CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020

Armin Biere Katalin Fazekas Mathias Fleury Maximilian Heisinger
Institute for Formal Models and Verification
Johannes Kepler University Linz

***Abstract*—This system description describes our new SAT solver Kissat, how it differs from CaDiCaL, as well as changes made to CaDiCaL. We further present our new distributed cube-and-conquer solver Paracooba. Previous parallel SAT solvers Plingeling and Treengeling in essence remain unchanged.**

I. CaDiCaL

Compared to the 2019 version of CaDiCaL [1], we have improved inprocessing by implementing conditioning [2]. However, this feature does not seem to improve performance and is not enabled by default ("`--condition=true`"). The major difference is the implementation of a three-tier system [3] to decide which clauses should be kept during clause reduction: tier-0 clauses (LBD ≤ 2) are kept forever, tier-1 clauses ($2 <$ LBD ≤ 6) survive one round of reduction, whereas tier-2 clauses can be deleted immediately. In any case, clauses used since the last reduction are not deleted.

Moreover, even though CaDiCaL was designed to allow up to `INT_MAX` $= 2^{31} - 1 = 2\,147\,483\,647$ variables, represented with the `int` type of C++, it had a serious flaw because the idiom "`for (int i = 1; i <= max_var; i++)`" was used throughout the code. This lead to undefined behaviour if `INT_MAX` variables are used even though it works fine for fewer. To avoid complicated iteration code and also to avoid such issues in the future we implemented variable and literal iterators, used as in "`for (auto idx : vars)`" or as in "`for (auto lit : lits)`". We would like to thank Håkan Hjort for bringing this issue to our attention.

II. Kissat

Experiments with large formulas, such as the DIMACS formula "`p cnf 2147483647 0`" resulted in the following observations. Even though CaDiCaL can handle formulas with `INT_MAX` variables, it needs a substantial amount of main memory (more than 512 GB) as well as long time for initialization. One reason is using the C++ container "`std::vector`" for most data structures (e.g., to hold flags, values, decision levels, reasons, scores). They are also mostly zero initialized. Instead, we now use the C memory allocator "`calloc`". It provides zero initialization on-demand by the virtual memory system and reduces resident set size accordingly.

This design decision also raised the question, whether we can reuse some other features of Lingeling [4] to further reduce memory. In Kissat we therefore completely inline binary clauses in watcher stacks to reduce the size of watches from 16 bytes in CaDiCaL to 4 bytes for binary and 8 bytes for large clauses (due to the blocking literal). This in turn requires to use 4-byte offsets instead of pointers to reference large (non-binary) clauses. Note that binary clauses were still allocated in CaDiCaL in the memory arena holding clauses in the same way as larger clauses. In Kissat they now really only exist in watcher lists. Lingeling even inlined ternary clauses which we consider less useful now.

We also revisited the data structure for holding watches (watched lists). In Kissat we use a dedicated implementation of stacks of watchers, requiring only two offsets (of together 8 bytes in the compact competition configuration) instead of 3 pointers (requiring 24 bytes on a 64-bit architecture). This became possible by assuming that the all-bits-one word is not a legal watch and free memory in the watcher stack arena is marked with all-bits-one words. Pushing a watch on a watcher stack requires checking whether the word after the top element is illegal (all-bits-one). If so, it is overwritten. Otherwise the whole stack is moved to the end of the allocated part in the watcher arena. This produces the overhead that once in a while the watcher arena requires defragmentation and is usually performed after collecting redundant clauses in "reduce".

In order to distinguish binary and large watches in watcher stacks, we use bit-stuffing as in Lingeling. This leaves effectively 31 bits to reference large clauses. Since these large clauses are allocated 8-byte aligned in the clause arena, the maximum size of this arena is $8 \cdot 2^{31}$ bytes (16 GB). Note that in practice many large CNFs consist mostly of binary clauses, which due to inlining do not require any space in this arena. Further, beside data structures for variables, watch lists occupy a large fraction of the overall memory. Actually the largest CNFs we ever encountered in applications easily stay below this limit while in total Kissat reaches 100 GB memory usage. On top of that, other solvers including CaDiCaL often need more than 4 times more main memory than Kissat.

Due to inlining binary clauses redundant and irredundant binary clauses have to be distinguished [5], which requires an-

Supported by Austrian Science Fund (FWF) projects W1255-N23 and S11408-N23, by the LIT AI and LIT Secure and Correct Systems Labs and the LIT project LOGTECHEDU all three funded by the State of Upper Austria.

other watcher bit ("`redundant`"). Finally, as in CADICAL, hyper binary resolvents are generated in vasts amounts [6] during failed literal probing and vivification [7] and have to be recycled quite aggressively. To mark these hyper binary resolvents we need a third watcher bit ("`hyper`") and the effective number of bits for literals is reduced to 29. Thus, the solver can only handle $268\,435\,455 = 2^{28} - 1$ variables.

In "dense mode" (during for instance variable elimination) the solver maintains full occurrence lists for all irredundant clauses. In the default "sparse mode" (during search) only two literals per large clause are watched and large clause watches have an additional blocking literal. Thus, as in LINGELING, watch sizes vary between one and two words, whichs lead to very cumbersome and verbose watch list traversal code in LINGELING repeated all over the source code. For KISSAT we were able to almost completely encapsulate this complexity using macros. The resulting code resembles ranged-based `for` loops in C++11 as introduced in CADICAL last year.

These improved data structures described above obviously require too many changes and we decided to start over with a new solver. In order to keep full control of memory layout, it was written in C. Otherwise we ported all the important algorithms from CADICAL, and were also able to reconfirm their effectiveness in a fresh implementation. In this regard using "target phases" as introduced last year in CADICAL [1] should be emphasized, which after careful porting, gave a large improvement on satisfiable instances.

We want to highlight the following algorithmic differences. The first version of CADICAL had a sophisticated implementation of forward subsumption, building on the one in SPLATZ inspired by [8], which was efficient enough to be applied to learned clauses too. Only later we added vivification [9], which is now used in most state-of-the-art solvers, and is particularly effective on learned clauses [7]. Thus subsumption on learned clauses becomes less important and we only apply it on irredundant clauses before and during bounded variable elimination. We have both a fast forward subsumption pass for all clauses as well incremental backward but now also forward subsumption during variable elimination, carefully monitoring variables occurring in added or removed (irredundant) clauses, which allows us to focus the inprocessing effort.

The clause arena keeps irredundant clauses before redundant clauses, which allows during reduction of learned clauses in "reduce" to traverse only the redundant part of the arena. Since watches contain offsets to large clauses in the arena we can completely avoid visiting irredundant (original) clauses during this procedure. This substantially reduces the hot-spot of flushing and reconnecting watchers in watch lists during clause reduction. Note, that "reduce" beside "restart" is the most frequently called procedure in a CDCL solver (after the core procedures "propagate", "decide", and "analyze").

In comparison to CADICAL inprocessing procedures are scheduled slightly differently. First there is no forward subsumption of clauses outside of the "eliminate" procedure. In KISSAT compacting the variable range is part of "reduce" and actually always performed if new variables became inactive (eliminated, substituted or unit). Otherwise "probe" and "eliminate" call the same algorithms as in CADICAL, except for vivification which became part of "probe" and duplicated binary clause removal (aka hyper unary resolution), which has moved from "subsume" (thus in CADICAL triggered during search and during variable elimination) to "eliminate".

More importantly we have a more sophisticated scaling procedure for the number of conflicts between calls to "probe" and "eliminate", which as in CADICAL takes the size of the formula into account, but now applies an additional scaling function instead of just linearly increasing the base interval in terms of n denoting how often the procedure was executed.

For variable elimination ("elim") the scaling function of the base conflict interval is $n \cdot \log^2 n$. For "probe" it is $n \cdot \log n$. Similarly we scale the base conflict interval for "reduce" by $n/\log n$, while for "rephase" it remains linear. More precisely as logarithm we use $\log_{10}(n+10)$. Thus "reduce" occurs most often, followed by "rephase", then "probe" and least often "elim", all in the long run, independently of the base conflict interval, and the initial conflict interval.

Since boolean constraint propagation is considered the hot-spot for SAT solvers, CADICAL uses separate specialized propagation procedures during search, failed literal probing and vivification. In KISSAT we have factored out propagation code in a header file which can be instantianted slightly differently by these procedures, so taking advantage of dedicated propagation code while keeping the code in one place.

The concept of quiet "stable phases" without many restarts and "non-stable phases" with aggressive restarting was renamed. We call it now "stable mode" and "focused mode" to avoid the name clash with "phases" in "phase saving" (and "target phases"). We further realized that mode switching should not entirely be based on conflicts, since the conflict rate per second varies substantially with and without frequent restarts (as well as using target phases during stable mode).

Since the solver starts in focused mode, these focused mode intervals can still be based on a (quadratically) increasing conflict interval. For the next stable interval we then attempt to use the same time. Of course, in order to keep the solver deterministic, this requires to use another metric than run time. In CADICAL we simply doubled the conflict interval after each mode switch which did not perform as well in our experiments as this new scheme.

Our first attempt to limit the time spend in stable mode was to use the number of propagations as metric. But this was not precise enough, since propagations per second still vary substantially with and without many restarts. Instead we now count "ticks", which approximate the number of cache lines accessed during propagations. This refines what Donald Knuth calls "mems" but lifted to cache lines and restricted to only count watcher stack access and large clause dereferences, ignoring for instance accessing the value of a literal.

Cache line counting is necessary because in certain large instances with almost exclusively binary clauses most time is spend in accessing the watches with inlined binary clauses in watcher stacks and not in dereferencing large clauses, while in

general, and for other instances with a more balanced fraction of large and binary clauses, a single clause dereference is still considerably more costly than accessing an inlined binary clause. Computing these "ticks" was useful limit the time spent in other procedures, e.g., vivification, in terms of time spent during search (more precisely the time spend in propagation).

While porting the idea of target phases [1], we realized that erasing the current saved phases by for instance setting them to random phases, might destroy the benefit of saved phases to remember satisfying assignments of disconnected components of the CNF [10]. Instead of decomposing the CNF explicitly into disconnected components, as suggested in [10], we simply compute the largest autarky of the full assignment represented by saved phases, following an algorithm originally proposed by Oliver Kullman (also described in [2]).

This unique autarky contains all the satisfying assignments for disconnected components (as well as for instance pure literals). If the autarky is non-empty, its variables are considered to be eliminated and all clauses touched by it are pushed on the reconstruction stack. We determine this autarky each time before we erase saved phases in "rephase" and once again if new saved phases have been determined through local search.

Finally, combining chronological backtracking [11] with CDCL turns out to break almost the same invariants [12] as on-the-fly self-subsuming resolution [13], [14] and thus we added both, while CaDiCaL is missing the latter. Both techniques produce additional conflicts without learning a clause and thus initially we based all scheduling on the number of learned clauses instead on the number of conflicts, but our experiments revealed that using the number of conflicts provides similar performance and we now rely on that for scheduling.

As last year for CaDiCaL we submit three configurations of Kissat, one targeting satisfiable instances ("`sat`") always using target phases (also in focused mode), one for unsatisfiable instances ("`unsat`"), which stays in focused mode, and the default configuration ("`default`"), which alternates between stable and focused mode as described above, but only uses target phases in stable mode.

III. Paracooba

Our new solver Paracooba [15] has been submitted to the cloud track. It is a distributed cube-and-conquer solver. The input DIMACS is split on the master node into various subproblems (cubes) that can be solved independently. The work is distributed over the network first from the master node to other nodes and then across nodes depending on the workload of nodes.

The "quality" of the cubes is important for the efficiency of the solver. We have submitted two versions to the competition: one relies on the state-of-the-art lookahead solver March [16] for splitting; another uses our own implementation of tree-based lookahead [6]. Our implementation is part of CaDiCaL and is much less tuned than March. It is run with a timeout and, whenever splitting takes too long (more than 30 s), we fall back on the number of occurrences.

During solving, whenever a subproblem takes too long, i.e., based on a moving average of solving times, then we split the problem again into two or more subproblems. If many nodes are unused, we generate more (and hopefully simpler) subproblems in order to increase the amount of work that can be distributed onto further nodes.

Generated subproblems are solved using the incremental version of CaDiCaL described below in Sect. V and we aim at solving similar cubes on the same CaDiCaL instance to reuse the results of previous inprocessing.

IV. Plingeling and Treengeling

We submitted Plingeling and Treengeling to the parallel track. Compared to the version submitted to the 2018 SAT Competition [17] we have made essentially no changes to Plingeling and Treengeling nor to the SAT solver Lingeling that is used internally.

V. Incremental Track

CaDiCaL also enters the incremental track of the competition. It relies on our method [18] to identify and restore the necessary clauses when new clauses are added and can thereby make use of most of all the implemented inprocessing techniques. A sequence of incremental problems is considered as a stand-alone run from the perspective of inprocessing scheduling, i.e. none of the relevant inprocessing counters are reset in between iterations. The assumptions of each iteration are internally frozen (i.e. excluded from inprocessing), but beyond that there is no special treatment regarding them.

VI. License

Our solvers are all available under MIT license at http://fmv.jku.at/cadical for CaDiCaL, http://fmv.jku.at/kissat for Kissat, https://github.com/maximaximal/Paracooba for Paracooba, and https://github.com/arminbiere/lingeling for Plingeling and Treengeling.

References

[1] A. Biere, "CaDiCaL at the SAT Race 2019," in *SAT Race 2019*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, 2019, pp. 8–9.

[2] B. Kiesl, M. J. H. Heule, and A. Biere, "Truth assignments as conditional autarkies," in *ATVA 2019*, ser. LNCS, Y. Chen, C. Cheng, and J. Esparza, Eds., vol. 11781. Springer, 2019, pp. 48–64. [Online]. Available: https://doi.org/10.1007/978-3-030-31784-3_3

[3] C. Oh, "Between SAT and UNSAT: the fundamental difference in CDCL SAT," in *SAT 2015*, ser. LNCS, M. Heule and S. A. Weaver, Eds., vol. 9340. Springer, 2015, pp. 307–323. [Online]. Available: https://doi.org/10.1007/978-3-319-24318-4_23

[4] A. Biere, "Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016," in *SAT Competition 2016*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2016-1. University of Helsinki, 2016, pp. 44–45.

[5] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *IJCAR 2012*, ser. LNCS, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370. [Online]. Available: https://doi.org/10.1007/978-3-642-31365-3_28

[6] M. Heule, M. Järvisalo, and A. Biere, "Revisiting hyper binary resolution," in *CPAIOR 2013*, ser. LNCS, C. P. Gomes and M. Sellmann, Eds., vol. 7874. Springer, 2013, pp. 77–93. [Online]. Available: https://doi.org/10.1007/978-3-642-38171-3_6

[7] C. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li, "Clause vivification by unit propagation in CDCL SAT solvers," *Artif. Intell.*, vol. 279, 2020. [Online]. Available: https://doi.org/10.1016/j.artint.2019.103197

[8] R. J. Bayardo and B. Panda, "Fast algorithms for finding extremal sets," in *SDM 2011*. SIAM / Omnipress, 2011, pp. 25–34. [Online]. Available: https://doi.org/10.1137/1.9781611972818.3

[9] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for CDCL SAT solvers," in *IJCAI 2017*, C. Sierra, Ed. ijcai.org, 2017, pp. 703–711. [Online]. Available: https://doi.org/10.24963/ijcai.2017/98

[10] A. Biere and C. Sinz, "Decomposing SAT problems into connected components," *J. Satisf. Boolean Model. Comput.*, vol. 2, no. 1-4, pp. 201–208, 2006. [Online]. Available: https://satassociation.org/jsat/index.php/jsat/article/view/26

[11] A. Nadel and V. Ryvchin, "Chronological backtracking," in *SAT 2018e*, O. Beyersdorff and C. M. Wintersteiger, Eds., vol. 10929. Springer, 2018, pp. 111–121. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_7

[12] S. Möhle and A. Biere, "Backing backtracking," in *SAT 2019*, ser. LNCS, M. Janota and I. Lynce, Eds., vol. 11628. Springer, 2019, pp. 250–266. [Online]. Available: https://doi.org/10.1007/978-3-030-24258-9_18

[13] H. Han and F. Somenzi, "On-the-fly clause improvement," in *SAT 2009*, ser. LNCS, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 209–222. [Online]. Available: https://doi.org/10.1007/978-3-642-02777-2_21

[14] Y. Hamadi, S. Jabbour, and L. Sais, "Learning for dynamic subsumption," in *ICTAI 2009*. IEEE Computer Society, 2009, pp. 328–335. [Online]. Available: https://doi.org/10.1109/ICTAI.2009.22

[15] M. Heisinger, M. Fleury, and A. Biere, "Distributed cube and conquer with Paracooba," in *SAT 2020*, L. Pulina and M. Seidl, Eds. Springer, 2020, (Accepted).

[16] M. J. H. Heule, M. Dufour, J. van Zwieten, and H. van Maaren, "March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver," in *SAT 2004*, ser. LNCS, vol. 3542. Springer, 2004, pp. 345–359. [Online]. Available: https://doi.org/10.1007/11527695_26

[17] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018," in *SAT Competition 2018*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2018-1. University of Helsinki, 2018, pp. 13–14.

[18] K. Fazekas, A. Biere, and C. Scholl, "Incremental inprocessing in SAT solving," in *SAT 2019*, ser. LNCS, M. Janota and I. Lynce, Eds., vol. 11628. Springer, 2019, pp. 136–154. [Online]. Available: https://doi.org/10.1007/978-3-030-24258-9_9