Width-Based Restart Policies for Clause-Learning Satisfiability Solvers

Knot Pipatsrisawat and Adnan Darwiche

{thammakn,darwiche}@cs.ucla.edu
Computer Science Department
University of California, Los Angeles, USA

Abstract. In this paper, we present a new class of restart policies, called *width-based* policies, for modern clause-learning SAT solvers. The new policies encourage the solvers to find refutation proofs with small widths by determining restarting points based on the sizes of conflict clauses learned rather than the number of conflicts experienced by the solvers. We show that width-based restart policies can outperform traditional restart policies on some special classes of SAT problems. We then propose different ways of adjusting the width parameter of the policies. Our experiment on industrial problems shows that width-based policies are competitive with the restart policy used by many state-of-the-art solvers. Moreover, we find that the combination of these two types of restart policies yields improvements on many classes of problems.

1 Introduction

Restarting has become an essential component of modern SAT solvers since the work of Gomes *et al* [1], which pointed out a problem of backtracking algorithms on combinatorial problems. In the past, restart policies used by SAT solvers were mostly static and were based on the number of conflicts experienced by the solvers (e.g., [2–5]). The intuition behind these approaches is that conflicts indicate bad assignments. So, if the solver experiences a lot of conflicts, it might have made some bad assignments early on and restarting, together with a dynamic decision heuristic, may allow these assignments to be fixed. However, this class of approaches does not take into account the actual search behavior of the solvers and may yield a bad performance on even some easy problems.

Recently, some researchers tried to improve this idea further by studying dynamic restart policies. For example, in [6], the notion of *agility*, which approximates the diversity of the assignments recently explored by the solver, was used to prevent the solver from restarting too frequently. In [7], the authors argued that restarts should be triggered based on the number of conflicts experienced under each search branch and proposed some restart policies based on this idea.

It is well-known that modern clause-learning SAT solvers can be viewed as resolution engines, which produce refutation proofs on unsatisfiable problems [8]. From this perspective, the class of restart policies based on the number of conflicts can be viewed as a way of biasing the solvers to find short unsatisfiable proofs for unsatisfiable problems.

In this work, we utilize the notion of *proof width* [9], which can also be used to measure the quality of resolution proofs, to control restarts in clause-learning SAT solvers. In particular, we propose a new class of restart policies, called *width-based policies*. According to these policies, the solvers maintain a width limit at any moment and restart as soon as too many clauses of size greater than the limit are learned. This class of policies simply tries to encourage the solvers to find a refutation proof with a small width, which could also lead to a small proof. We demonstrate how simple width-based restart policies with constant width limits can significantly outperform policies used by state-of-the-art solvers on some families of SAT problems. Then, we propose a general algorithm for adjusting the width limits used in such policies. Finally, we evaluate several width-based policies based on this algorithm on various classes of problems.

The rest of this paper is organized as follows. We review some basic notations about resolution proofs and modern clause-learning SAT solvers in the next section. In Section 3, we review existing restart policies used by leading SAT solvers and describe width-based restart policies. In Section 4, we present the results of our empirical studies on some special classes of problems, which demonstrate the strengths of width-based policies. In Section 5, we describe a general algorithm for adjusting the width limit in width-based policies. Then, we present experimental results on industrial and crafted problems in Section 6. Finally, we discuss some related work in Section 7 and conclude in Section 8.

2 Preliminaries

In this section, we discuss some basic notions that form the basis of our later discussions. First, we review basic notations about resolution and resolution proofs. Then, we briefly describe how modern clause-learning SAT solvers work from a resolution perspective. Finally, we point out the relationship between our work and an existing SAT algorithm based on proof width.

2.1 Resolution Proofs

A *resolution* between two clauses $C_1 = (x \vee \alpha)$ and $C_2 = (\neg x \vee \beta)$ is the derivation of the clause $C = (\alpha \vee \beta)$. In this case, C is called the *resolvent* of the resolution. A resolution proof Π of clause C_k from CNF Δ is a sequence of clauses $\Pi = C_1, C_2, ..., C_k$, where each clause C_i is either in Δ or is the resolvent of some clauses preceding C_i. The *size* of Π is simply the number of clauses in it, while its *width* is the size of the largest clause in it. In this work, we are mostly interested in *refutation proofs*, which are resolution proofs of the empty clause (i.e., false) from unsatisfiable CNFs. The width of an unsatisfiable CNF is simply the smallest width of any of its refutation proofs.

2.2 Modern Clause-Learning SAT Solvers

A typical modern clause-learning SAT solver works by repeatedly making decisions and using unit resolution to derive implications. Upon a conflict, the solver

derives a conflict clause to allow unit resolution to see an implication that was missed earlier. Then, it backtracks, asserts the learned clause, and continues making decisions. This process is repeated until either a solution is found or the empty clause is derived. For a more detailed description see [10].

For example, consider the following CNF:

$$\begin{aligned}\Delta = &(\neg a \vee \neg b \vee c), (\neg a \vee \neg c \vee d), (\neg a \vee \neg c \vee e), (\neg a \vee \neg d \vee \neg e),\\ &(\neg a \vee c \vee d), (\neg a \vee c \vee e), (a \vee \neg b \vee c), (a \vee \neg b \vee \neg c),\\ &(a \vee b \vee \neg c), (a \vee b \vee e), (a \vee b \vee \neg f), (c \vee \neg e \vee f).\end{aligned}$$

We can view the execution of a clause-learning SAT solver as a series of decision making and clause learning. Table 1 shows the sequence of decisions made and clauses learned by the solver in chronological order. In this example, we assume that the solver makes decisions in alphabetical order and always sets decision variables to true. Implications derived by unit resolution after each decision and after each clause learning are also shown. Lastly, each step is associated with a level, which is simply the number of decisions currently in effect. After the

Decisions/learned clauses	a	b	$(\neg a \vee \neg c)$	$(\neg a)$	b	$(a \vee \neg b)$	(a)
Implications	-	$c, d, e,$ false	$\neg c, d, e,$ false	$\neg a$	$c,$ false	$\neg c, e, \neg f,$ false	false
Levels	1	2	1	0	1	0	0

Table 1. An execution trace of a typical modern clause-learning SAT solver.

first decision ($a =$ true), no implication is derived. However, after setting $b =$ true, unit resolution will derive implications c, d, e and find that $(\neg a \vee \neg d \vee \neg e)$ is falsified (indicated by false in the implication row). From this conflict, the solver will learn $(\neg a \vee \neg c)$ and backtrack to level 1. Applying unit resolution on this clause will result in implications $\neg c, d, e$ and another conflict, from which the solver learns $(\neg a)$ and backtracks to the top level (level 0). Asserting $(\neg a)$ produces only one implication. The next decision is $b =$ true, because a is already set to false. The solver will encounter yet another conflict and derive $(a \vee \neg b)$. Asserting this clause at the top level yields a conflict and (a) can be derived. Since the solver has learned $(\neg a)$ and (a), the empty clause (false) can be derived and the solver can now conclude that Δ is unsatisfiable.

Each conflict clause learned by the solver can be derived by resolving clauses present in the knowledge base of the solvers at the time of the conflict. Hence, when a clause-learning solver solves an unsatisfiable CNF, the conflict clauses learned by the solver can be thought of as traces of the refutation proof produced by the solver. A full refutation proof can be extracted from any run of clause-learning SAT solvers (on an unsatisfiable CNF) if the solvers keep track of every resolution performed during their executions [11].

Figure 1 shows the refutation proof of Δ produced by the solver in the above example, demonstrating how the conflict clauses come together to form a proof of the empty clause. The conflict clauses are enclosed in boxes in this figure. Other clauses in the proof are either original clauses in Δ or are intermediate resolvents, which are not kept by the solver. The width of this refutation proof is 4, because the longest clause, $(a \vee b \vee c \vee f)$, contains 4 literals.

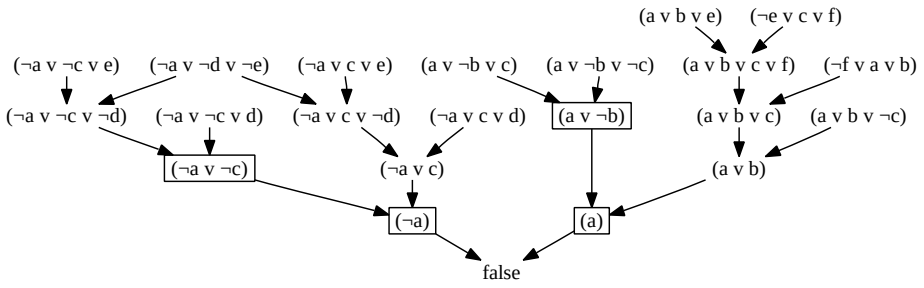

Fig. 1. A refutation proof generated by a modern SAT solver. Conflict clauses are shown in boxes.

2.3 A Width-Based Algorithm for SAT

Galil [12] proposed a SAT algorithm which runs in time exponential in the width of the CNF formula. This algorithm, which was later reformulated in [13] and [9], works by deriving all resolvents of size $\leq k$, for increasing k. Since there are only $O(n^k)$ clauses of size $\leq k$, where n is the total number of variables, this algorithm works well on formulas with bounded or small widths. Moreover, it was shown in [9] that this algorithm runs in time that is at most quasi-polynomial in the size of the smallest tree-like refutation proof (i.e., optimal DPLL).

Nevertheless, one drawback which limits the practicality of this approach is the amount of memory it requires. Even though the space complexity of the algorithm is only exponential in the width of the proof, in practice, this could be a serious limiting factor–especially when compared to the clause-learning descendants of DPLL, which perform resolution in a more directed way and keep only a fraction of the resolvents in the knowledge base.

The restart policies that we propose in this work can be thought of as a way to efficiently combine the benefits of both approaches. In other words, our approach can be viewed as a way of using the low memory requirement of modern clause-learning SAT algorithms to loosely imitate the above width-based algorithm.

3 Existing and Width-Based Restart Policies

3.1 Existing Restart Policies

In this section, we briefly review existing restart policies used by state-of-the-art clause-learning SAT solvers. One common characteristic of these policies is that they use the number of conflicts experienced by the solver to determine restarting points. According to these policies, the solvers restart as soon as the number of conflicts (since the last restart) exceeds the current threshold. Since a typical clause-learning SAT solver learns one clause per conflict, this class of restart policies can be viewed as a way of roughly enforcing a limit on the size of the refutation proof currently considered by the solver. For this reason, we will refer to this class of policies as *size-based restart policies*. These policies only differ in

the way the size threshold is updated at each restart. In the following discussion, we group these policies based on their methods of updating the threshold.

1. Arithmetic series: the threshold is increased by a constant amount (≥ 0) at every restart. This type of policy was used (with different parameters) in zChaff (2004) [14], Berkmin [4], Siege [10], and Eureka [15].
2. Geometric series: the threshold is multiplied by a constant factor (> 1) at every restart. This type of policy is used in MiniSat 1.14 and 2.0 [3].
3. Inner-outer geometric series: the solver maintains two thresholds (inner and outer). The inner threshold is used to trigger restarts and is multiplied by a constant factor (> 1) at every restart. However, if the value of the inner threshold exceeds the value of the outer threshold, the inner threshold is reset back to its minimum value, while the outer threshold is multiplied by a constant factor (> 1). PicoSAT [16] uses this policy.
4. Luby's series [5]: the threshold is updated according to the following sequence: $x, x, 2x, x, x, 2x, 4x, x, x, 2x, x, x, 2x, 4x, 8x, ...$, where x is a constant called Luby's unit (see [5] for more details). TiniSAT [17] , Rsat [18], and the latest version of MiniSat [19] use this restart policy.

Clearly, one drawback of these policies is that they are insensitive to the actual search behavior of the solver. Dynamic policies leverage on additional information generated during the execution of the solver to improve performance. In [6], the diversity of partial assignments current explored by the solver is used to create another layer of control, which helps prevent the solvers from restarting too frequently on some problems. In [7], the number of conflicts experienced below each search branch is used to determine when to restart.

3.2 Width-Based Restart Policy

Our approach to restart is based on a different model, which does not rely on the number of conflicts experienced by the solvers. Rather, we pay attention to the sizes of conflict clauses learned by the solver. If the CNF in question has a short refutation proof, the well-known result in [9] states that the formula must also have a refutation proof with a small width. If a formula has a proof with width k, then we know we can certainly find such a proof in time $O(n^k)$. Thus, enforcing a limit on proof width allows us to bound not just the size of the proof found, but also the amount of work needed to find such a proof.

In a *width-based* restart policy, the solver maintains a width limit W at any given moment. Any conflict clause whose length is greater than the current value of W is called a *violating clause*. In the most general form, the solver restarts as soon as it derives N or more violating clauses since the last time it restarted. For example, if $W = 10$ and $N = 3$, the solver will restart once at least 3 clauses of length 11 or greater are derived.[1] Note that violating clauses are not deleted

[1] We found that restarting only when the solver is not in a conflict state simplifies the implementation. In this approach, it is possible for the number of violating clauses to be (usually slightly) greater than N when the solver actually restarts.

by the solver immediately, but are treated normally just like non-violating ones. During the execution of the solver, the value of W may be kept constant or changed based on some criteria. This choice does not affect the completeness of clause-learning solvers as long as a complete clause-deleting policy is employed. Note also that, the absence of conflict clauses of size $> W$ does not guarantee that the width of the refutation proof generated by the solver will be $\leq W$. For instance, consider again the refutation proof in Figure 1. Even though every conflict clause in this proof has length at most 2, the width of this proof is actually 4. In general, the clause-learning algorithm may generate some long intermediate clauses, which do not get learned by the solver. These clauses are not taken into account in our approach.

4 Potential Benefits of Width-Based Policies

In this section, we demonstrate the potential benefits of width-based restart policies by comparing them against size-based policies on interesting SAT problems with relatively small widths. If the width k of an unsatisfiable CNF is given, one natural restart policy is to restart as soon as a conflict clause of size $> k$ is learned. To demonstrate the benefits of this approach, we will show that a width-based policy with an appropriate width limit can significantly outperform size-based policies used by state-of-the-art solvers. All experiments discussed in this section were performed on a computer with a 1.83GHz CPU and 1.5GB RAM. We set the timeout limit to 2000 seconds. We used Rsat [18] (without the preprocessor) in the following experiments.

In the first experiment, we used the unsatisfiable grid pebbling formulas with two variables per node as described in [8]. All formulas have a very small constant width (4). Nevertheless, this family was shown to be difficult for tree-like resolution [20]. Evaluated in this experiment are size-based policies (using arithmetic, geometric, and Luby's series), and a width-based restart policy. An increment of 700 was used for the arithmetic series (like zChaff 2004), a factor of 1.5 was used for the geometric series (like MiniSAT 1.14), and the Luby's unit was set to 512 (like TiniSAT, Rsat) for the Luby's series. The width limit of the width-based policy was set to 4. Table 2 reports the running time of Rsat with the considered policies on this set of problems. The first row shows the grid sizes of the grid pebbling formulas (i.e., the numbers of layers in [8]). Each remaining row shows the running time of a restart policy on these problems.

Grid size	51	52	53	151	152	153	201	202	203	238	239	240
Arith.	1	1	3	124	111	81	193	225	262	410	477	T/O
Geo.	21	210	379	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O
Luby	3	3	4	200	87	202	569	903	640	T/O	T/O	T/O
Width	1	1	1	36	27	21	85	93	108	244	414	257

Table 2. Running time (in seconds) of Rsat with different restart policies on unsatisfiable grid pebbling formulas of different sizes.

Figure 2 (a) is a plot of the running time of all restart policies as functions of grid size. According to the result, the geometric size-based policy has the worst

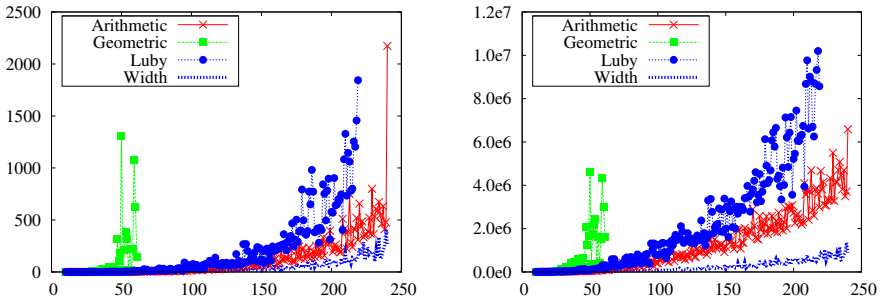

Fig. 2. Performance of Rsat with different restart policies on grid pebbling problems. In both plots, the x-axes represents the grid size. The y-axis of left (right) plot represents the running time (number of conflicts).

performance as it begins to timeout when the grid size is only about 60. The policy based on Luby's series starts to timeout when the grid size gets larger than 220. The policy based on arithmetic series performs quite well on these problems and could solve the problems up to grid size equal to 239, because it has relatively short restart periods, which are very effective for preventing the solver from getting stuck deriving long, useless clauses. In any case, the width-based policy has the best performance on these problems–up to an order of magnitude faster than the Luby size-based policy and several times faster than the arithmetic size-based policy. The superiority of the width-based policy becomes even more apparent when the solvers are compared in terms of number of conflicts needed to solve the problems (Figure 2 (b)). In our experiment (result not shown here), Rsat with the width-based restart policy could solve the problem with grid size equal 500 in 1,373 seconds.

We also experimented with the satisfiable version of the grid pebbling formulas (as described in [8]). Again, the width-based policy dominates the size-based policies both in terms of running time and conflicts. For instance, at grid size equal 260, the arithmetic size-based policy took 421 seconds, the Luby size-based policy took 922 seconds, while the width-based policy only took 134 seconds (the geometric size-based timed out for grid size ≥ 150).

Figure 3 shows similar results for the GT_n family of unsatisfiable problems [8]. A GT_n formula is a formula over $\sim n^2$ variables whose width is linear in n. In this case, we use the same set of size-based policies and set the width limit of the width-based policy to be 20. The result shows that the geometric size-based policy timed out for $n \geq 21$, the arithmetic size-based policy timed out after $n = 24$, and Luby size-based policy timed out after $n = 25$. The width-based policy was able to solve the problem with $n = 30$ in about 480 seconds. To give a sense of the hardness of these problems, consider the problem gt-ordering-sat-gt-040.sat05-1297.reshuffled-07 from the crafted category of the SAT competition 2007. In the competition, this problem was not solved by MiniSat, Rsat, TiniSat, or PicoSat under a 5000-second timeout. However, it could be solved with a width-based policy with the width limit set to 20 in 40 seconds.

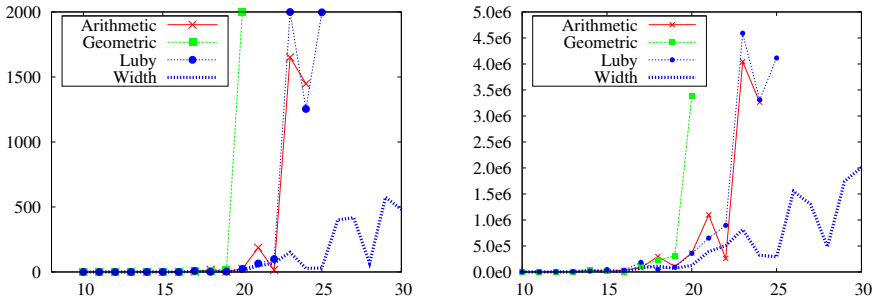

Fig. 3. (Left) running time of Rsat with different policies on unsatisfiable GT_n formulas as functions of n. (Right) number of conflicts as functions of n.

We found that some industrial problems can also be solved without requiring any long clauses to be learned. For example, consider the dspam_dump family from the SAT competition 2007. These problems were generated by CALYSTO [21, 22] from a software verification task (NULL-pointer dereferencing) on a spam filter. Based on our experiment, these unsatisfiable problems could be easily solved without any long conflict clauses, yet clause-learning solvers may derive many long clauses. Table 3 shows the information collected from running Rsat, which uses Luby size-based policy, on selected problems from this family.[2] The first column shows the names of the problems. The second and third columns reports the number of variables and clauses. The forth column reports the size of the largest conflict clause of each refutation proof found by Rsat. The remaining columns show the running time, the size of the largest conflict clause learned, and the percentage of conflict clauses longer than the ones needed by the proof, respectively. Rsat with Luby size-based restart policy took over 2,200

Problem	vars	clauses	largest clause in proof	running time (s)	largest clause size	% long clauses
dspam_dump_vc1080	118,298	372,017	4	205	1,147	86.5
dspam_dump_vc1081	118,426	372,337	3	1,716	2,533	96.6
dspam_dump_vc1103	280,972	921,211	5	195	1,408	85.9
dspam_dump_vc1104	280,972	921,147	3	169	1,229	89.1

Table 3. Information obtained from the executions of Rsat (using Luby size-based restart policy) on dspam_dump problems from SAT competition 2007.

seconds to solve all four problems. Moreover, most of the clauses learned by the solver were unnecessarily long. Nevertheless, these problems become easy if a width-based restart policy (with an appropriate width limit) is used. Rsat using a width-based restart policy with width limit set to 6 can solve all these problems within 25 seconds (combined).

The results of these experiments show that a width-based restart policy (with the right width limit) can dramatically reduce the running time of the solver on some problems. However, in practice, the width of the problem is not known

[2] We disabled conflict clause deletion in order to collect some statistics.

beforehand and width computation is believed to be very hard [23].[3] Therefore, we need to adjust the width limit dynamically to obtain good performance.

5 Adjusting Width Limits

In this section, we describe a general algorithm for updating the width limit and propose several methods for updating the width limit. In a general width-based policy, at any given time, the solver maintains one width limit W and restarts as soon as it derives N or more conflict clauses with size greater than W. We consider the following methods for updating W.

1. Arithmetic series: after R restarts at the current limit, W is incremented by a constant C_1.
2. Geometric series: after R restarts at the current limit, W is multiplied by a constant C_2.
3. Inner-outer geometric series: after R restarts at the current limit, W is multiplied by a constant factor C_2. However, as soon as the value of W reaches V, it is reset to its initial value and V is multiplied by a constant factor C_3.
4. Luby series: after R restarts at the current limit, W is updated to be the next number in the Luby series with unit U.

The values of $N, R, U, V, C_1, C_2, C_3$ are parameters that need to be fine-tuned for these policies. Nevertheless, finding optimal values for these parameters is not the main focus of this work. In subsequent experiments, we set $W = 15$ (initially), $N = 10, R = 1, U = 6, V = 20$ (initially), $C_1 = 1, C_2 = 1.005, C_3 = 1.05$. One can certainly envision policies which adjust these parameters dynamically.

In addition to these pure width-based restart policies, we also consider their combinations with a size-based restart policy. In this case, the proofs explored by the solvers are loosely bounded both in terms of size and width. For this combination, we used the size-based restart policy based on Luby's series, which has been found to yield good performance on industrial problems [5]. In such a hybrid policy, the width limit and the size threshold are enforced independently. That is, the solver restarts based on clause size as described above and, moreover, if the number of conflicts experienced by the solver (since the last time it restarted based on number of conflicts) reaches the size threshold, the solver also restarts (without updating the width limit).

6 Experimental Results

In this section, we evaluate the performance of the restart policies discussed in the previous section. All experiments were performed on a computer with a 3.8GHz CPU and 4GB of RAM. The timeout was set to 30 minutes per problem. The use of proprocessor was diabled in all experiments in order to obtain the impacts of the restart policies on pure clause-learning solvers.

[3] The problem of computing width was conjectured to be EXPTIME-complete.

In the first experiment, we compared the proposed width-based restart policies against the Luby size-based policy (unit=512) on 175 industrial problems from the last SAT competition.[4] Table 4 reports the number of problems solved by each policy. According to the table, (pure) width-based policies seem to consistently result in worse performance on satisfiable problems. This could be due to the significant increase in the number of restarts introduced by the width-based policies. More frequent restarts cause the solver to remake many decisions and spending more time in unit propagation.[5] For unsatisfiable problems, the results are more comparable. The geometric width-based policy actually solved 5 more unsatisfiable problems than the size-based policy. Overall, the performance of the geometric width-based policy is about the same as that of the Luby size-based policy. This result demonstrates that width-based policies, could be competitive to a size-based policy. Note that the parameters used in our experiment were not fine-tuned. The table also shows that the hybrid policies consistently outperformed the Luby size-based policy (except the one with inner-outer width-based policy, which performed poorly on satisfiable problems). The combination of the geometric width-based policy and the Luby size-based policy, in particular, appears to be the best version on this set of problems.

Policy	Solved problems		
	Total	SAT	UNSAT
Size-based (Luby,unit=512)	107	49	58
Width-based (Arith.)	100	46	54
Width-based (Geo.)	108	45	63
Width-based (Luby)	103	47	56
Width-based (In-out.)	90	36	54
Width-based (Arith.)+size-based (Luby)	110	48	62
Width-based (Geo.)+size-based (Luby)	115	52	63
Width-based (Luby)+size-based (Luby)	114	54	60
Width-based (In-out.)+size-based (Luby)	106	43	63

Table 4. Number of industrial problems from the SAT competition 2007 solved by different restart policies.

Next, we compare the best policies from the previous experiment against other dynamic restart policies in order to establish a context for the benefit of width-based restart policies. This time, we also consider problems from SAT-Race 2006 and some hardware verification problems.[6] Considered in this experiment are the following versions of Rsat.

1. Rsat 2.00 (SAT competition 2007 version) [Rsat]. This version of Rsat uses a size-based restart policy based on Luby's series with Luby's unit set to 512.

[4] Obtained from http://www.satcompetition.org.

[5] For example, whenever the width-based policy (geo.) solves the problem with the number of conflicts comparable (within 5%) to that of the size-based policy (Luby), it makes 42% more decisions on average.

[6] The hardware verification problems were obtained from http://www.miroslav-velev.com/sat_benchmarks.html.

2. Rsat with agility-based restart policy [Rsat-ag]. Restart is disabled if the agility of the solver is greater than 0.25 (as described in [6]).
3. Rsat with local restarts [Rsat-lc]. A restart is triggered only when the number of conflicts under some search branch exceeds the threshold (as described in [7]). The Luby's series (unit=512) is used to update the threshold.
4. Rsat with hybrid restart policies [Rsat-ws-1,2,3]. The Luby size-based policy is combined with (1) the geometric width-based, (2) the Luby width-based, and (3) the arithmetic width-based policies.

Family	Total	Solved problems					
		Rsat	Rsat-ag	Rsat-lc	Rsat-ws-1	Rsat-ws-2	Rsat-ws-3
SAT comp. 07	175	107	109	114	**115**	114	110
SAT-Race'06	100	86	87	84	**90**	88	84
dlx-iq-unsat-1.0	32	11	12	7	**18**	14	17
fvp-unsat-1.0,2.0,3.0	32	26	26	26	26	26	26
liveness-sat-1.0	10	5	5	6	**7**	6	6
liveness-unsat-2.0	9	3	3	3	3	3	3
pipe-ooo-1.0,1.1	29	12	12	11	**13**	12	**13**
pipe-unsat-1.0,1.1	27	14	14	13	**16**	**16**	**16**
vliw-unsat-2.0,4.0	13	0	0	0	**2**	0	0
Total	427	264	268	264	**290**	279	275

Table 5. Number of problems solved by different versions of Rsat.

Table 5 shows the number of problems solved by each solver. Overall, the agility-based restart policy allows Rsat to solve a few more problems, while local restarts yield about the same performance as the original Rsat. These techniques seem to be most effective on problems from the SAT competition and SAT-Race'06.[7] The hybrid restart policies have the best overall performance. Rsat-ws-1,2,3 solved 26, 15, 11 more problems than Rsat, respectively. Clearly, the geometric width-based and Luby size-based combination (Rsat-ws-1) yielded the best performance. Note that, in a hybrid policy, number of restarts triggered by width violations usually dominates size-based restarts.[8] Moreover, using width-based restart policies also tends to reduce the sizes of clauses learned. On the SAT competition 2007 problems, the sizes of clauses learned by Rsat-ws-1 is only 76% of those learned by Rsat on average. On SAT-Race'06 problems, this percentage is 81%.

Figure 4 shows the running time profiles of different versions of Rsat on the sets of problems in Table 5. For clarity, we show only one profile of Rsat with a hybrid policy, Rsat-ws-1. The left plot shows the profiles on satisfiable problems, while the right plot shows the profiles on unsatisfiable problems. The left plot indicates that Rsat-ws-1 actually performed slightly worse than Rsat on satisfiable problems, even though it ended up solving 2 more problems. The right plot, however, shows that Rsat-ws-1 is the clear winner on unsatisfiable problems. Rsat-ag and Rsat-lc appear to have comparable profiles to Rsat.

[7] We did not optimize the parameters used in these techniques.

[8] E.g., on SAT'07 problems, 78% of restarts are width-based, while on SAT-Race'06 problems, 73% of restarts are width-based (based on the execution of Rsat-ws-1).

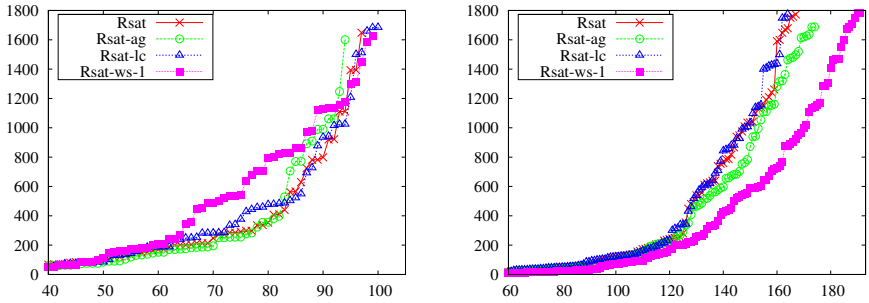

Fig. 4. Running time profiles of Rsat with different restart policies on (left) satisfiable and (right) unsatisfiable problems. Both x-axes represent the number of solved problems, while the y-axes represent running time in seconds.

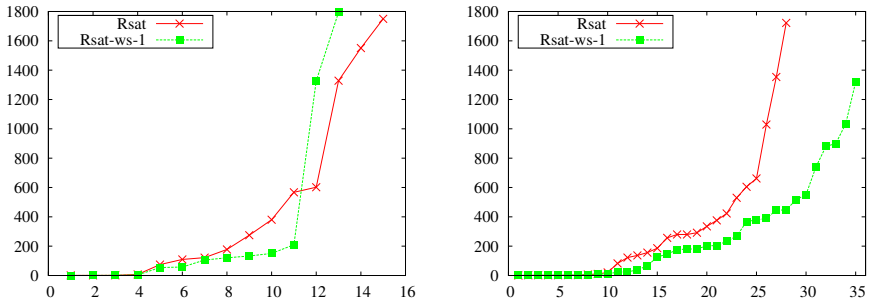

Fig. 5. Running time profiles of Rsat with different restart policies on (left) satisfiable and (right) unsatisfiable problems from the crafted category of SAT'07 competition.

Our experiment on crafted problems from the SAT competition 2007 also confirms the benefit of the hybrid policy. We compared the performance of Rsat against the best hybrid policy, Rsat-ws-1. Figure 5 shows the running time profiles of these solvers on satisfiable (left) and unsatisfiable (right) problems. Even though Rsat-ws-1 solved fewer satisfiable problems, it took less time on most of the ones it solved. Moreover, Rsat-ws-1 solved 7 more unsatisfiable problems and took less time on those that both versions could solve.

We also tested the hybrid restart policy on MiniSat 2.0 (no preprocessor). By default, MiniSat uses a geometric size-based restart policy. We added the geometric width-based policy on top of this to obtain a hybrid policy (geometric width-based + geometric size-based). Our experiment on the industrial problems of SAT competition 2007 showed that MiniSat solved 109 problems, while MiniSat with the hybrid restart policy solved 114 problems.[9]

[9] Here, we used progress saving [24] in both versions of MiniSat as this technique seems to allow a frequent restart policy to realize its full potential. Without progress saving, both versions solved fewer problems and the improvement is less significant.

7 Related Work

The concept of *space-bounded learning* has long been studied in CSP [25, 26] and SAT [27]. This approach restricts the algorithm to learning only those constraints with a limited number of variables. The restart policies we propose still allows long clauses to be learned, but use restarts to discourage their learning.

A technique in SAT, which tries to achieve similar goals, is known as *decision stack shrinking*, which was introduced by JeruSAT [28] and later used by zChaff2004 [14]. This technique tries to force the solver to discover a conflict at a lower level, thus deriving shorter a conflict clause. The technique is invoked whenever a long conflict clause is learned. Upon learning such a clause, the solver examines the decision levels of the literals in the clause and backtracks to the lowest level that is sufficiently smaller than the next higher level of any literal in the conflict clause. The solver then makes assignments in order to falsify the conflict clause (and run into the same conflict). Since some of the variables unassigned by the backtrack may not get assigned by the time the new conflict is discovered, the size of the decision stack is likely going to reduce, leading to potentially a shorter conflict clause. Our approach utilizes restarts to direct the solvers away from undesirable parts of the search space, thus inducing shorter conflict clauses. Decision stack shrinking, however, aims at improving the quality of conflict clause learned from a given (or similar) conflict.

8 Conclusions

We presented a new class of restart policies, called width-based restart policies, for clause-learning SAT solvers. These policies trigger a restart whenever the number of long clauses learned by the solver is sufficiently large. They can be thought of as ways to encourage the solvers to discover refutation proofs with small widths (instead of small sizes as done in traditional policies). Our study shows that width-based restart policies can be orders of magnitude faster than policies based on number of conflicts on special classes of problems. We then propose a general algorithm for adjusting the width limits of width-based policies. Our experiment on industrial problems showed that pure width-based policies are competitive to the policy used by state-of-the-art solvers. Moreover, we show that width-based policies, when combined with a size-based policy, can lead to significant improvements on industrial and crafted problems.

References

1. Gomes, C.P., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In: Principles and Practice of Constraint Programming. (1997) 121–135
2. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: Proc. of DAC'01. (2001) 530–535
3. Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT'03. (2003) 502–518
4. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. In: DATE '02. (2002) 142–149

5. Huang, J.: The effect of restarts on the efficiency of clause learning. In: Proc. of IJCAI-07. (2007) 2318–2323
6. Biere, A.: Adaptive restart strategies for conflict driven sat solvers. In: SAT. (2008) 28–33
7. Ryvchin, V., Strichman, O.: Local restarts. In: SAT. (2008) 271–276
8. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. JAIR **22** (2004) 319–351
9. Ben-Sasson, E., Wigderson, A.: Short proofs are narrow—resolution made simple. J. ACM **48**(2) (2001) 149–169
10. Ryan, L.: Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University (2004)
11. Zhang, L., Malik, S.: Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE '03. (2003) 880–885
12. Galil, Z.: On resolution with clauses of bounded size. SIAM Journal on Computing **6**(3) (1977) 444–459
13. Beame, P., Pitassi, T.: Simplified and improved resolution lower bounds. Foundations of Computer Science, Annual IEEE Symposium on **0** (1996) 274
14. Mahajan, Y.S., Fu, Z., Malik, S.: Zchaff2004: An efficient sat solver. In: Proc. of SAT-05. (2005) 360–375
15. Alexander Nadel, Moran Gordon, A.P., Hanna, Z.: Eureka-2006 sat solver Solver description for SAT-Race 2006.
16. Biere, A.: Picosat essentials. JSAT (2008) 75–97
17. Huang, J.: A case for simple sat solvers. In: CP-07. (2007) 839–846
18. Pipatsrisawat, K., Darwiche, A.: Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Comp. Sci. Department, UCLA (2007)
19. Sörensson, N., Eén, N.: Minisat 2.1 and minisat++ 1.0–sat race 2008 edtns (2008)
20. Ben-Sasson, E., Impagliazzo, R., Wigderson, A.: Near optimal separation of tree-like and general resolution. Combinatorica **24**(4) (2004) 585–603
21. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. In: Proc. of CAV'07. (2007) 371–383
22. Babić, D., Hu, A.J.: Calysto: Scalable and Precise Extended Static Checking. In: Proc. of ICSE-08. (2008) 211–220
23. Hertel, A., Urquhart, A.: Comments on eccc report tr06-133: The resolution width problem is exptime-complete. Technical Report TR09-003, ECCC (2009)
24. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Proc. of SAT'07. (2007) 294–299
25. Dechter, R.: Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artif. Intell. **41**(3) (1990) 273–312
26. Bayardo, R.J., Miranker, D.P.: A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In: AAAI-96. (1996) 298–304
27. Bayardo, R.J.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proc. of AAAI-97, Providence, Rhode Island (1997) 203–208
28. Nadel, A.: Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis (2002)