266Mhz), running Linux RedHat 8.0 (kernel version 2.4.18-14). Since this platform cannot normally measure CPU times below the threshold of $10^4 \mu$sec, to overcome this limitation we patched the kernel with the tool perfctr contained in the distribution of the package papi 2.3.1 [19]. We used a binary distribution of zChaff 2001.2.17 for the exploratory analysis, and our solver Simo 3.0 to run all the experiments presented. Simo has been compiled with gcc 3.2. For practical reasons, a timeout of 1200 CPU seconds (20 minutes) has been placed on the run time of Simo on each instance.

2.3 Simo-Up vs. Simo-Fp

Using the algorithms presented in Sub. 2.1 and the test set presented in Sub. 2.2 we run the comparison between Simo-Up and Simo-Fp. For both versions of Simo, the distributions of the run time and the number of of calls to Heuristic (branches), are summarized in Fig. 2. The x-axis in both plots is an ordinal in the range (0-180), and the y-axis is, respectively, CPU seconds in Fig. 2 (left) and number of branches in Fig. 2 (right). The total number of problems visualized is 181 out of 483 since we discarded

- instances in which either Simo-Up or Simo-Fp exceeded the time out, and

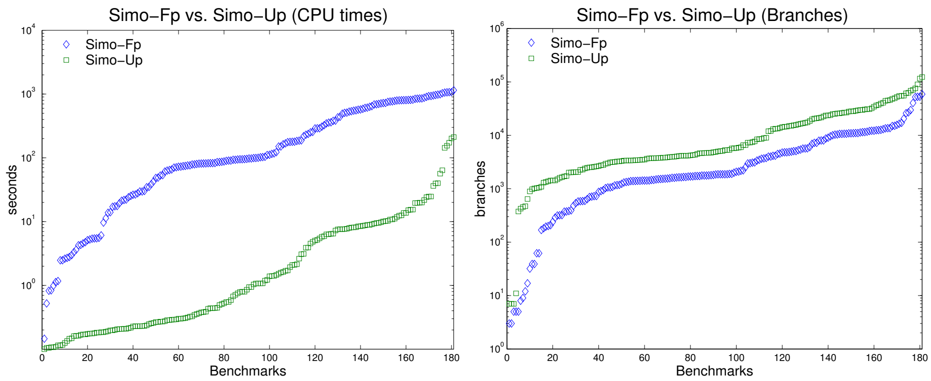

Fig. 2. Simo-Up vs. Simo-Fp considering CPU time (left) and Branches (right).

– instances in which the run time of Simo-Up was less than 0.1 seconds.

Both plots in Fig. 2 are obtained by ordering the results of Simo-Up and Simo-Fp independently and in ascending order. Therefore the middle point in each plot is approximately the 50% percentile (median) of the distributions, the end points are, respectively, the 0% (minimum) and the 100% (maximum) value, and other percentile values can be spotted similarly.

As you can see from Fig. 2, the performance of Simo-Fp in terms of CPU time is clearly worse than the performance of Simo-Up, also considering that Simo-Up solves another 161 instances (not shown in Fig. 2) on which Simo-Fp times out. On the other hand, if we consider the number of branches, then the search space explored by Simo-Fp appears to be always, although not considerably, smaller than the one explored by Simo-Up. At first glance, this seems to point out a specific inefficiency in the implementation of FAILED-PROPAGATE in Simo-Fp. As we will show in the next section, this is indeed not the case.

3 (In)Effectiveness of failed literal detection

In this section we will motivate the experimental results shown in the previous one, and we will give strong empirical reasons against the effectiveness of failed literal detection in modern SAT solvers like Simo. To accomplish this, we introduce a family of oracle-based algorithms, where an oracle is an ideal procedure that performs some computation for free (see, e.g., [20]). In particular, we will show how the CPU time of algorithms incorporating oracles can be calculated starting from the CPU time of Simo-Fp, and then we compare the performances of such oracle-based algorithms with Simo-Up.

In the following, we use *tries* as a CPU independent performance measure, instead of branches. The number of tries is the number of times that a literal is assigned a value, for whatever reason, be it a choice of the heuristic, a unit literal, a failed literal, or a tentative assignment performed during FAILED-PROPAGATE. Clearly, the number of branches is always less than (or equal to) the number of tries. As we will see, tries are more informative than branches as a CPU independent performance measure, because most of the overall run time of the solver is spent on assigning literals (as also reported in [1]).

3.1 Introducing oracles in Simo-Fp

As we have seen in Sub. 2.1, Simo-Fp scans the list of open literals and propagates each open literal to see whether it fails. We can think of three oracle-based versions of Simo-Fp, that we call Simo-Fp(TO), Simo-Fp(FO), and Simo-Fp(FRO). In particular, we assume to have

- in Simo-Fp(TO), an oracle testing whether a literal will fail, 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 literals which will fail in Simo-Fp and their reasons, thus saving also the time necessary to calculate the reasons of the failed literals.

Since the oracles cannot be implemented in practice, we need to calculate the performance of the oracle-based versions using the experimental data of Simo-Fp. In order to accomplish this, we introduce four CPU time counters inside FAILED-PROPAGATE (line numbers refer to FAILED-PROPAGATE in Fig. 1):

- Total time (T_f) is the sum of the run times of each call to FAILED-PROPAGATE.
- Time spent on failed (T_s) is the sum of the run times spent to perform literal propagations when the literals are failed (lines 2-5 and lines 8-11, when the tests on lines 4 and 10 are successful, respectively).
- Time wasted on failed (T_w) is the same as above, but when the literals are not failed (lines 2-3 and lines 8-9, when the tests on lines 4 and 10 are not successful, respectively).
- Time spent on reason (T_r) is the sum of the run times spent to calculate the reason of each failed literal when the literal is failed (lines 2-3 and lines 8-9 when the tests on lines 4 and 10 are successful, respectively).

Let T be the CPU time of Simo-Fp, and $T(*)$ be the CPU time of Simo-Fp(*). The performance of the oracle-based algorithms can be calculated as follows:

$$
\begin{aligned}
T(\mathsf{TO}) &= T - T_w \\
T(\mathsf{FO}) &= T - T_f + T_s \\
T(\mathsf{FRO}) &= T - T_f + T_s - T_r
\end{aligned}
$$

Notice that $T(\mathsf{FRO}) = T(\mathsf{FO}) - T_r$.

To calculate the tries of the oracle-based versions of Simo-Fp we introduce four more counters (still with reference to FAILED-PROPAGATE in Fig. 1):

- Total tries (N_f) is the sum of the tries performed in each call to FAILED-PROPAGATE.
- Tries spent on failed (N_s) is the sum of the tries spent to perform literal propagations when the literals are failed (the tries performed by UNIT-PROPAGATE in lines 2,5 or 8,11 when the literal is failed).
- Tries wasted on failed (N_w) is the same as above, but when the literals are not failed (the tries performed by UNIT-PROPAGATE in lines 2 or 8 when the literal is not failed).
- Tries spent on reason (N_r) is the sum of the tries spent to calculate the reason of each failed literal when the literal is failed (the tries performed by UNIT-PROPAGATE in lines 2 or 8 when the literal is failed).

Let N be the number of tries performed by Simo-Fp, and $N(*)$ the number of tries performed by Simo-Fp(*). The number of tries performed by the oracle-based algorithms can be calculated as follows:

$$N(\mathsf{TO/FO}) = N - N_w$$
$$N(\mathsf{FRO}) = N - N_f + N_s - N_r$$

Notice that $N(\mathsf{TO}) = N(\mathsf{FO})$, so we do not need two distinct measures for the tries of Simo-Fp(TO) and Simo-Fp(FO).

3.2 Simo-Up vs. Simo-Fp(*)

Considering the oracles presented in the previous subsection, we can enrich the comparison between Simo-Up and Simo-Fp with the data of Simo-Fp(TO), Simo-Fp(FO), and Simo-Fp(FRO). For all such real and oracle-based versions of Simo, the distributions of the run time and the number of tries are summarized in Fig. 3. The layout of the plots, as well as the selection of the data to show, follows the same guidelines of Fig. 2. By looking at Fig. 3 we can immediately conclude that aggressive failed literal detection is bound to be ineffective, both in terms of run time and, more interestingly, also in terms of search space explored. The only version of Simo-Fp that can barely compete with Simo-Up is Simo-Fp(FRO), the version of Simo-Fp embodying the most powerful oracle presented in Sub. 3.1. In spite of its power, the number of tries performed by Simo-Fp(FRO) is, on average, only about 80% of the number of tries performed by Simo-Up. As we can deduce from the plots, Simo-Fp performances are influenced by two major factors:

(i) the time (and the tries) spent to check whether a given literal is failed or not, and
(ii) the time (and the tries) spent to calculate the reasons of failed literals.

By looking at Fig. 3 (right) we can see two order-of-magnitude gaps in the number of tries: one between Simo-Fp and Simo-Fp(TO/FO), which confirms point (i), and one between Simo-Fp(TO/FO) and Simo-Fp(FRO), which confirms point (ii).

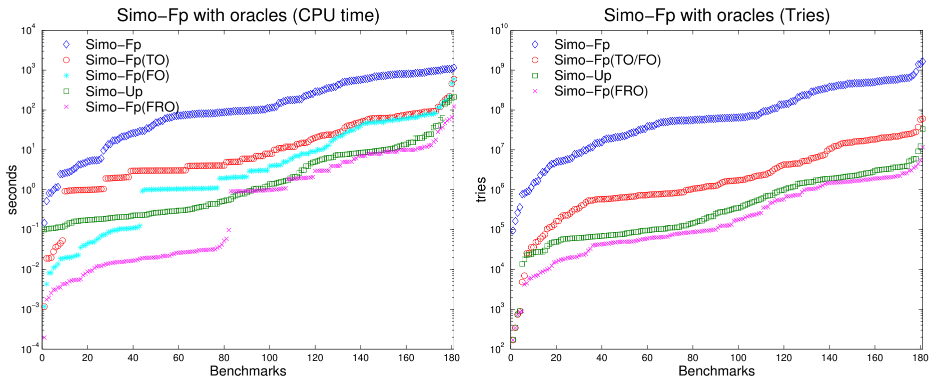

Fig. 3. Simo-Up, Simo-Fp and oracle-based versions of Simo-Fp, considering CPU time (left) and Tries (right).

In the following we concentrate on tries as a complexity measure. The explanation for the performance gap between Simo-Fp and Simo-Fp(TO/FO) lies in the fact that FAILED-PROPAGATE, on a formula with N atoms, performs $O(N^2)$ tests along each path of the search tree. Moreover, each test involves a round of unit propagation which performs $O(N)$ tries. The total complexity of failed propagation along a single path is therefore $O(N^3)$ tries. Considering that the percentage of tests that yield a failed literal propagation is always less than 5% and about 0.2% on average, the gap between Simo-Fp and Simo-Fp(TO/FO) grows as $O(N^3)$. The reason of the performance gap between Simo-Fp(TO/FO) and Simo-Fp(FRO) is explained by the fact that in order to calculate the reason of a failed literal, Simo-Fp(TO/FO) has to pursue the "wrong" branch till it generates a contradiction. This involves one assignment plus several unit propagations: in our test set, such propagations cost about 160 tries on average.

To complete our experimental analysis, we need to confirm that the cumulative results of Fig. 3 are true also of each single family, i.e., there are no compensation effects among different families of benchmarks. In Table 1 we present the data regarding Simo-Up, Simo-Fp and the oracle-based versions of Simo-Fp. Each row of the Table contains data about a single family among those detailed in Sub. 2.2. For each family we report: the number of benchmarks left in the family after instances have been filtered out as described in Sub. 2.3 (column Tot); the number of satisfiable instances (column Sat); the average number of atoms and clauses (columns At# and Cl#, respectively); the total number of tries performed by Simo-Up, Simo-Fp, Simo-Fp(TO/FO) and Simo-Fp(FRO) divided by 1,000. Notice that clauses and atoms statistics have been rounded to 1, and tries statistics have been rounded to 1,000. Although we cannot show here the complete data, compensation effects are absent also when looking at single instances in each family. In other words, there is no single instance in our test set

Benchmarks					Simo Tries (x1000)			
Family	Sat	Tot	At#	Cl#	Up	Fp	Fp(FRO)	Fp(TO/FO)
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	8370	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

Table 1. Simo-Up, Simo-Fp and oracle-based versions of Simo-Fp, considering Tries arranged by benchmark family.

on which Simo-Fp(TO/FO) performs less tries than Simo-Up or Simo-Fp(FRO), while on all the instances Simo-Fp(FRO) performs less tries than Simo-Up.

3.3 Ineffectiveness of failed literal in a modern SAT solver

All the points we made so far might explain some of the deficiencies of failed literal propagation, but they still do not explain why Simo-Fp(TO/FO) loses the comparison with Simo-Up. Considering the implementation of Simo, as shown in Fig. 1, it is easy to see that the solver is parametric on UNIT-PROPAGATE, HEURISTIC and LOOK-BACK. In particular, while the efficient implementation of UNIT-PROPAGATE is not likely to interfere negatively with the performances of Simo-Fp, the same is not true for HEURISTIC and LOOK-BACK. Therefore, in order to understand the impact of HEURISTIC and LOOK-BACK we introduce four more versions of Simo:

Simo(R)-Up and Simo(R)-Fp are obtained, respectively, from Simo-Up and Simo-Fp by substituting the Chaff-like heuristic described in Sub. 2.1 with a random one; in Simo(R)-Up, Simo(R)-Fp and its variants Simo(R)-Fp(*) HEURISTIC chooses a literal uniformly at random from the set of open literals at each branch.

Simo(CR)-Up and Simo(CR)-Fp are obtained from Simo(R)-Up by substituting the advanced learning technique mentioned in Sub 2.1 with a straightforward chronological backtracking routine; in Simo(CR)-Up, Simo(CR)-Fp and its variants Simo(CR)-Fp(*) LOOK-BACK always backtracks to the latest open branch node, irrespective of whether the branch caused the conflict or not, and learning is disabled, i.e., no clauses are ever added to the initial set.

Further, we selected about one hundred instances from the initial test set of 483 described in Sub. 2.2 by considering all the benchmarks that Simo-Fp solved in less than 10 seconds. The resulting test set contains about 50% of satisfiable

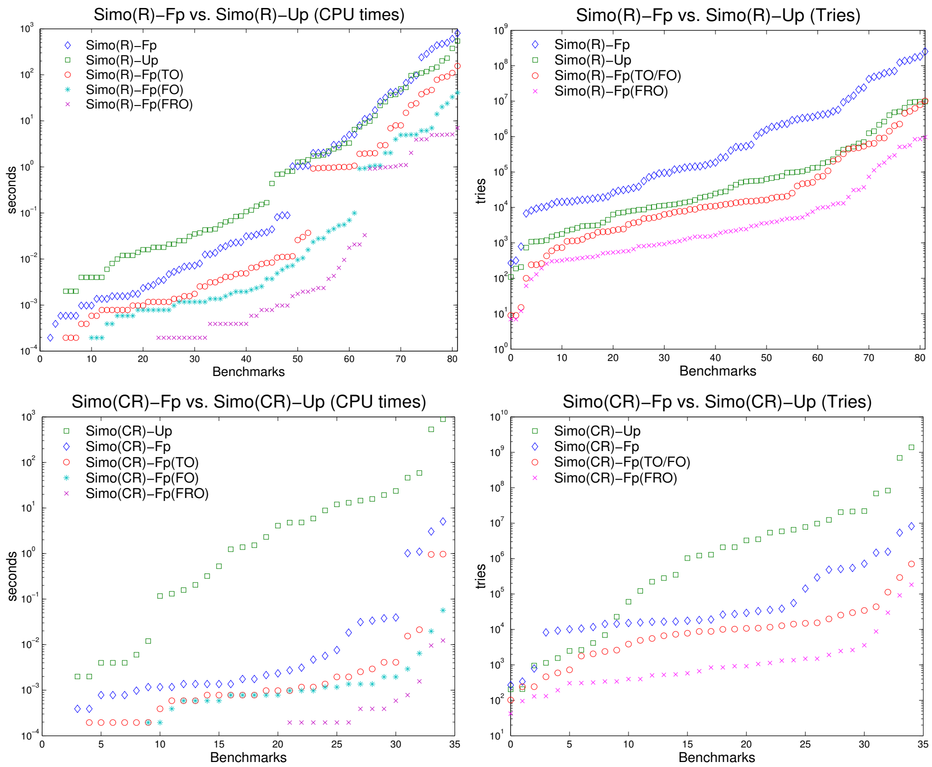

Fig. 4. Simo(R)-Up, Simo(R)-Fp and oracle-based versions of Simo(R)-Fp (top); Simo(CR)-Up, Simo(CR)-Fp and oracle-based versions of Simo(CR)-Fp (bottom); CPU times (left) and Tries (right).

formulas, and about 10% of formulas that could not be solved by the fastest of the four versions of Simo described above. The results presented hereafter are still preliminary in the sense that we did not run multiple instances of the randomized algorithms. The very lengthy tests with 100 runs for each instance and for each randomized solver are currently running.

In Fig. 4 we present the results obtained by running Simo(R)-Up, Simo(R)-Fp, Simo(CR)-Up and Simo(CR)-Fp on the reduced test set. The results are arranged into four plots, where each set of dots represents the runtime (resp., tries) distribution of a particular version of Simo. Considering Fig. 4 (top row) we can observe that Simo(R)-Up is still superior to Simo(R)-Fp: on average, Simo(R)-Fp performs one order of magnitude tries more than Simo(R)-Up, and it is barely competitive in terms of CPU. To this, we must add that Simo(R)-Fp times out on 15 instances, while Simo(R)-Up can solve an additional 8. On the other hand, Simo(R)-Fp(TO/FO) is very close to the performances of Simo(R)-Up: Simo(R)-Fp(TO/FO) performs, on average, about 50% of the number of tries of Simo(R)-

Up, and Simo(R)-Fp(TO) is about a factor of 6 faster than Simo(R)-Up. Therefore, although the real version of Simo(R)-Fp does not show any real advantage over Simo(R)-Up, the least powerful of the oracles introduced in Sub. 3.1 is sufficient to yield a substantial improvement. Considering Fig. 4 (bottom row) we can observe that Simo(CR)-Fp dominates Simo(CR)-Up: on average, Simo(CR)-Fp performs about 50 times less tries than Simo(CR)-Up, and it is about 500 times faster. Moreover, Simo(CR)-Up times out on 64 instances, while Simo(CR)-Fp can solve an additional 30.

The preliminary results presented in Fig. 4 point out that (*i*) the look-back based heuristic and (*ii*) the advanced look-back techniques implemented in Simo-Up are the major causes of the ineffectiveness of failed literal detection. The fact that failed literal detection *per se* appears to be an effective technique is witnessed by the comparison between Simo(CR)-Up and Simo(CR)-Fp. Nevertheless, the addition of a clever look-back technique can change this result dramatically, as witnessed by the comparison between Simo(R)-Up and Simo(R)-Fp. Our current explanation of this, is that the look-back schema implemented in Simo *shadows* the effects of failed literal propagation. Intuitively, even though Simo(R)-Up may assign failed literals late in the search tree, its look-back engine will assign the negation of the failed literals and place them at the proper level in the search stack. Although we do not have yet a precise intuition about the underlying phenomena in the case of the heuristic, by contrasting the results of Simo(R)-Up and Simo(R)-Fp with those presented in Sub. 3.2, it is clear that also the implementation of HEURISTIC is playing a major role in the ineffectiveness of failed literal detection.

4 Conclusions

In this paper we have presented strong empirical evidence that enhanced look-ahead based on failed literal detection does not pay off in modern SAT solvers, at least in the case of real-world problems. In particular we showed that

- the number of tries performed by Simo-Fp is, on average, about three orders of magnitude bigger than the number of tries performed by Simo-Up;
- the number of tries performed by Simo-Fp(TO/FO) is, on average, about one order of magnitude bigger than the number of tries performed by Simo-Up;
- the number of tries performed by Simo-Fp(FRO) is, on average, only about 80% of the number of tries performed by Simo-Up.

The CPU times of Simo-Fp(*) and Simo-Up share a similar behavior.

On the basis of the preliminary results shown in Sub. 3.3 we conjecture that look-back based heuristics and advanced look-back techniques account for the ineffectiveness of Simo-Fp. The additional tests needed to confirm the conjecture are currently running.

Acknowledgments

This work is partially supported by MIUR, ASI, and by a grant from the Intel Corporation.

References

1. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of DAC*, 2001.
2. L. Simon, D. Le Berre, and E. A. Hirsch. The SAT2002 Competition.
3. F. Bacchus. Enhancing Davis Putnam with Extended Binary Clause Reasoning. In *Proc. of AAAI*. AAAI Press, 2001.
4. J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.
5. C. M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proc. of IJCAI*, pages 366–371. Morgan-Kauffmann, 1997.
6. R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc. of AAAI*, pages 203–208. AAAI Press, 1997.
7. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–153. AMS, 1997.
8. T. H. Cormen, C. E. Leiserson, and R. R. Rivest. *Introduction to Algorithms*. MIT Press, 1998.
9. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7):394–397, 1962.
10. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.
11. J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of AAAI*, volume 2, pages 1092–1097. AAAI Press/MIT Press, 1994.
12. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.
13. O. Shtrichmann. Tuning SAT checkers for Bounded Model Checking. In *Proc. of CAV*, volume 1855 of *LNCS*, pages 480–494. Springer Verlag, 2000.
14. Massacci and Marraro. Logical Cryptanalysis as a SAT Problem. *JAR: Journal of Automated Reasoning*, 24, 2000.
15. I. Markov, F. Aloul, A.Ramani, and K. Sakallah. Solving difficult SAT instances in the presence of simmetry. In *DAC*, 2002.
16. S. Prestwich. A SAT approach to query optimization in mediator systems. In *SAT 2002*, 2002.
17. J. P. M. Silva and T. Glass. Combinational Equivalence Checking Using Satisfiability and Recursive Learning. In *DATE*, 1999.
18. M. N. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. In *Proc. of DAC*, 2001.
19. Using PAPI for hardware performance monitoring on Linux systems, July, 2001. Presented at Linux Clusters: The HPC Revolution.
20. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.