Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 144 (2006) 35–50

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Adaptive Application of SAT Solving Techniques

Ohad Shacham[1,2] Karen Yorav[1,2]

IBM Haifa Research Laboratory
Israel

Abstract

New heuristics and strategies have enabled major advancements in SAT solving in recent years. However, experimentation has shown that there is no winning solution that works in all cases. A degradation of orders of magnitude can be observed if the wrong heuristic is chosen. The problem is that it is impossible to know, in advance, which heuristics are best for a given problem. Consequently, many ideas - those that turn out to be useful for a small subset of the cases, but significantly increase run times on most others - are discarded.
We propose the notion of *Adaptive Solving* as a possible solution to this problem. In our framework, the SAT solver monitors the effectiveness of the search on-the-fly using a *Performance Metric*. The metric gives a score according to its assessment of the search progress. Based on this score, one or more heuristics are turned on or off. The goal is to use a specific heuristic or strategy when it is advantageous, and turn it off when it is not, before it does too much damage. We suggest several possible metrics, and compare their effectiveness. Our adaptive solver achieves significant speedups on a large set of examples. We also show that applying different heuristics on different parts of the search space can improve run times even beyond what can be achieved by the best heuristic on its own.

Keywords: SAT solving

1 Introduction

Recent years have seen great amounts of research on SAT solving [6,12,15]. The problem is interesting theoretically, as well as important for practical reasons. The high capabilities of advanced solvers has encouraged their use in

[1] Email: {ohads,yorav}@il.ibm.com
[2] This research is supported in part by EU contract FP6-IST-507219 (PROSYD)

doi:10.1016/j.entcs.2005.07.018

various fields such as verification, artificial intelligence, CAD, and more. This, in turn, created great incentive to invest in research of SAT solving techniques.

Our perspective on SAT solving comes from its use in *Bounded Model Checking* [3] (BMC), where the verification problem of a hardware design is translated into a Boolean formula such that a satisfying assignment, if one exists, represents a counterexample. Most tools that implement this framework are based on DPLL-style SAT solvers. Although the ideas presented in this work are general, and may easily be applied to other types of SAT solvers, our results are tuned for BMC instances. We believe the method is most efficient when applied to instances that have internal structure.

Modern SAT solvers rely heavily on various heuristics and strategies such as decision heuristics, restart strategies, learning strategies, clause deletion strategies, etc [2,6,11,12,15,16]. However, many ideas that seem appealing in theory turn out not to perform well in practice, decreasing the run time on a few examples while increasing it on most others. As a result, these ideas are discarded. Even successful heuristics are not useful in all cases, but it is impossible to know beforehand which heuristics are most suitable for a given example.

In this paper we propose the concept of *Adaptive Solving*. Adaptive Solving optimizes the way different strategies are used, by applying them when they are useful and turning them off when they are not. The adaptive solver tracks the performance of the search and evaluates it using a *Performance Metric*. Whenever the search seems not to be progressing well enough, it changes run-time parameters by enabling or disabling heuristics. In this way the adaptive solver is capable of making use of heuristics that do not work well in all cases.

We propose several metrics to be used in adaptive solving. These metrics are easy to compute and incur a negligible overhead. They track different aspects of the search and give a score accordingly. We compare their effectiveness and present some insights to their use.

Our BMC tool is part of RuleBasePE [9], a parallel model checker developed at the IBM Haifa Research Laboratory. This tool uses our in-house solver called ***Mage***. We have implemented an adaptive version of Mage and used it on a large number of examples. Results show that adaptive solving reduces the overall run time by more than a third, and achieves speedups of up to 12**x** on single examples.

Naturally, there are examples for which run time is increased as a result of enabling or disabling a heuristic. The overall reduction is achieved by significantly reducing run times on many examples, while increasing the run time of others to a lesser degree. Results show that in general speedups are better for larger examples, making the method highly scalable. Furthermore, we show that even when a heuristic performs badly for a certain example, applying it on parts of the search may give better results than not using it at all. This means that adaptive application of heuristics gives performance improvements over the best heuristic on its own.

Our implementation is an initial experiment in Adaptive Solving. It controls only one heuristic, and uses only one metric at a time, and yet it achieved impressive speedups.

Related Work

Previous attempts have been made at assessing the progress of the solver's search for a satisfiable assignment. Aloul, Sierawski, and Sakallah view the conjunction of conflict clauses as representing the space that is yet to be covered, with each added conflict clause reducing this space until it is empty. Their Satometer [1] tool keeps a BDD representation of this conjunction. Of course, the exact space cannot be calculated, so the tool uses an approximation. The drawback of this approach is its huge overhead - both in space and in computation, which prevents it from being used as a performance metric.

Another related work is presented by Herbstritt and Becker in [7], where decision heuristics are chosen dynamically, according to a set of probability functions. The probabilities are changed according to several criteria. Our work is more general because we address any run-time parameter of the solver, not just the decision heuristic.

Nudelman et al [14] use machine learning to identify features of CNFs. Using a large training set they learn the correlation between the hardness of the problem and the result of different kinds of analyses of the CNF. Their SATzilla tool [13] profiles the run times of several different solvers using the same training set and features. When a new problem is to be solved, the tool computes the different features and then chooses the solver predicted to have the least run time. However, the features they use to choose the solver are not applicable for us, because they are more relevant for random instances than for structured ones, and because they need to be computed beforehand.

In Lagoudakis and Littman's work [10], decision heuristics are chosen according to a value function, which is calculated on the current state of the search. The value function is created beforehand, using a training set. The training set must be a significantly large set of examples that are similar in some sense to the CNFs we want to solve. Using a training set is problematic both because it incurs a high overhead, and because it is difficult to generate an adequate training set, especially in the setting of SAT-based verification.

The remainder of the paper is structured as follows. Section 2 provides an overview of DPLL-style SAT solver algorithms. Section 3 presents a variety of performance metrics. Section 4 defines Adaptive Solving. Finally, Section 5 shows the experimental results, and Section 6 presents our conclusions.

2 Background on SAT Solving

Given a Boolean formula F over a set of variables V, the task of the SAT solver is to find an assignment to all variables in V such that F is satisfied. The formula is given in *Conjunctive Normal Form* (CNF). A CNF formula is a conjunction of *clauses*, where each clause is a disjunction of *literals*. A *literal* is an instance of a variable, x, or its negation, $\overline{x}$.

A literal may have one of three values: *true*, *false*, or *undef* (undefined). A clause in which one literal is undefined and all the rest are *false* is called a *unit clause*. Such a clause forces an assignment of *true* to the undefined literal, as this is the only way to satisfy the formula.

2.1 SAT Solving Algorithms

Our implementation is geared towards DPLL-style solvers [5] with conflict clause learning and non-chronological backtracking [2,11], although the adaptive solving concept can be applied to other solving schemes. We give a brief description of DPLL with learning. For a more thorough discussion see [17].

```
while(1) {
  if (decide_next_branch())
    while (deduce() == CONFLICT) {
      blevel = analyze_conflicts();
      if (blevel == 0) return UNSAT;
      else bactrack(blevel);
    }
  else
    return SAT;
}
```

Fig. 1. Basic DPLL algorithm with learning

Figure 1 gives a bird's eye view of a DPLL algorithm with learning. The algorithm iteratively chooses a value for a variable (`decide_next_branch()`). If all variables become assigned the algorithm halts, and outputs a satisfying assignment. Otherwise, the implications of this assignment are carried through by the `deduce()` function. If `deduce()` reveals a conflict, the reason for the conflict is analyzed and a conflict clause is added to the database. The conflict clause summarizes the combination of values that lead to the conflict and prevents this combination from being assigned a second time. The function `analyze_conflicts()` returns a decision level to which the algorithm backtracks. If this is level 0, it means that there exists a conflict even without a single decision, which means that the formula is unsatisfiable. Otherwise, the algorithm backtracks and continues the search.

2.2 Decisions

The `decide_next_branch()` function chooses a variable that is undefined and assigns to it either *true* or *false*. This is called a decision, and the variable is called the decision variable. The algorithm that controls the way this choice is made is the *decision heuristic*. Each decision is associated with a number, called the *decision level*. All the implications that result from one decision are associated with the same decision level. When constant values are revealed, they are associated with decision level zero.

2.3 Boolean Constraint Propagation

Boolean Constraint Propagation (BCP) is the process of propagating the effect of an assignment. This is the task of the function `deduce()` in Figure 1. Each assignment may cause several clauses to become unit. Each unit clause implies an assignment, which may in turn result in more unit clauses. During BCP, this process is iterated until no further assignments can be implied.

Since modern solvers spend roughly 80% of their time carrying out BCP, it is crucial that this process be implemented efficiently. The technique used in Mage, as in zChaff and others, is to mark two literals in every clause as *watched literals*. The rational is that a clause of length n (with $n \geq 2$) can become a unit clause only after $n-1$ of its literals have been assigned *false*. Only unit clauses can cause implications, so as long as the two watched literals of a clause are undefined (or *true*) this clause is not unit, and there is no need to examine it during BCP. Whenever a literal l is assigned *false*, all clauses in which l is watched are examined to see whether they have become unit.

2.4 Clause Learning and Non-Chronological Backtracking

A conflict happens when BCP propagates a certain assignment and discovers a clause with all its literals set to *false*. During conflict analysis, the chain of implications that resulted in the conflict is analyzed, and the reason for the conflict is summarized in a *conflict clause*. This clause describes a combination of assignments that should not be repeated as they are conflicting. The conflict clause is added to the clause database, thus pruning the search space that remains to be traversed.

3 Performance Metrics

The performance of a SAT solver is measured by the time it takes to solve a given instance. In order to evaluate whether a certain heuristic or strategy is beneficial, the overall run time on many different formulas is compared. The reality is that even for the best heuristics, there are examples on which they do not work well. The role of the performance metric is to assess the compatibility of the solver settings for a specific example during the run.

We propose the following requirements on performance metrics:

(i) The metric can be evaluated during the solver's run.

(ii) It can be calculated efficiently, i.e. with low overhead, and with minimal additional space consumption.

(iii) The metric gives a score that (roughly) corresponds to the effectiveness of the search.

The nature of the SAT problem is such that it is unrealistic to expect to find a metric that will give a perfect correlation to the end result. However, based on our understanding of the way the solver operates, we suggest several candidates, which are listed below. Each metric is evaluated on a *sample* of the run, consisting of a constant number of decisions.

3.1 Decision Level

When a decision causes a conflict, the solver backtracks to a previous decision level and cancels all the assignments made in between. In this case the decision level of the next decision will be some smaller number. Otherwise, if there was no conflict, the decision level increments by one.

The **DL** metric looks at the average decision level in a single sample (in our case - 2048 decisions). The solver reaches high decision levels when it makes a large number of decisions without conflicts, or when the conflicts do not set it back by much. As a result many variables keep their value for long periods of time. This could mean that the solver spends significant amounts of time searching in a small part of the state space. On the other hand, a low average may indicate a high conflict rate. Since each conflict clause restricts the space that remains to be searched, a high conflict rate is a good sign. This leads us to expect that an efficient run is one in which the average decision level is relatively low.

It should be noted that the average decision level is greatly influenced by the internal structure of the solver and the chosen decision heuristic. When experimenting with this metric, we found that the average decision level in zChaff was, in general, twice as high as the average level for Mage when running on the same formula.

3.2 Conflict Clause Size

As mentioned earlier, every conflict clause that is added is a constraint that reduces the state space that remains to be searched. In general, smaller conflict clauses are capable of posing a greater restriction. So, although it is possible for a specific small clause to be less useful than a very large one, in general we expect that very small conflict clauses advance the search more rapidly towards its goal.

Our second metric, **CCS**, is the average length of all the conflict clauses that were learned in a sample - the smaller the better. Note that the score does not explicitly reflect the actual number of conflict clauses, which could be a metric on its own.

3.3 Binary Conflict Clauses

Some Solving strategies emphasize preference towards short conflict clauses [4] and binary clauses in particular [15]. This makes sense in light of the fact that these clauses have the highest potential of generating implications (a single assignment makes the clause unit). We therefore expect that adding many binary clauses to the database greatly advances the search.

The **BIN** metric measures the percentage of binary conflict clauses out of the total number of conflict clauses learned in a given sample.

We have also considered looking at ternary clauses as a part of this metric. However, extensive experimentation revealed that the percentage of ternary conflict clauses is linearly correlated to the percentage of binary clauses. In all of our examples, these two numbers are almost equal, and they increase and decrease in the same manner from one sample to the next. We concluded that there is no added benefit in tracking ternary conflict clauses.

3.4 BCP Ratio

When a watched literal l in a clause c becomes $false$, the BCP process must go over the literals in c and look for a new watched literal. In the worst case scenario, all the literals in c are examined. The **BCP** metric measures the ratio between the number of literals examined (all together) and the number of clauses visited, i.e., it calculates the average number of literals examined per clause. This ratio is indicative of the speed at which implications are carried out. Since the BCP operation is the major part of the computation, it is important to keep this number low.

3.5 Unary Clauses Learning

The `analyze_conflicts()` function is capable of producing unary conflict clauses. This amounts to learning the value some variable must have in order to satisfy the formula. The variable is then assigned a permanent value. When this happens, the algorithm backtracks to decision level zero and applies BCP to discover all implications of this assignment. Any assignment resulting from this BCP process is also permanent.

The **UNARY** metric tracks the rate at which permanent values are assigned. It gives the number of such values assigned in the last sample. An examination of the behavior of this metric reveals that learning happens in bursts. Typically, there are extended periods of time with little or no learning, and then suddenly tens or even hundreds of variables are assigned a value.

4 Adaptive Solving

As mentioned earlier, there is no one heuristic that works well in all cases, whether it is a decision heuristic or any other heuristic that is used during the search. Our solution is an *Adaptive Solver* that is capable of adapting its run-time parameters to the specific CNF it is solving. During the search, the solver looks for signs that the run-time parameters with which it is running are not optimal, and changes them on-the-fly.

An Adaptive Solver works according to the following scheme:

- The run is divided into *samples*, where each sample consists of the computation performed during a certain number of decisions.
- At the end of each sample, a *performance metric* is used to evaluate the effectiveness of the search in this sample.
- A *switching condition* decides whether the solver is progressing.
- If the switching condition evaluates to *true*, one or more *parameters* of the run are changed. We call this *a switch*.

The specifics of an adaptive solving algorithm include the size of a sample, the choice of a performance metric, the choice of parameters to change, and the condition for switching. The possibilities for all these are endless. In the end, the right choice of these elements can make the difference between success and failure. Furthermore, the choices for each element depend on the specifics of the SAT solver implementation. There are no clear cut rules for building an adaptive solver. The rest of this section describes, and motivates, some of the choices made for Mage, and the insights gained from experimentation.

4.1 The Parameter

The choice of parameters to switch is important. The idea is to choose a parameter that has a high impact on the run time. At the same time, there is no point in adaptively switching a parameter that is always useful. Luckily, there is no shortage of such options. There are numerous heuristics and strategies that are not used in practice because they are beneficial only for some examples, but detrimental for most.

In our adaptive implementation of Mage, we chose the `-sign` option as the parameter to control. As this is an initial experimentation in Adaptive Solving, only one aspect of the SAT solving algorithm was controlled adaptively. This option controls the way a value is chosen for a decision variable in the `decide_next_branch()` function. Normally, after choosing a variable to decide upon, the function chooses whether to assign it *true* or *false* by examining the scores of the corresponding literals. The default is to assign *true* to the literal with the highest score. Activating the `-sign` option will make `decide_next_branch()` assign *true* to the literal with the lower score.

Experimentation shows that in general, it is better to choose the literal with the higher score. In most examples this will result in shorter run times. However, for some examples, choosing the lower scored literal can result in a speedup of up to 4`x` (see results in Section 5).

4.2 Switching Conditions

We implemented a mechanism to compute all of the metrics mentioned in Section 3. The sample size we chose is 2048 decisions. When using a smaller sample we found that the metric score did not stabilize, and another switch would occur too soon [3].

For each of the DL, CCS, BIN, and BCP metrics, we chose an initial bound, so that a switch is made when the metric score exceeds the bound. The initial bound was chosen by running a large number of examples without adaptive solving, and inspecting the scores each metric produces. After the runs completed, we saw which were the best in terms of run times. We looked at the average score of the "bad" runs, and the average score of the "good" ones, and chose the initial bound for each metric to be some number in between.

In the case of the UNARY metric a switch is made if the number of added permanent values is low over a period of several samples. Because of the effect mentioned earlier, in which permanent values are added in bursts, the condition for this metric was set to: "perform a switch if in 14 out of the

[3] The sample size is an exponent of 2 because this makes the implementation more efficient.

last 16 samples the number of permanent values added was low" (less than 3). Again, this scheme was developed by examining the runs of good and bad examples. We noticed, for example, that even for the worst cases there may be one or two added values in each sample, which is why we do not require this number to be zero.

Initial experimentation revealed that for some of the hard problems the metric scores were consistently high, and the adaptive solver was switching parameters throughout the run. This caused a significant increase in run times. It seems that in order for a heuristic to be effective, it needs to run for a certain period of time without interruption. Consequently, we placed several mechanisms to prevent the adaptive solver from switching too often:

- When a switch is made the metric bound is incremented (or decremented), so that in order for another switch to occur the metric score will have to be slightly worse than it was in the last switch.
- After each switch, further switching is prevented during the next 20 samples.
- A limit is placed on the number of switches allowed during a single run.

5 Experimental Results

We conducted extensive experimentation on our adaptive version of Mage. Our benchmark suite includes 50 examples from the IBM Benchmarks Suite [8]. This is a collection of CNF files that originate from the verification of various industrial designs using SAT-based BMC. The benchmark is very diverse, with both long and short examples, satisfiable and unsatisfiable, various depths, etc.

Our Adaptive Solver enables switching only after $20,000$ decisions have been executed, so that no switches are made for easy problems. The benchmarking suite includes only examples that require more than $20,000$ decisions. We also made sure that the examples we consider do not abort because of time-out. This is done so that the time we choose to time out on will not influence the speedup results. Although it may seem impressive to report that there are several examples on which the native version timed-out while the adaptive version succeeded, in reality this only depends on the time-out constant. Instead of reducing this constant to generate such "impressive" examples, we chose to enlarge it to the point where all of the examples we want to run do not time-out. This required a relatively high timeout of $10,000$ seconds. The only two runs that time out on $10,000$, are examples that run with the `-sign` option and no Adaptive Solving. This version does not influence the analysis of the results for the adaptive algorithm, it is merely used to demonstrate that the `-sign` option is overall detrimental.

The analysis of the results is done by comparing the overall speedup, i.e., the speedup on the sum of the run times of all examples. This number is indicative of the effect Adaptive Solving has on a large number of examples of various sizes, such as in the case of a full verification project. This analysis places more weight on the larger examples. This is suitable for our experimentation, because our goal is to reduce the run times of large examples without hurting the small ones too much, thus reducing the overall time needed for a verification project to be completed. Calculating the average speedup, for example, would place more emphasis on reducing a two-second run to one second, than adding an hour to an example that runs two hours. This may be suitable for theoretical analysis, but it is not suitable for an industrial tool. Because this is a prototype implementation of a new idea, however, we also give the minimum and maximum speedups. We consider these an indication to the potential of Adaptive Solving.

Our experimentation was conducted on an Intel Pentium 4, with a single 2GHz CPU, 1GB RAM, running Linux. Tables 2, 3 give the run time results for all 50 examples, in seconds. It compares the run times between seven versions. **Native** is Mage running with its default parameters and no adaptive algorithms. In particular, the `-sign` option is not used, so in all decisions the sign with the highest score is chosen. **Sign** is Mage running with the `-sign` option all of the time. Versions **DL**, **CCS**, **BIN**, **BCP**, and **UNARY** correspond to adaptive versions each using the metric implied by its name (see Section 3). The Native and Sign versions do not calculate any of the metrics. Initial experimentation showed that the overhead incurred by the computation of the metric scores is negligible (only a few seconds even for the largest examples). Because the difference is so small we omit the results for a version that computes all metrics but does not apply adaptive solving.

Table 1 gives a summary of the results. In this table the "Time" rows display the sum of run times on all the examples in the benchmark (in seconds). The "Speedup" for each version is the runtime of Native divided by the runtime of that version. The "Min" and "Max" rows show the minimum and maximum speedups achieved by each version on a single example. The results for satisfiable and unsatisfiable examples are given separately, and the "ALL" section summarizes all the examples.

Table 1 shows that BIN and UNARY are the best metrics, giving speedups of 1.6 and 1.5 respectively. The BCP and CCS versions give modest speedups, and DL has a negligible speedup. For the CCS, BIN, and UNARY versions, the speedup is better on SAT instances than on UNSAT. On SAT instances alone, UNARY gives a $2x$ speedup, while on the UNSAT instances, there is hardly any gain. On the other hand, the BCP version works better on UNSAT instances. The `-sign` option is, indeed, not recommended as a default option, since it significantly increases the overall run time.

Examining the minimum and maximum speedups reveals how the overall speedup is achieved – by significantly reducing run times on some examples and only slightly increasing run times on others. For example, the worst damage the BIN version causes is a speedup of 0.75 (which equals to an increase of about a third), while its best performance is almost $12x$ faster. Note that all of the examples in the benchmark suite are non-trivial. The best speedup was achieved on an example that runs 3821 seconds on the Native version, and 325 seconds with the BIN version.

A phenomenon we encountered, and can be seen in tables 2, 3 is that in many cases the adaptive version performs better than either Native or Sign. From this we learn that different sub-spaces of the search space require different settings. This encourages us that Adaptive Solving has great potential.

	Version	`Native`	`Sign`	`DL`	`CCS`	`BIN`	`BCP`	`UNARY`
UNSAT	Time	8662	14579	8609	7726	6702	7057	7933
	Speedup	-	0.594	1.006	1.121	1.292	1.227	1.091
	Min	-	0.097	0.771	0.874	0.831	0.830	0.913
	Max	-	4.354	1.641	3.878	4.042	2.951	4.017
SAT	Time	14955	25256	13067	9228	8269	12637	7313
	Speedup	-	0.592	1.144	1.620	1.808	1.157	2.044
	Min	-	0.067	0.168	0.509	0.751	0.411	0.523
	Max	-	4.437	3.900	5.287	11.749	1.326	5.900
ALL	Time	23618	39835	21676	16954	14971	19695	15247
	Speedup	-	0.593	1.089	1.393	1.578	1.182	1.549
	Min	-	0.067	0.168	0.509	0.751	0.411	0.523
	Max	-	4.437	3.900	5.287	**11.749**	2.951	5.900

Table 1
Summary of run time results for all adaptive versions

6 Conclusions

We view the Adaptive Solving algorithm presented here as a starting point rather than a finished product. As mentioned before, there are many design decisions in the implementation of an Adaptive Solver that can make a difference in its performance. Our choices are by no means guaranteed to be the best possible.

Nevertheless, the prototype algorithm was able to achieve up to 12x speedup in run times on satisfiable examples, up to 4x speedup on unsatisfiable exam-

	`Native`	`Sign`	`DL`	`CCS`	`BIN`	`BCP`	`UNARY`
IBM_02_1_1_cycle_45	26.67	29.11	23.67	24.79	26.84	26.76	26.62
IBM_02_1_1_cycle_50	26.87	23.51	24.8	27.11	27.11	27.17	26.98
IBM_04_cycle_45	39.95	30.15	38.73	31.08	40.01	30.98	39.82
IBM_05_cycle_45	19.28	32.71	28.25	19.21	19.13	23.06	19.1
IBM_05_cycle_50	23.58	42.47	30.48	23.56	23.53	30.88	23.53
IBM_06_cycle_45	47.62	11.32	47.24	41.52	47.5	47.42	47.43
IBM_07_cycle_10	62.35	341.05	40.29	44.42	43.65	46.62	42.59
IBM_07_cycle_15	24.28	18.5	19.63	21.81	16.4	16.69	19.49
IBM_07_cycle_20	24.57	30.14	20.17	20.18	19.19	19.28	20.64
IBM_07_cycle_25	24.74	48.81	22.02	26.55	21.42	23.67	18.41
IBM_07_cycle_30	25.14	50.95	17.43	22.63	18.91	18.32	15.42
IBM_07_cycle_35	25.53	22.8	28.7	22.02	14.01	15.57	17.45
IBM_07_cycle_40	25.9	27.19	20.88	19.99	19.68	21.96	17.8
IBM_07_cycle_45	26.02	192.98	15.86	22.02	28.04	29.16	26.58
IBM_07_cycle_50	26.33	19.4	25.45	22.9	16.45	16.41	24
IBM_11_1_cycle_45	1140.99	1232.29	679.36	1175.36	952.89	1054.36	894.69
IBM_14_2_cycle_25	25.62	22.42	25.88	19.43	25.87	20.7	25.68
IBM_14_2_cycle_30	51.18	58.86	66.39	46.96	43.52	37.62	50.96
IBM_14_2_cycle_50	669.84	6881.23	668.32	535.39	618.71	593.9	684.73
IBM_17_1_2_cycle_40	16.23	8.89	19.14	16.25	16.19	16.17	16.3
IBM_17_1_2_cycle_45	19.38	7.79	24.37	19.39	19.92	19.37	19.32
IBM_17_1_2_cycle_50	15.98	11.64	15.77	16.15	19.23	16.03	15.29
IBM_19_cycle_35	37.96	32.8	30.09	51.42	43.29	37.8	37.71
IBM_19_cycle_40	69.01	120.82	91.02	60.07	84.79	70.69	70.41
IBM_19_cycle_45	112.26	120.35	124.43	220.46	144.64	112.29	155.17
IBM_19_cycle_50	283.95	194.38	401.87	215.21	310.72	341.76	542.8

Table 2
Run times of the adaptive solver versions on each one of the CNF instances.

ples, and an overall 1.6 speedup on the whole benchmark suite. The speedup on the sum of all run times is particularly significant in the setting of SAT-based verification, since it represents the impact Adaptive Solving can have on a whole verification project. Our results show that the overall time needed for the project can be reduced, by having some of the runs finish significantly faster and others slightly slower. The result is a verification effort that is completed in less time, and reveals bugs much faster.

We showed that the `-sign` option has an overall detrimental effect on run times. Although there are some examples for which this option is useful, when using it on all of our examples the overall run time is much larger. It is

	Native	Sign	DL	CCS	BIN	BCP	UNARY
IBM_20_cycle_25	73.59	75.19	73.32	67.84	73.88	88.68	73.86
IBM_20_cycle_30	378.49	313.49	378.6	304.09	378.23	366.73	377.86
IBM_20_cycle_40	2915.99	2193.85	2913.52	3338.03	2098.47	2612.28	3193.65
IBM_20_cycle_45	2262.22	1384.75	1584.61	4376.08	1495.49	1793.88	2104.08
IBM_20_cycle_50	5688.42	1281.89	1458.45	1075.95	3400.8	4287.91	964.08
IBM_21_cycle_35	30.56	24.51	28.6	35.64	30.45	30.47	30.38
IBM_21_cycle_40	97.71	73.52	90.42	92.78	97.43	97.39	97.09
IBM_21_cycle_45	321.15	221.58	230.48	198.86	235.65	322.9	320.48
IBM_21_cycle_50	207.1	335.21	216.49	211.32	275.71	207.17	270.92
IBM_22_cycle_35	38.34	41.92	47.13	40.96	38.24	37.13	38.18
IBM_22_cycle_40	115.97	155.49	124.63	113.16	116.41	121.43	116.21
IBM_22_cycle_45	368.1	612.17	400.4	378.86	366.91	443.05	371.93
IBM_27_cycle_45	12.83	21.72	19.23	18.31	12.73	19.67	12.68
IBM_28_cycle_30	21.22	18.12	21.11	31.1	21.21	21.31	21.19
IBM_28_cycle_40	21.72	21.41	27.08	39.31	21.74	52.79	21.73
IBM_28_cycle_45	22.25	55.86	28.96	25.27	22.29	22.26	22.14
IBM_29_cycle_15	305.51	149.21	283.99	176.02	173.99	183.16	194.88
IBM_29_cycle_20	1828.63	1792.38	1817.36	1745.12	1948.38	1542.37	1764.72
IBM_29_cycle_30	3821.98	10000	3875.55	1013.73	325.29	3505.24	859.39
IBM_29_cycle_50	673.73	10000	4015.01	647.92	663.63	815.04	758.6
IBM_new_2_cycle_20	214.05	152.21	213.85	127.8	69.35	92.03	207.56
IBM_new_2_cycle_25	916.42	1020.32	906.14	236.31	226.7	310.5	228.15
IBM_new_5_cycle_20	137.81	31.65	118.62	122.9	75.56	124.26	80.63
IBM_new_6_cycle_20	252.53	245.97	252.3	146.51	140.69	170.39	217.72

Table 3
Run times of the adaptive solver versions on each one of the CNF instances.

impossible to predict beforehand which examples will benefit from this option. For this reason it has not been used in practice until now. The Adaptive Solver is thus capable of making the best out of a heuristic that overall did not prove beneficial. Our implementation controlled only one such heuristic, but there are many others that could be used. We plan to investigate how to combine this strength on multiple options.

An interesting phenomenon is that on some examples, although the `-sign` option performs badly, using it on parts of the search space gave better results than not using it at all. This shows that different sub-spaces require different approaches, and clearly demonstrates that the potential of Adaptive Solving is greater than that of the parameters it controls.

As for the performance metrics – we continue to search for better metrics.

We have discovered that some metrics perform better on satisfiable instances, while others are better for unsatisfiable instances. This implies that a combination of metrics may be more beneficial.

The details of the adaptive algorithm need to be tuned according to the specific implementation of the solver. The organization of the database and the decision heuristics used by a solver influence the right choices for the adaptive algorithm. This means that implementing the exact same algorithm on different solvers, may yield different results.

There are many directions that require further research. Finding the best metric and parameters to use is crucial. The algorithm used to determine when to switch options has also not been perfected. Beyond this, we would also like to experiment with incorporating learning algorithms into the process.

References

[1] F. A. Aloul, B. D. Sierawski, and K. A. Sakallah. Satometer: how much have we searched? In *DAC*, pages 737–742. ACM Press, 2002.

[2] R. J. Jr. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Conf. on AI*, pages 203–208, 1997.

[3] A. Biere, A. Cimatti, E. Clark, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207, 1999.

[4] E. Carvalho and J. Marques-Silva. Using rewarding mechanisms for improving branching heuristics. In *SAT*, pages 275–280, May 2004.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7):394–397, 1962.

[6] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT solver. In *DATE*, pages 142–149, 2002.

[7] M. Herbstritt and B. Becker. Conflict-based selection of branching rules. In *SAT*, pages 441–451, May 2003.

[8] IBM HRL. IBM formal verification benchmark library, 2004. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html.

[9] IBM. Rulebase Parallel Edition, 2004. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/index.html.

[10] M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *SAT*, volume 9, June 2001.

[11] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC*, 2001.

[13] E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, and H. Hoos. Satzilla: An algorithm portfolio for sat. In *SAT competition*, 2004.

[14] E. Nudelman, K. Leyton-Brown, A. Devkar, H. Hoos, and Y. Shoham. Understanding random sat: Beyond the clauses-to-variables ratio. In *Conf. on Princ. and Prac. of Const. Prog.*, 2004.

[15] Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2002.

[16] H. Zhang. SATO: an efficient propositional prover. In *Conf. on Auto. Deduc.*, pages 272–275, 1997.

[17] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.