cmcSAT - A Cooperative MultiCore SAT Solver

Ricardo Marques Luís G. Silva Paulo Flores L. Miguel Silveira
ALGOS Group at INESC-ID Lisbon
Instituio Superior Técnico - Technical University of Lisbon
email:{rsm,lgs,pff,lms}@inesc-id.pt

INESC-ID Technical Report RT 20/2012
July 2012

***Abstract*—In this paper we present our results on exploiting multi-core shared memory architectures to improve the effectiveness of standard SAT solvers. Many problems in Electronic Design Automation (EDA) as well as many other fields can be converted to a SAT problem or a sequence of SAT problems and solved with a state of the art SAT solver. In EDA typical applications include problems in testing, timing, verification, routing, etc. Despite the enourmous progress achieved over the last decade in the development of SAT solvers, there is strong demand for higher algorithm efficiency to solve harder and larger problems. The widespread availability of multi-core, shared memory parallel environments provides an opportunity for such improvements and in this paper we present a cooperative approach to improving SAT solver efficiency and capacity. Multiple instances of the same basic solver using different heuristic strategies for search-space exploration and problem analysis share information and cooperate towards the solution of a given problem. Results from application of our methodology to known problems from SAT competitions and EDA problems show relevant improvements over the state of the art and yield the promise of further advances.**

I. Introduction

Over the last decade interest over propositional satisfiability (SAT) and SAT solvers has increased manifold. The propositional Satisfiability or SAT problem has long been one of the most studied problems in computer science since it was the first problem proven to be NP-complete. Nowadays, due to the enourmous advances in computational SAT solvers, the satisfiability problem evidences great practical importance in a wide range of disciplines. Many problems in many fields of science can be formulated as a set of SAT instances which are then amenable to analysis by state of the art SAT solvers. In Electronic Design Automation (EDA) examples include problems in hardware verification, timing analysis, optimal circuit design, FPGA routing, combinatorial equivalence checking and automatic test and pattern generation.

One of the main reasons for this increased interest in SAT is the considerable efficiency improvement that SAT solvers have undergone in the past decade. Nowadays many real-life, industrial problems with hundreds of thousands of variables and millions of clauses are routinely solved within a few minutes by off the shelf, state of the art, SAT solvers. This impressive progress can be traced back to remarkable algorithmic improvements as well as significant progress in the ability of SAT solvers to exploit the hidden structures of many practical problems. However, this increased capability is continuously challenged by emerging applications which bring to the fold instances of increasing size and complexity. As a consequence, in spite of the remarkable gains we have seen in the area, many problems remain very challenging and unsolved by even the best solvers. Perhaps the main cause for this situation is that the large, steady algorithmic improvements that were made available in the last decade with the introduction of powerful techniques such as non-chronological backtracking, restarts, improvements in decision heuristics, etc, seem to have slowed down. Improvements to SAT solvers nowadays appear to be more incremental, sometimes problem-related. Faced with this situation, researchers have started to look elsewhere for ways to continue improving the efficiency of SAT solvers and the widespread availability of parallel computing platforms provided renewed opportunities to achieve this goal. The generalization of multicore processors as well as the availability of fairly standard clustering software provided access for the common user to parallel computing environments. In this context, many parallel SAT solvers have been proposed and SAT competitions now routinely include a parallel track. The main goal of parallel SAT solvers is to improve solver efficiency, which can be viewed in absolute terms (how many problems are solved and in what time), but is generally measured in terms of efficiency or speedup obtained given the resources used. The basic obstacles to improved efficiency are generally the same ones as encountered by other parallel implementations, namely load balancing and robustness, i.e. the ability to sustain similar efficiency over a large range of problems. An enticing feature of parallelization is that in search-based problems, super-linear speedups are achievable if one is smart or lucky enough to search the right region of the search space. Unfortunately such smarts/luck are hard to ensure. Many different parallelization strategies have been attempted and the field is fairly crowded. An approach that has seen many followers is where for instance the search space is broken into non-overlapping regions which are then searched in parallel. A similar approach can be applied to the actual problem definition and tentative solutions can be sought in sub-problems consisting of partial descriptions of the problem (which must later be made consistent with the remaining problem constraints). In both cases, the parallel searches being conducted are complementary to each other and the searches are generally non-overlapping. In all such approaches, the main difficulty is related to ensuring a balanced workload between processors which is very hard to achieve. To minimize

This work was supported by the Portuguese Foundation for Science and Technology (FCT) research project Parsat - Parallel Satisfiability Algorithms and its Applications (PTDC/EIA-EIA/103532/2008) and by FCT through the PIDDAC Program funds.

potential unbalancing strategies such as workload stealing, or breaking the problem into non-overlapping searches with a finer granularity and distributing queued searches to available processors have been proposed. Unfortunately, it is in general hard to pick the most relevant set of variables for the initial breakup of the search space.

In this paper we present CMCSAT, a cooperative multi-threaded, MultiCore SAT solver which exploits a different approach to resource utilization. The general strategy pursued in CMCSAT is not novel and has in fact been previously proposed in other SAT solvers [1]. The idea is to launch multiple instances of the same (or different) solvers, sometimes called a portfolio, with different parameter configurations, which cooperate in the search for a solution by sharing relevant information. This approach has the advantage that it minimizes the dependence of current SAT solvers on the specific parameter configuration chosen to regulate their heuristic behavior, namely the decision process on the choice of variables, on when and how to restart, on how to backtrack, etc. Instead of attempting to guess the parameter configuration that better leads to the problem solution, we exploit multiple configurations in parallel, enforce some level of cooperation between them and hope that one of them, with the help of the shared information, might find a solution faster. The set of parameter configurations chosen should be such that they represent complementary ideas and strategies. Each solver instance will attempt to find a solution to the problem or prove that no solution exists (problem is unsatisfiable). To do so, it will use the information it gathers plus the information gathered by others which are concurrently attempting to find the same solution. As we will see, a clever set of multiple parameter settings may lead to speedups and the sharing leads to further improvements.

The remainder of our paper is as follows. In Section II we present some background on the basic techniques in which current SAT solvers are based. We also summarize several representative parallel approaches previously presented and try to surmise their advantages and disadvantages. Then in Section III we discuss our approach in some detail. In Section IV some preliminary results are presented, including comparisons with both serial as well as alternative parallel implementation. Finally, in Section V, conclusions are drawn regarding the proposed ideas.

II. Background

The SAT problem consists in determining if there exists an assignment to the variables of a propositional logic formula such that the formula becomes satisfied. A problem to be handled by a SAT solver is usually specified in a conjunctive normal form (CNF) formula of propositional logic. A CNF formula is represented using Boolean variables that can take the values 0 (*false*) or 1 (*true*). Clauses are disjunction of literals, which are either a variable or its complement, and a CNF formula is a conjunction of clauses.

Basic SAT solvers are based on the *Davis-Putnam-Loveland-Logemann* (DPLL) algorithm [2]. The DPLL algorithm improves over the simple assign, test and backtrack algorithm by the using two simple rules at each search step: unit propagation and pure literal elimination. The unit propagation occurs when a clause contains only a single unassigned literal (unit clause). In order to satisfy the unit clause, no choice is necessary, since the value to assign the variable is the value necessary to make the literal *true*. The pure literal elimination consists in determining if a propositional variable occurs with only one polarity in the formula. Such literals can always be assigned in a way that makes all clauses containing them *true*. Thus, these clauses do not constrain the search anymore and can be deleted

During the search process a conflict can arise when both Boolean values have been tried on a variable and the formula is not satisfied. In this situation the algorithm backtracks to the previous decision level, where some variable has yet to toggle its value. The idea to analyse the reason of the conflict led to the *conflict driven clause learning* (CDCL) algorithm [3]. Resolving the conflict implies the generation of new clauses that are learned. These learned clauses are added to the original propositional formula and can lead to a non-chronologic backtrack, where large parts of the search space are avoides since no solution can exist there. For this reason the CDCL algorithm is very effective and is the basis of most modern SAT solvers.

A. SAT Solver Techniques

In the following we provide a brief overview on the core techniques employed in modern SAT solvers.

1) Decision Heuristics: Decision heuristics play a key role in the efficiency of SAT algorithms, since they determine which areas of the search space get explored in first place. A well chosen sequence of decisions may yield a solution almost immediately, while a poorly chosen one may require the entire search space to be explored before a solution is reached. Older SAT solvers would employ *static* decision heuristics, where variable selection was only based on the problem structure. Modern SAT solvers use *dynamic* decision heuristics, where variable selection is not only based on the problem structure, but also on the current search state. The most relevant of such decision heuristics is VSIDS (Variable State Independent Decaying Sum) , introduced by CHAFF [4], whereby decision variables are ordered based on their *activity*. Each variable has an associated activity, which is increased every time that variable occurs in a recorded conflict clause. The purpose of VSIDS, and similar activity-based heuristics, is to avoid scattering the search, by directing it to the most constrained parts of the formula. These techniques are particularly effective when dealing with large problems.

2) Non-Chronological Backtracking and Clause Recording: When a conflict is identified, backtracking needs to be performed. *Chronological* backtracking simply undoes the previous decision, and associated implications, resuming the search afterwards. On the other hand, *non-chronological* backtracking, introduced by GRASP [3], can undo several decision, if they are deemed to be involved in the conflict. When a conflict is identified, a diagnosis procedure is executed, which builds a *conflict clause* encoding the origin of the conflict. That clause

is recorded (learnt), i.e. added to the problem. Backtracking is then performed, possibly undoing several decisions, until the newly-added conflict clause becomes unit (with only one free literal). While the immediate purpose of learnt clauses is to drive non-chronological backtracking, they also enable future conflicts to show up earlier, thus significantly improving performance. However, clause recording slows down propagation, since more clauses must be analyzed. Therefore, modern SAT solvers periodically remove a number of learnt clauses, deemed irrelevant by some heuristic.

3) Watched Literals: Since any SAT algorithm relies extensively on accessing and manipulating large amounts of information, its data structures are of paramount importance for its overall performance. The single most effective improvement on the data structures of SAT algorithms was the introduction of *watched literals*, as proposed by CHAFF [4]. During propagation, only unit clauses (with only one free literal) can be used to imply new variable assignments. For each clause, two free literals are selected to be watched. When a watched literal becomes false, the corresponding clause is analyzed to check whether it has become unit or if a new free literal should be watched instead of the previous one. When no other literal can be chosen to be watched this means that the clause is unit and that the remaining free literal is the other watched literal. This technique provides an efficient method to assess whether a clause has become unit or not, and to determine its free literal. One interesting advantage of this techniques is that it is not necessary to change the watched literals associated with each clause when backtrack is performed.

4) Restarts: SAT algorithms can exhibit a large variability in the time required to solve any particular problem instance. Indeed, huge performance differences can be observed when using different decision heuristics. This behavior was studied by [5] and observed that the runtime distributions for backtrack search SAT algorithms are characterized by heavy tails. Heavy tail behavior implies that, most often, the search can get stuck in a particular regions of the search space. Therefore, the introduction of *restarts* [5], [6] was proposed as a method of avoiding trashing during backtrack search. The restart strategy consists of defining a threshold value in the number of backtracks, and aborting a given run and starting a new run whenever the number of backtracks reaches that threshold value. In order to preserve the completeness of the algorithm, the backtrack threshold value must be increased after every restart, thus enabling the entire search space to be explored, after a certain number of restarts. Restarts and activity-based decision heuristics are complementary, since the first one moves the search to a new region of the search space, while the second one enables the search to be focused in that new region.

B. Parallel Approaches

Currently, parallel computing environments are an affordable reality which has forced a significant shift in the programming paradigm. This situation led many programmers to develop concurrent applications that are tuned for specific parallel environments/architectures, such as: computer clusters, multi-core processors, graphics processing units (GPUs), etc. The development of parallel SAT solvers, using parallel computing environments, are a promising approach to speed-up the search for a solution when compared to sequential SAT solvers. Moreover, parallel solvers should also be able to solve larger and more challenging problems (industrial problems) for which sequential SAT solvers are not able to find a solution in a reasonable time.

According to the parallelization technique used we can divide most parallel implementations of SAT solvers in two categories: cooperative SAT-solvers or competitive SAT-solvers. In the former, the search space is divide and each computational unit (either a core, a processor or computer) search for a solution in their sub-set of search space. In general this often leads to master-slave scheme where the workload balance between of the different computation units are difficult to achieve. In the latter, each computation unit try to solve the same SAT instance, but using alternative search paths. This is achieved by assigning different algorithms to each computation unit and/or using the same algorithm but with a different set of configuration parameters (portfolios). For this reason, this latter category is often called the portfolio SAT-solvers. In both categories the computation units can work collaboratively by sharing information about learnt clauses to speed-up the search process. This implies some sort of communication between the processing units that may introduce some overhead. Deciding which clauses to share and when to share them, may have a significant impact on the time that a parallel SAT solver takes to find a solution. Among all the parallel SAT solvers that have been developed over the past decade we present here some of the most noticeable approaches. A more complete overview related to parallel SAT solvers can be found in [7], [8] or [9].

The `PMSat` [10] solver is based on `MiniSAT` [11], which a sequential SAT solver that implements most of the techniques used on advanced solvers. The `PMSat` solver run on a cluster of computers (grid) using MPI (Message Passing Interface) for communication and it is based on the master-slave approach with a fixed number of slaves. The search space is divided by the master using several partition heuristics. Each slave search for a solution in subset of space and share the selected learned clauses when reporting the solution to the master. The load balancing is implicility implemented by providing sufficiently tasks for the slaves.

The `c-sat` [12] SAT solver is also a parallelization of `MiniSAT` that uses MPI. The master-slave architecture provides a master responsible by clause sharing and by dynamic partitioning of the search space. The slaves perform the search on a subset of the search space using different heuristics and random number seed. Therefore, this solver, which runs on a network of computers, combines the search space splitting with a portfolio of algorithms for each subspace.

The `PaMiraXT` [13] solver is another parallel SAT solver that follows the master-slave model and can run on any cluster of computers. Each slave is based on the `MiraXT` [14] solver, which is a thread-based solver. The `MiraXT` uses a divide-and-conquer approach where all threads share a unique clause

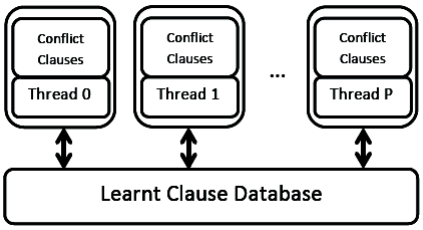

Fig. 1. Depiction of the main architecture of the cmcSAT solver.

database that includes learned clauses.

The ManySAT [1] is a portfolio-based multithread solver that won the parallel track of the SAT-race 2008 [15]. Since then the portfolio solvers become popular. In this solver, that was built on top of MiniSAT, there are four parallel instances with different restart, decision and learning heuristics. Additionally, each sequential instance share clauses, with a given threshold size, to improve the overall system performance.

The SAT4J// [8] solver is a kind of hybrid solver that mixes the competitive search and the cooperative, search, but without clause sharing. The solver, implemented in Java, starts with a portfolio approach (weak portfolio) where an heuristic based on the VSIDS determine the variables with the highest activities Then, the solver splits the search space based on those variable and when the search reaches a certain limit is switches back to a portfolio approach (full portfolio).

Cooperative SAT solvers, through the search space splitting, are one of the most used techniques to implement parallel SAT solvers. Although, recently the has been an increasing interest on the portfolio approaches, which seem to have better performance [1], [12].

III. Multicore SAT Solver - cmcSAT

The cmcSAT solver is a Cooperative MultiCore SAT solver based on portfolios. The solver has eight threads that explore the search space independently, following different paths, due to the way each thread is configured. However, this is not just a purely competitive solver because the threads cooperate by sharing the learnt clauses resulting from conflict analysis. Figure 1 presents the main architecture of the cmcSAT solver. The underlying solver running on each thread is based on the MiniSAT sequential SAT solver (version 2.2.0). The solver was however modified to support clause sharing and the ability to implement different heuristic schemes. In the following sections we briefly describe a few strategies that were adopted in the implementation of cmcSAT.

A. Heuristics

Heuristics are used on SAT solvers for selecting the next variable to be assigned, and the corresponding value, when no further propagation can be done. Although some randomness is incorporated into most heuristics, we would like to keep a tight control over the search space explored by each thread. Therefore, we introduce the notion of variable priority. Variables with higher priority are assigned before variables with lower priority. The well proven VSIDS heuristic is used as the main decision heuristic but the variable selection is constrained by the priority assigned to each variable.

TABLE I
Priority assignment schemes for each thread.

Thread	Variable priority assignment
0	All the variables have the same priority, therefore this thread mimics the original VSIDS heuristic.
1	The first half of the variables read form the file have higher priority then the second half.
2	The second half of the variables read form the file have higher priority then the first half.
3	The priority is sequential decreased as the variables are read from the file.
4	The priority is sequential increased as the variables are read from the file.
5	The priority is assigned randomly for each variable read from the file.
6	The priority is sequential increased as the variables are read from the file but its priority can be increase randomly up-to 5 times.
7	The priority is sequential increased as the variables are read from the file but its priority can be increase randomly up-to 10 times.

In order to ensure that each thread follows divergent search paths, we defined distinct priority assignment schemes, one for each thread of the cmcSAT solver. Table I describes the eight priority schemes that were used. Note that, for most industrial SAT instances we can take advantage of the fact that the variables appear in the CNF file in a particular order, which is not random, but related to the problem structure.

B. Clause Sharing

It is well known that during the search process the clauses learnt by each thread (conflict clauses), as a result of conflict analyses, are vital to speed-up its own search process. However, it turns out that the information learnt from a conflict in one particular thread can be very useful to other threads, in order to prevent the same conflict to take place. Therefore, clause sharing between threads was implemented in cmcSAT. We limit the size of the clauses to be shared, to avoid the overhead of copying large clauses, which may contain very little relevant information. In [1], the authors show that the best overall performance is achieved with a maximum size of 8 literals per clause.

To reduce the communication overhead introduced by clause sharing, and its overall impact in performance, we have designed data structures that eliminate the need for read and write locks. These structures are stored in shared memory, which is shared among all threads. We will consider that shared clauses are sent by a *source* thread and received by a *target* thread. As illustrated in Figure 2, each source thread owns a set of queues, one for each target thread, where the clauses to be shared are inserted. While this flexible structure enables sharing different clauses with different threads, we will restrict ourselves to sharing the same clauses with every thread. Therefore, every thread is a source thread and their target threads are all the others. On each queue, the *lastWrite* pointer marks the last clause to be inserted. The *lastWrite* pointer is only written by the source thread, but can be read by each target thread. On the other hand, the *lastRead* pointer which marks the last clause received by the target thread, is only manipulated by each target thread. This data structure eliminates the need for a locking mechanism, since *lastRead*

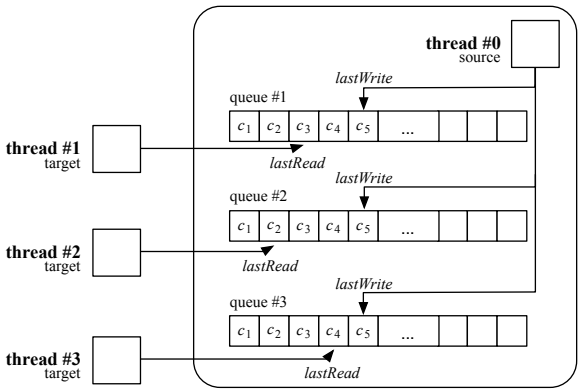

Fig. 2. Data structures to share learnt clauses.

is only manipulated by one thread and even though *lastWrite* is read and written by different threads, the reading thread does not have to read its latest value. Clause sharing can occur after a conflict analysis. If the conflict clause has less than 8 literals, then is it also shared.

IV. Evaluation Results

In this section we present preliminary results from applying CMCSAT to a slew of problems gathered from a recent SAT race. These problems have multiple origins and their complexity varies considerably. In Table II we show the characteristics of some of the SAT instances we used to test CMCSAT. These include problems which are both known to be SAT or UNSAT, as well as a couple of problems whose satisfiability is unknown.

In Table III we show experimental results for these benchmarks. The columns in the table provide runtime information for application of MiniSAT [11], ManySAT [1], as well as CMCSAT with and without clause sharing. Also shown in the table are an indication of which thread(s) first found a solution for each of the problems, whenever a solution was found. When sharing of learnt clauses is turned off, the comparison reduces to determining which of the strategies depicted in Table I is more appropriate to a given problem or set of problems. In this case, it turns out that the standard MiniSAT performs quite well but many other strategies do equally well, including random picking of variables. When sharing is turned on, the scenario changes considerably and picking chuncks of variables from the top or bottom of the order seems to do quite well on many occasions.

Finally, the table also shows four columns of speedup computations (the last four) which allow us to attest the potential gains of our solver. Here we compare CMCSAT with clause sharing against ManySAT, which is a fair comparison between two cooperative portfolio solvers which both share learnt clauses. The results are quite interesting with an average speedup of over 15. We should caution however that this is just a small subset of instances. We run the same experiments on a larger subset and the average speedup was closer to 9 to 10, still a very interesting result for MCMSAT considering that ManySAT was the SAT race winner as recently as 2008. The other three columns illustrate respectively, the advantages of

TABLE II
Benchmark Characteristics

Instance	Result	#Vars	#Clauses
aloul-chnl11-13	UNSAT	286	1742
anbul-dated-5-15-u	UNSAT	151952	684567
anbul-part-10-13-s	???	234673	1052492
babic-dspam-vc1080	UNSAT	118298	327017
babic-dspam-vc949	UNSAT	112728	356938
babic-dspam-vc973	UNSAT	274451	902703
cmu-bmc-longmult13	UNSAT	6565	20438
cmu-bmc-longmult15	UNSAT	7807	24298
een-pico-prop05-75	UNSAT	76639	245312
goldb-heqc-alu4mul	UNSAT	4736	30465
goldb-heqc-dalumul	UNSAT	9426	59991
ibm-2002-04r-k80	SAT	104450	449746
ibm-2002-11r1-k45	SAT	156626	633125
ibm-2002-22r-k75	SAT	191166	793646
ibm-2004-23-k100	SAT	207606	847320
ibm-2004-23-k80	SAT	165606	672840
ibm-2004-29-k25	UNSAT	17494	74526
ibm-2004-29-k55	SAT	37714	165426
jarvi-eq-atree-9	UNSAT	892	3006
marijn-philips	UNSAT	3641	4456
mizh-md5-47-3	SAT	65604	234719
mizh-md5-47-4	SAT	65604	234811
mizh-md5-47-5	SAT	65604	235061
mizh-sha0-35-3	SAT	48689	173748
mizh-sha0-35-4	SAT	48689	173757
mizh-sha0-36-1	SAT	50073	179811
narain-vpn-clauses-8	SAT	1461772	5687553
palac-sn7-ipc5-h16	SAT	114548	398984
simon-s02b-r4b1k1.2	SAT	2424	14812
simon-s03-w08-15	UNSAT	132555	469456
velev-npe-1.0-9dlx-b71	SAT	889302	14582952
velev-vliw-sat-4.0-b4	SAT	520721	13348116
velev-vliw-sat-4.0-b8	SAT	521179	13378616
velev-vliw-uns-2.0-iq1	UNSAT	24604	261472
velev-vliw-uns-2.0-iq2	UNSAT	44095	542252
velev-vliw-uns-2.0-uq5	???	151669	2465730

clause sharing, as well as the speedups over standard MiniSAT both when clause sharing is turned on and off. The advantages of clause sharing seem obvious: as expected, using information from other threads which are exploring problem structure elsewhere in the search space, leads to some speedup. However the advantage of this approach is also offset by the cost of doing clause sharing (both preparing clauses for sharing, as well as using clauses originally from other threads. For this reason in a reasonable number of problems no speedup is obtained. The comparisons to MiniSAT serve two purposes. When clause sharing is turned off we are, as previously mentioned, really testing the appropriateness of the strategies in Table I. When clause sharing is on, we are measuring the advantages of cooperation between threads. Either way, the speedups are interesting, with the average speedup over MiniSAT when clauses are shared being quite large (over 100 for the subset of instance shown but a more reasonable but also eye popping 35 or so. Overall, the results, seem very promising and justify further investment in adding new approaches to obtain further speedups.

V. Conclusion

The widespread availability of multi-core, shared memory parallel environments provides an opportunity for boosting the effectiveness of SAT solution. In this paper we presented a co-operative multi-core SAT solver to improve solution efficiency and capacity. Multiple instances of the same basic solver using

TABLE III
EVALUATION RESULTS

Instance	Result	T MiniSAT	Winner Thread no share	Tparallel no share	Tparallel share	Winner Thread w/ share	ManySAT	Speedup share vs ManySAT	Speedup share vs no share	Speedup no share vs MiniSAT	Speedup share vs MiniSAT
aloul-chnl11-13	UNSAT	3600.00	4	14.02	14.01	4	2426.34	173.19	1.00	256.78	256.96
anbul-dated-5-15-u	UNSAT	3600.00	0	439.00	185.51	0	79.37	0.43	2.37	8.20	19.41
anbul-part-10-13-s	???	3600.00	—	3600.00	3600.00	—	3600.00	1.00	1.00	1.00	1.00
babic-dspam-vc1080	UNSAT	647.34	3	0.33	0.33	3	46.26	140.18	1.00	1961.64	1961.64
babic-dspam-vc949	UNSAT	58.86	3	0.33	0.33	3	47.37	143.55	1.00	178.36	178.36
babic-dspam-vc973	UNSAT	10.95	3	0.75	0.75	3	2.17	2.89	1.00	14.60	14.60
cmu-bmc-longmult13	UNSAT	26.27	1.5	21.81	7.12	”0;1;4”	7.38	1.04	3.06	1.20	3.69
cmu-bmc-longmult15	UNSAT	15.60	1	13.13	6.30	0.1	5.23	0.83	2.08	1.19	2.48
een-pico-prop05-75	UNSAT	105.88	0	105.88	65.98	0	36.23	0.55	1.60	1.00	1.60
goldb-heqc-alu4mul	UNSAT	118.69	0	118.69	41.98	0.2	29.92	0.71	2.83	1.00	2.83
goldb-heqc-dalumul	UNSAT	3600.00	—	3600.00	875.00	1	244.23	0.28	4.11	1.00	4.11
ibm-2002-04r-k80	SAT	29.89	0	29.89	11.25	”0;1;2”	20.03	1.78	2.66	1.00	2.66
ibm-2002-11r1-k45	SAT	38.19	0	38.19	4.39	”0;1;6”	21.14	4.82	8.70	1.00	8.70
ibm-2002-22r-k75	SAT	251.07	1	170.56	8.37	1.6	112.03	13.38	20.38	1.47	30.00
ibm-2004-23-k100	SAT	83761.00	3	697.06	62.91	3	177.85	2.83	11.08	120.16	1331.44
ibm-2004-23-k80	SAT	232.34	0	232.34	16.31	3	87.87	5.39	14.25	1.00	14.25
ibm-2004-29-k25	UNSAT	98.99	0	98.99	34.64	0	24.05	0.69	2.86	1.00	2.86
ibm-2004-29-k55	SAT	279.83	7	28.67	17.74	0.1	25.19	1.42	1.62	9.76	15.77
jarvi-eq-atree-9	UNSAT	106.08	4	59.17	22.36	1.2	22.90	1.02	2.65	1.79	4.74
marijn-philips	UNSAT	2496.31	3	940.08	101.56	1.2	550.11	5.42	9.26	2.66	24.58
mizh-md5-47-3	SAT	265.53	0	265.53	21.20	0.1	68.23	3.22	12.53	1.00	12.53
mizh-md5-47-4	SAT	87.00	1	39.87	17.85	0.1	334.92	18.76	2.23	2.18	4.87
mizh-md5-47-5	SAT	563.58	5	129.37	53.09	1	56.83	1.07	2.44	4.36	10.62
mizh-sha0-35-3	SAT	29.36	1	6.00	5.52	1.3	36.72	6.65	1.09	4.89	5.32
mizh-sha0-35-4	SAT	262.22	5	68.24	17.55	1	47.27	2.69	3.89	3.84	14.94
mizh-sha0-36-1	SAT	353.95	5	88.99	10.46	1.3	339.40	32.45	8.51	3.98	33.84
narain-vpn-clauses-8	SAT	3600.00	-	3600.00	373.91	0	629.53	1.68	9.63	1.00	9.63
palac-sn7-ipc5-h16	SAT	938.23	5	845.45	81.70	7	178.43	2.18	10.35	1.11	11.48
simon-s02b-r4b1k1.2	SAT	96.80	5	74.32	5.73	0	11.37	1.98	12.97	1.30	16.89
simon-s03-w08-15	UNSAT	121.27	0	121.27	33.04	1.2	22.26	0.67	3.67	1.00	3.67
velev-npe-1.0-9dlx-b71	SAT	190.91	7	37.49	15.49	1	268.84	17.36	2.42	5.09	12.32
velev-vliw-sat-4.0-b4	SAT	72.90	1	48.03	9.03	”0;1;6;7”	30.98	3.43	5.32	1.52	8.07
velev-vliw-sat-4.0-b8	SAT	101.53	1	32.05	21.55	0.1	42.41	1.97	1.49	3.17	4.71
velev-vliw-uns-2.0-iq1	UNSAT	435.23	2	256.55	41.77	2.4	789.34	18.90	6.14	1.70	10.42
velev-vliw-uns-2.0-iq2	UNSAT	3600.00	2	959.62	416.14	2.4	3600.00	8.65	2.31	3.75	8.65
velev-vliw-uns-2.0-uq5	???	3600.00	—	3600.00	3600.00	—	3600.00	1.00	1.00	1.00	1.00

different heuristic strategies for search-space exploration and problem analysis share information and cooperate towards the solution of a given problem. Results from application of our methodology to known problems from SAT competitions and EDA problems show relevant improvements over the state of the art and yield the promise of further advances.

REFERENCES

[1] Y. Hamadi and L. Sais, “ManySAT: a parallel SAT solver,” *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 6, 2009.

[2] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving,” *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, July 1962.

[3] J. a. P. M. Silva and K. A. Sakallah, “Grasp: A new search algorithm for satisfiability,” in *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, California, United States, 1996, pp. 220–227.

[4] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, Las Vegas, Nevada, USA, June 2001, pp. 530–535.

[5] C. P. Gomes, B. Selman, and H. Kautz, “Boosting combinatorial search through randomization,” in *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI)*, Madison, Wisconsin, USA, 1998, pp. 431–437.

[6] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman, “Dynamic restart policies,” in *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI)*, Edmonton, Alberta, Canada, 2002, pp. 674–681.

[7] D. Singer, *Parallel Combinatorial Optimization*. John Wiley & Sons, Inc, 2006, ch. Parallel Resolution of the Satisfiability Problem: A Survey.

[8] R. Martins, V. Manquinho, and I. Lynce, “Improving search space splitting for parallel sat solving,” in *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, Oct., 27–29, 2010, pp. 336–343.

[9] S. Holldolber, N. Manthey, V. H. Nguyen, J. Stecklina, and P. Steinke, “A short overview on modern parallel sat-solvers,” in *International Conference on Advanced Computer Science and Information System (ICACSIS)*, Dec., 17–18, 2011, pp. 201–206.

[10] L. Gil, P. Flores, and L. M. Silveira, “PMSat: a parallel version of MiniSAT,” *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 6, pp. 71–98, 2008.

[11] N. Eén and N. Sörensson, “An Extensible SAT-solver,” *Theory and Applications of Satisfiability Testing*, vol. 2919, pp. 502–518, 2004.

[12] K. Ohmura and K. Ueda, “c-sat: A parallel SAT solver for clusters,” in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed. Springer Berlin / Heidelberg, 2009, vol. 5584, pp. 524–537.

[13] B. B. T. Schubert, M. Lewis, “Pamiraxt: Parallel sat solving with threads and message passing,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 203–222, 2009.

[14] T. Schubert, M. Lewis, and B. Becker, “Pamira - a parallel sat solver with knowledge sharing,” in *Proceedings of International Workshop on Microprocessor Test and Verification*. IEEE Computer Society, 2005, pp. 29–36.

[15] C. Sinz and et al, “Sat-race 2008,” http://baldur.iti.uka.de/sat-race-2008/index.html, 2008, accessed on May 2012.