Concurrent Cube-and-Conquer

Peter van der Tak
Delft University of Technology,
The Netherlands

Marijn J.H. Heule
Delft University of Technology,
The Netherlands

Armin Biere
Johannes Kepler University Linz,
Austria

I. Introduction

The concurrent cube-and-conquer (CCC) solver implements the ideas in the paper we submitted to the PoS 2012 workshop [1]. This system description describes the main concepts, a more detailed explanation is in the paper.

Recent work has introduced the cube-and-conquer (CC) technique [2], which first partitions the search space into disjunctive sets of assumptions (cubes) using a lookahead (LA) solver (the cube phase) and then solves each cube using a CDCL solver (the conquer phase). It uses a *cutoff heuristic* to control after what number of decisions the lookahead solver should be cut off and store its decision variables (its current cube) for the CDCL solver to solve in the conquer phase. However, this heuristic is not ideal particularly because no information about the performance of CDCL on the cubes is present in the cube phase. Concurrent cube-and-conquer uses a synchronized LA and CDCL solver concurrently in the cube phase to improve the cutoff heuristic.

II. Motivation

Cube-and-conquer shows strong performance on several hard application benchmarks [2], beating both the lookahead and CDCL solvers that were used for the cube and conquer phases respectively. However, on many other instances, either lookahead or CDCL outperforms CC. We observed that for benchmarks for which CC has relatively weak performance, two important assumptions regarding the foundations of CC do not hold in general.

First, in order for CC to perform well, lookahead heuristics must be able to split the search space into cubes that, combined, take less time for the conquer solver (CDCL) to solve. Otherwise, cube-and-conquer techniques are ineffective and CDCL would be the preferred solving technique. Second, if lookahead can refute a cube, then this must mean that nearby cubes can be efficiently solved using CDCL. When this assumption fails, the cube phase either generates too few cubes and leaves a potential performance gain unused, or generates too many cubes because cubes with fewer decisions are also easy for CDCL to solve.

CCC solves these problems separately. The first by predicting on which instances cube-and-conquer techniques are ineffective and aborting in favor of a classical CDCL search. The second by also using a CDCL solver in the cube phase to better estimate the performance of CDCL on nearby cubes. This naturally cuts off easy cubes. We first discuss CCC_∞, a simplified version of CCC with no cut off heuristic and prediction in the next section, and add these two features in sections IV and V respectively. The submitted solver includes all features.

III. Concurrent cube-and-conquer

CCC_∞ is implemented by sending messages between the CDCL and the lookahead solvers using two queues: the decision queue Q_{decision} and the result queue Q_{solved}. Whenever the lookahead solver assigns a decision variable, it pushes the tuple $\langle$cube c_{id}, literal l_{dec}, *backtrackLevel*$\rangle$ comprising a uniquely allocated id, the decision literal, and the number of previously assigned decision variables (*backtrackLevel*). When the CDCL solver reads the new decision from the queue, it already knows all previous decision literals, and only needs to backtrack to the *backtrackLevel* and add l_{dec} as an assumption to start solving c_{id}. The id is used to identify the newly created cube.

If the CDCL solver proves unsatisfiability of a cube before it receives another decision, it pushes the c_{id} of the refuted cube to Q_{solved}. The solver then continues with the parent cube, by backtracking to the level where all but the last decision literal were assigned. When the lookahead solver reads the c_{id} from Q_{solved}, it backtracks to the level just above this cube's last decision variable and continues its search as if it proved unsatisfiability of the cube by itself.

To keep track of the cubes that are pending to be solved, both solvers keep the trail of decision literals (or assumptions for the CDCL solver) and the id's of the cubes up to and including each decision literal (or assumption). Whenever either solver proves unsatisfiability of the empty cube, or when it finds a satisfying assignment, the other solver is aborted.

The submitted version of CCC first simplifies the instance using Lingeling, and then uses march_rw [3] (LA) and MiniSAT 2.2 [4] (CDCL) concurrently. The CCCeq version runs march_rw with *equivalence reasoning* [3] enabled, CCCneq with *equivalence reasoning* disabled, as this has shown to affect the performance of CCC.

IV. Cutoff heuristic

One advantage of CC was that the conquer phase can be parallelized efficiently by using multiple CDCL solvers in parallel, each solving a single cube. With CCC_∞ this is no longer possible, since the lookahead solver will continue with a single branch until it is solved by either CDCL or lookahead. Additionally, CCC_∞ always uses twice as much CPU time as

wall clock time, because the lookahead and CDCL solver run in parallel.

To reduce this wasted resource utilization and allow for parallelization of the CDCL solver, we reintroduce the conquer phase by applying a suitable cutoff heuristic. As with CC, we pass cubes from the cube phase to the conquer phase using the iCNF[1] format (via the filesystem, unlike the shared memory queues in the cube phase), which is basically a concatenation of the original formula F and the generated cubes as assumptions. An incremental SAT solver iterates over each cube c_{id} in the file, and solves $F \wedge c_{id}$ until a solution is found or all cubes have been refuted.

The cutoff heuristic of CC is based on a rough prediction of the performance of CDCL on a cube. Given a cube c_{id}, it computes its difficulty[2][3] $d(c_{id}) := |\varphi_{\mathrm{dec}}|^2 \cdot (|\varphi_{\mathrm{dec}}| + |\varphi_{\mathrm{imp}}|)/n$, where $|\varphi_{\mathrm{dec}}|$ and $|\varphi_{\mathrm{imp}}|$ are the number of decision and implied variables respectively, and n is the total number of free variables. If $d(c_{id})$ is high, the CDCL solver is expected to solve c_{id} fast.

The cutoff heuristic in CC focuses on identifying cubes that are easy for CDCL to solve. It cuts off a branch if $d(c_{id})$ exceeds a dynamic threshold value t_{cc}. Initially $t_{cc} = 1000$, and it is multiplied by 0.7 whenever lookahead solves a cube (because it assumes that CDCL would have solved this cube faster) or when the number of decisions becomes too high (to avoid generating too many cubes). It is incremented by 5% at every decision to avoid the value from dropping too low.

For CCC, the same heuristic does not work because easy cubes are solved quickly by the CDCL solver. This makes the threshold very unstable so that it quickly converges to 0 or infinity depending on the instance. We therefore use a different heuristic, but using the same difficulty metric $d(c_{id})$.

Easy cubes can be detected better by CCC than by CC, because CCC can detect for which cubes CDCL finds a solution before the lookahead solver does. CCC would ideally cut off these cubes so that they can be solved in parallel. The contrary goes for when the lookahead solver solves a cube: it then seems that lookahead contributes to the search, which means that it is not desirable to cut off.

CCC uses the same difficulty metric $d(c_{id})$ as CC, but a different heuristic for determining the threshold value t_{ccc}. If a cube c_{id} is solved by CDCL, the value is updated towards $s := 0.4 \cdot d(c_{id})$, whereas it is updated towards $s := 3 \cdot d(c_{id})$ if c_{id} was solved by lookahead. To avoid too sudden changes, t_{ccc} is not changed to s directly but is filtered by $t'_{ccc} := 0.4 \cdot s + 0.6 \cdot t_{ccc}$. To furthermore avoid the threshold from dropping too low, it is incremented for every cube that is cut off.

The submitted implementation of CCC uses iLingeling to solve the cubes that were cut off by the heuristic. iLingeling basically submits these cubes to a number of independent incremental Lingeling solvers in parallel.

[1] http://users.ics.tkk.fi/swiering/icnf

[2] CC's heuristic has been improved slightly since it was initially published [2]; it now uses $|\varphi_{\mathrm{dec}}|^2$ instead of $|\varphi_{\mathrm{dec}}|$.

[3] The notation is ours.

V. Prediction

Since cube-and-conquer techniques do not work well on all instances, CCC aims to detect quickly if an instance is unsuitable. It does this based on two measurements.

First, lookahead techniques appear effective if they can solve some cubes faster than CDCL. While running the lookahead and CDCL solver in parallel, we count the number of times that lookahead is faster than CDCL. For benchmarks for which this count is increased very slowly, say less than once per second, we observed that CC was generally not an effective solving strategy.

Second, if the variable heuristics are effective then each discrepancy should result in a large reduction of the formula. Hence after a certain number of discrepancies the solver should be able to refute that branch. Preliminary experiments suggest that if CCC finds a leaf with over 20 discrepancies early in the search-tree, then lookahead variable heuristics should be considered as ineffective.

These metrics are combined as follows. CCC runs the LA and CDCL solver for a few seconds concurrently. If the LA solver enters a branch with more than 20 discrepancies terminate the solvers and use fallback solver pLingeling instead. If after 5 seconds the solvers are still running and less than 10 cubes were solved by lookahead, the solvers should also be terminated in favor of pLingeling. Otherwise, CCC is the preferred solving method and the solvers can continue. For CC, the same instances usually work well, but they cannot be detected as early because CDCL is only used in the conquer phase.

VI. Conclusion

Without performance prediction, cube-and-conquer techniques are not competitive with current state-of-the-art solvers. CCC's predictor is able to efficiently select instances for which cube-and-conquer techniques are suitable and fall back to pLingeling if not. This allows cube-and-conquer to compete with other solvers. In addition, CCC improves over CC's performance by using concurrency and improved heuristics in the cube phase.

CCC uses march_rw (LA) and the same versions of Lingeling (simplification and CDCL), iLingeling (conquer), and pLingeling (fallback) submitted to this SAT challenge. All sources are compiled into a single binary with -O3. Threading is implemented using pthreads, and communication in the cube phase using lockless queues. Communication between the simplification, cube, conquer, and fallback solvers is done via temporary CNF and iCNF files.

References

[1] P. van der Tak, M. J. H. Heule, and A. Biere, "Concurrent cube-and-conquer," 2012, submitted to PoS 2012.

[2] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," 2011, accepted for HVC.

[3] M. J. H. Heule, "Smart solving: Tools and techniques for satisfiability solvers," Ph.D. dissertation, Delft University of Technology, 2008.

[4] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT'03*, ser. LNCS, vol. 2919. Springer, 2004, pp. 502–518.