Improving SAT Solver Efficiency using a Cooperative Multicore Approach

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 28/2012
October 2012

***Abstract*—Many problems in Computer-Aided Design of Electronic Systems are addressed by converting them to a sequence of SAT problems solved with a state of the art SAT solver. 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. In this paper we present our results on improving the effectiveness of standard SAT solvers on such architectures, through a portfolio approach. 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 in CAD of electronic systems.**

I. Introduction

Propositional Satisfiability (SAT) is the workhorse for many tasks in Computer-Aided Design of Electronic Systems. Typical applications include problems in hardware verification [1], timing analysis [2], optimal circuit design, FPGA routing [3], combinational equivalence checking [4] and automatic test pattern generation [5]. Each problem instance of a particular domain is converted into a SAT instance, or sequence of SAT instances, and solved using a state-of-the-art SAT solver. One of the main reasons for the increased interest in SAT is the considerable efficiency improvements that SAT solvers have undergone in the past decade [6], [7]. This progress can be traced back to remarkable algorithmic improvements as well as significant progress in the ability of SAT solvers to exploit the hidden structure of many practical problems. However, this added capability is continuously challenged by emerging applications which bring to the fold problems and systems of increasing size and complexity. As a consequence, many problems remain very challenging and unsolved by even the best solvers.

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 and has opened up new opportunities for improvement in many areas. The main goal of parallel SAT solvers is to be able to solve problems faster. Several metrics can be used to quantify the resulting improvements, including speedup and efficiency. An enticing feature of parallelization in search-based problems is that super-linear speedups are achievable if one is lucky to search the right region of the search space. Often, one is satisfied with the ability to demonstrate speedup by solving problems faster and doing so in a robust manner over a large range of problems. Combining such robustness with the elusive possibility of large gains is clearly a goal worth pursuing.

In this paper we present CMCSAT, a portfolio-based multi-threaded, MultiCore SAT solver which exploits a different approach to resource utilization. The general strategy pursued in CMCSAT is not entirely novel and has in fact been previously proposed in other SAT solvers [8]. The idea is to launch multiple instances of the same (or different) solvers, sometimes called a portfolio, with different parameter configurations, which cooperate to a certain degree by sharing relevant information when searching for a solution. For instance for a routing problem, different options can be made by the various instances and relevant information regarding for instance obstructions, can be shared. This approach minimizes the dependence of current SAT solvers on 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 optimal parameter configuration we exploit multiple configurations in parallel and hope that one of them, with the help of the shared information, might find a solution faster. Each solver instance will attempt to find a solution to the problem or prove that no solution exists. To do so, it will use the information it gathers plus the information gathered and shared by others which are concurrently attempting to find the same solution.

The remainder of this paper is organized as follows. Section II reviews core SAT solver techniques and parallelization strategies. Section III details the proposed multicore SAT approach. Section IV presents experimental results, including comparisons with serial and other parallel implementations. Finally, Section V presents a few concluding remarks.

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

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.

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 [9], which 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 no longer constrain the search 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 [10]. 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 avoided 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. Using SAT for CAD of Electronic Systems

SAT formulations of EDA-related problems usually consist of two sets of constraints involving the logic values of circuit nodes. On one hand the circuit constraints encode its structure and enforce for every gate, relations between the logic values of input/output nodes and the gate function. The problem constraints on the other hand encode the property or set of properties to be verified.

B. 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 first. A well chosen sequence of decisions may instantly yield a solution, while a poorly chosen one may require the entire search space to be explored before a solution is reached. 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 [6], 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.

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 [10], can undo several decisions, 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. 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 and must therefore be carefully monitored.

3) Restarts: SAT algorithms can exhibit a large variability in the time required to solve any particular problem instance. Often, the search can get stuck in a region of the search space. The introduction of *restarts* [11] was proposed as a method of minimizing the effects of this problem. 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 that threshold value is reached. Randomizatin must also be incorporated into the decision heuristics, to avoid the same sequence of decisions to be taken on every run.

C. Parallel Approaches

Usage of parallel computing environments, is 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. Parallel implementations of SAT solvers can be divided in two categories: cooperative or competitive SAT-solvers. In the former, the search space is divided and each computational unit (either a core, a processor or computer) searches for a solution in their sub-set of search space. Often the workload balance between the different units is difficult to ensure. In the latter, each computational unit tries to solve the same SAT instance, but using alternative search paths. This is achieved by assigning different algorithms to each 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 portfolio SAT-solution. In both categories the computational 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 overhead. Deciding which clauses to share and when to share them, may have a significant impact on the time a parallel SAT solver takes to find a solution. A long and detailed list of parallel SAT solvers can be obtained from the parallel tracks of recent SAT competitions [12]. A significant example is MANYSAT [8], a portfolio-based multithread solver that won the parallel track of the SAT Race 2008 [12]. Since then the portfolio solvers became 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. Cooperative SAT solvers, through search space splitting, are one of the most used techniques to implement parallel SAT solvers. Although, recently there has been an increasing interest on the portfolio approaches, which have been shown to have very good performance [8], [13].

III. MULTICORE SAT SOLVER - CMCSAT

CMCSAT is a MultiCore SAT solver based on portfolios. The solver uses multiple threads (eight currently) that explore

TABLE I
PRIORITY ASSIGNMENT SCHEMES FOR EACH THREAD.

Thread #	Variable Priority Assignment Scheme
0	All the variables have the same priority, therefore this thread mimics the original VSIDS heuristic.
1	The first half of the variables read from the file have higher priority than the second half.
2	The second half of the variables read from the file have higher priority than the first half.
3	The priority is sequentially decreased as the variables are read from the file.
4	The priority is sequentially 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 sequentially increased as the variables are read from the file, but it has a random component which can yield a priority increase of up-to 5x.
7	The priority is sequentially increased as the variables are read from the file, but it has a random component which can yield a priority increase of up-to 10x.

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. The underlying solver running on each thread is based on the MINISAT sequential SAT solver (version 2.2.0) [7]. The solver was however modified to support clause sharing and the ability to implement different heuristic schemes. In the following we briefly describe a few strategies that were adopted in the implementation of CMCSAT.

A. Decision 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.

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. Lockless Clause Sharing

It is known that clauses learnt during the search process as a result of conflict analyses, are vital to speed-up the search process. In our approach 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 [8], 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 designed data structures that

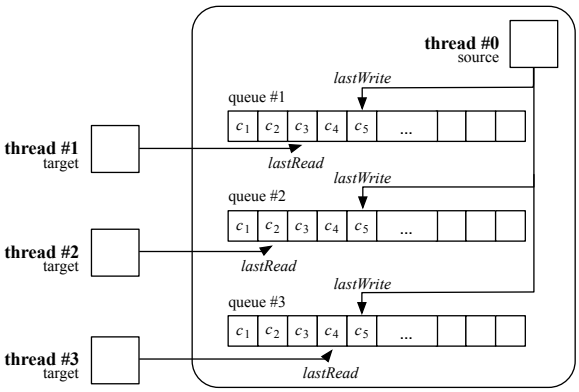

Fig. 1. Data structures to share learnt clauses.

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 1, 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* 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.

IV. EVALUATION RESULTS

In this section we present preliminary results from applying CMCSAT to a slew of problems gathered from a recent SAT race [12]. We pay particular attention to those originated from circuit examples, which are shown in Table II. These include problems which are both known to be SAT or UNSAT. We run additional examples which for lack of space are ommitted but are accounted for on the averages shown in the last line of Table II. In order, the columns of the table provide information about the problems chosen followed by runtimes for MINISAT [7] and CMCSAT using clause sharing. Also shown in the table are an indication of which thread(s) of CMCSAT first found a solution for each of the problems. Next we show speedup computations to attest the potential gains of our solver. First we compare CMCSAT with clause sharing versus non-clause sharing. When clause sharing is turned off we are really testing the appropriateness of the strategies in Table I. When clause sharing is on, we are measuring the advantages of cooperation between threads. The advantages of clause sharing seem obvious: using information from other threads which are exploring problem structure elsewhere in the search space, provides relevant information and speeds up problem solution. However the advantage of this approach is also offset by the cost of doing the sharing (both preparing clauses for sharing, as well as using clauses originally from other threads). For this reason in certain problems little speedup is obtained. Next we compare CMCSAT with clause sharing versus the serial

TABLE II
EVALUATION RESULTS

Instance	Sol.	#Vars	#Clauses	MINISAT (sec)	CMCSAT (sec)	Winning Thread	Speedup CMCSAT vs no sh	Speedup CMCSAT vs MINISAT	MANYSAT (sec)	PLINGELING (sec)	Speedup CMCSAT vs MANYSAT	Speedup CMCSAT vs PLINGELING
cmu-bmc-barrel6	U	2306	8931	1.32	0.46	0,3,5,6	2.04	2.87	0.47	0.22	1.02	0.47
cmu-bmc-longmult13	U	6565	20438	26.27	7.12	0,1,4	3.06	3.69	7.38	12.78	1.04	1.79
cmu-bmc-longmult15	U	7807	24298	15.60	6.30	0,1	2.08	2.48	5.23	10.84	0.83	1.72
ibm-2002-11r1-k45	S	156626	633125	38.19	4.39	0,1,6	8.70	8.70	21.14	22.47	4.82	5.12
ibm-2002-18r-k90	S	175216	717086	102.30	56.71	1,6	1.80	1.80	82.33	71.51	1.45	1.26
ibm-2002-20r-k75	S	151202	619733	166.08	128.10	0	1.30	1.30	107.25	52.55	0.84	0.41
ibm-2002-22r-k60	U	208590	845248	716.53	554.06	0,2	1.29	1.29	187.18	118.68	0.34	0.21
ibm-2002-22r-k75	S	191166	793646	251.07	8.37	1,6	20.38	30.00	112.03	87.37	13.38	10.44
ibm-2002-22r-k80	S	203961	846921	159.03	19.74	1	2.90	8.06	133.66	119.83	6.77	6.07
ibm-2002-23r-k90	S	222291	922916	680.85	234.78	0	2.90	2.90	158.94	212.08	0.68	0.90
ibm-2002-24r3-k100	U	148043	545315	202.90	104.05	0	1.44	1.95	95.15	74.52	0.91	0.72
ibm-2002-30r-k85	S	181484	888663	850.73	823.43	0	1.03	1.03	248.12	224.22	0.30	0.27
ibm-2004-1_11-k80	S	262808	1023506	145.46	90.96	0,1	1.43	1.60	126.74	51.12	1.39	0.56
ibm-2004-23-k100	S	207606	847320	837.61	62.91	3	11.08	13.31	177.85	89.17	2.83	1.42
ibm-2004-23-k80	S	165606	672840	232.34	16.31	3	14.25	14.25	87.87	147.61	5.39	9.05
ibm-2004-29-k25	U	17494	74526	98.99	34.64	0	2.86	2.86	24.05	27.43	0.69	0.79
mizh-md5-47-3	S	65604	234719	265.53	21.20	0,1	12.53	12.53	68.23	55.71	3.22	2.63
mizh-md5-47-4	S	65604	234811	87.00	17.85	0,1	2.23	4.87	334.92	61.24	18.76	3.43
mizh-md5-47-5	S	65604	235061	563.58	53.09	1	2.44	10.62	56.83	34.39	1.07	0.65
mizh-md5-48-5	S	66892	240181	312.49	30.16	0	6.29	10.36	367.53	42.29	12.19	1.40
mizh-sha0-35-3	S	48689	173748	29.36	5.52	1,3	1.09	5.32	36.72	14.50	6.65	2.63
mizh-sha0-35-4	S	48689	173757	262.22	17.55	1	3.89	14.94	47.27	21.75	2.69	1.24
mizh-sha0-36-1	S	50073	179811	353.95	10.46	1,3	8.51	33.84	339.40	42.22	32.45	4.04
mizh-sha0-36-4	S	50073	179989	217.09	131.42	3	1.65	1.65	733.75	22.26	5.58	0.17
velev-engi-uns-1.0-4nd	U	7000	67553	10.83	4.89	0,1	2.09	2.21	7.25	14.86	1.48	3.04
velev-fvp-sat-3.0-b18	S	35853	1012240	27.05	3.03	0,1	8.94	8.94	2.86	4.59	0.95	1.52
velev-npe-1.0-9dlx-b71	S	889302	14582952	190.91	15.49	1	2.42	12.32	268.84	37.84	17.36	2.44
velev-vliw-sat-4.0-b4	S	520721	13348116	72.90	9.03	0,1,6,7	5.32	8.07	30.98	72.51	3.43	8.03
velev-vliw-sat-4.0-b8	S	521179	13378616	101.53	21.55	0,1	1.49	4.71	42.41	60.81	1.97	2.82
velev-vliw-uns-2.0-iq1	U	24604	261472	435.23	41.77	2,4	6.14	10.42	789.34	17.52	18.90	0.42
velev-vliw-uns-2.0-iq2	U	44095	542252	TO	416.14	2,4	2.31	NA	TO	303.33	NA	0.73
avg. table set (30 inst.)	–	155385	1790335	251.34	211.23	–	4.82	8.19	158.43	69.19	5.81	2.53
avg. full set (78 inst.)	–	168240	1119370	309.48	128.87	–	3.24	37.45	182.56	102.04	9.47	2.48

MINISAT. In general the speedups are interesting, with the average speedup larger than 8 (excellent efficiency) for the instances shown and an even larger speedup (over 37) for all instances. This really shows how sensitive problem solution is to variable ordering and to which portion of the space is searched. Nex we show runtimes for MANYSAT [8] and plingeling [14], solver using approaches similar to CMCSAT, followed by the speedup comparisons. Again the results are quite interesting with good average speedups reported. These speedups clearly indicate that, as expected, there is substantial structure to be found on circuit descriptions and variable ordering. If this structural information can be obtained or guessed from the circuit itself, then a better ordering of variables during SAT solution can lead to relevant gains in problem solution. 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 portfolio-based multi-core SAT solver to improve solution efficiency and capacity. Multiple instances of the same basic solver using different heuristic strategies for search-space exploration and problem analysis share information and co-operate 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] M. N. Velev and R. E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," in *Proceedings of DAC*, 2001, pp. 226–231.

[2] L. G. e Silva, J. P. M. Silva, L. M. Silveira, and K. A. Sakallah, "Satisfiability Models and Algorithms for Circuit Delay Computation," *ACM TODAES*, vol. 7, no. 1, pp. 137–158, 2002.

[3] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar, "A New FPGA Detailed Routing Approach Via Search-Based Boolean Satisfiability," *IEEE TCAD*, vol. 21, no. 6, pp. 674–684, 2002.

[4] J. P. M. Silva and T. Glass, "Combinational Equivalence Checking Using Satisfiability and Recursive Learning," in *Proceedings of DATE*, 1999, pp. 145–149.

[5] H. Chen and J. Marques-Silva, "TG-Pro: A SAT-based ATPG System," *JSAT*, vol. 8, no. 1, pp. 83–88, January 2012.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of DAC*, Las Vegas, Nevada, USA, June 2001, pp. 530–535.

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

[8] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: A Parallel SAT Solver," *JSAT*, vol. 6, 2009.

[9] 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.

[10] J. P. M. Silva and K. A. Sakallah, "GRASP: A New Search Algorithm for Satisfiability," in *Proceedings of ICCAD*, San Jose, California, United States, 1996, pp. 220–227.

[11] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman, "Dynamic Restart Policies," in *Proceedings of AAAI*, Edmonton, Alberta, Canada, 2002, pp. 674–681.

[12] C. Sinz and et al, "SAT Race 2008," http://baldur.iti.uka.de/sat-race-2008/index.html, 2008, accessed on May 2012.

[13] K. Ohmura and K. Ueda, "c-sat: A Parallel SAT Solver for Clusters," in *Theory and Applications of Satisfiability Testing*, ser. LNCS, O. Kullmann, Ed. Springer Berlin / Heidelberg, 2009, vol. 5584, pp. 524–537.

[14] A. Biere, "Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010," FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 10/1, August 2010.