Concurrent Clause Strengthening

Siert Wieringa and Keijo Heljanko*

Aalto University, School of Science
Department of Information and Computer Science
P.O. Box 15400, FI-00076 Aalto, Finland
{siert.wieringa,keijo.heljanko}@aalto.fi

Abstract. This work presents a novel strategy for improving SAT solver performance by using concurrency. Rather than aiming to parallelize search, we use concurrency to aid a conventional CDCL search procedure. More concretely, our work extends a conventional CDCL SAT solver with a second computation thread, which is solely used to strengthen the clauses learned by the solver. This provides a simple and natural way to exploit the availability of multi-core hardware.

We have employed our technique to extend two well established solvers, MiniSAT and Glucose. Despite its conceptual simplicity the technique yields a significant improvement of those solvers' performances, in particular for unsatisfiable benchmarks. For such benchmarks an extensive empirical evaluation revealed a remarkably consistent reduction of the wall clock time required to determine unsatisfiability, as well as an ability to solve more benchmarks in the same CPU time.

The proposed technique can be applied in combination with existing parallel SAT solving techniques, including both portfolio and search space splitting approaches. The approach presented here can thus be seen as orthogonal to those existing techniques.

1 Introduction

Propositional satisfiability (typically abbreviated SAT) is the problem of finding a satisfying truth assignment for a given propositional logic formula, or determining that no such assignment exists. This classifies the formula as respectively *satisfiable* or *unsatisfiable*. SAT is an important theoretical problem as it was the first problem ever to be proven NP-complete [8].

Despite the theoretical hardness of SAT, current state-of-the-art decision procedures for SAT, so called *SAT solvers*, have become surprisingly efficient. The introduction of *Conflict Driven Clause Learning* (CDCL) [23] was a crucial step in the process of making these algorithms into industrial strength problem solvers. However, modern SAT solvers are not just efficient implementations of the CDCL search procedure. Instead, they implement several forms of extra reasoning. For example, formula simplification before CDCL search, so called

* This work was partially funded by the Helsinki Institute for Information Technology (HIIT) and the Academy of Finland, project 139402.

M. Järvisalo and A. Van Gelder (Eds.): SAT 2013, LNCS 7962, pp. 116–132, 2013.

preprocessing, is commonly used. An important work for the widespread adaptation of this technique was the introduction of an efficient preprocessor called SatELite [10]. Some modern solvers, such as Lingeling [6], use *inprocessing*, which is the sequential interleaving of search and simplification procedures. For a recent and extensive overview of pre- and inprocessing techniques, as well as a concise set of rules formalizing such techniques please refer to [21].

Another technique that is crucial for performance, but not part of the core CDCL search procedure is *conflict clause strengthening* (e.g. [11,14,29]). This is usually performed during conflict analysis (see Sec. 2). The length of a conflict clause can be efficiently reduced to a *minimal* clause implied by the set of clauses used in its derivation [14,29]. Hence, the name *conflict clause minimization* is also commonly used [11]. However, reducing a conflict clause to a *minimal* conflict clause implied by all clauses in the formula is NP-hard, as this clause is of length zero iff the formula is unsatisfiable. In [14] the related *Generalized Conflict-Clause Strengthening* problem was defined and proven NP-hard.

Sequentially interleaving a more expensive conflict clause strengthening procedure with the core CDCL search procedure may provide performance improvements, but it is hard to develop good heuristics for deciding when to switch between searching and conflict clause strengthening. As the majority of computer hardware is nowadays equipped with multi-core CPUs conflict clause strengthening can instead be performed *in parallel* with the core search procedure. We have developed a novel solving architecture that uses two computation threads, one for CDCL search and one solely for strengthening conflict clauses. The conflict clause strengthening algorithm used is similar to existing algorithms for problem clause strengthening [25,17]. In extensive experimental results we show the performance of two implementations of our architecture, based on two different well established SAT solvers, MiniSAT [12] and Glucose [2].

For all experiments we will present results regarding *wall clock time* and *CPU time*. Wall clock time is defined as the amount of time that passes from the start to the finish of the solving process, and this measure is independent of the amount of resources that are used during that time. CPU time on the other hand is the sum of the time spend by each of the cores used, i.e. if a single program uses all the computation power of two CPU cores concurrently then the CPU time grows twice as fast as the wall clock time.

The presented two threaded solver maintains all the features of a normal SAT solver, including for example its interface for incremental SAT solving [13]. Hence, our new solver could in principle replace the conventional solver inside the parallel incremental solver that we presented in recent work [30]. Although the performance of the technique presented here in combination with incremental SAT is interesting this is left for further work.

One may consider a parallelization of an algorithm as a strategy for assigning any number of simultaneously available computation resources to performing a single task. By that definition this work does not present a parallelization of a SAT solver, as only the use of exactly two concurrent computation threads is considered. However, existing techniques for parallelizing SAT algorithms can be

used in combination with our two threaded solver, in order to obtain a generic parallelization. Even running multiple copies of our two threaded solver is a practical proposition for such environments, given its good performance regarding CPU time. Hence, we will provide a short overview of relevant work on parallelizing SAT solvers.

Two major approaches for parallelizing SAT algorithms can be distinguished [19]. The first is the classic divide-and-conquer approach, which aims to partition the formula to divide the total workload evenly over multiple SAT solver instances [7,28,32]. The second approach is the so called *portfolio* approach [16,22]. Rather than partitioning the formula, portfolio systems run multiple solvers in parallel each of which attempt to solve the same formula. The system finishes whenever the fastest solver is done. Both approaches can be extended with some form of conflict clause sharing between the solver threads.

Although other techniques have recently been developed (e.g. [18,19]) portfolio solvers have received the majority of the research attention in recent years. Some insight into the good performance of these approaches is provided in [20]. ManySAT [16] is a well known adaptation of the portfolio strategy. It employs conflict clause sharing and is thus a so called *cooperative portfolio*. It is build around running multiple copies of the sequential solver MiniSAT [12] in parallel. Although each of the solver threads may be given different settings the threads are largely *homogeneous*.

Other portfolio solvers are *non-cooperative*, but use truly *heterogeneous* solver threads. Examples are ppfolio[1], SATzilla [31], and 3S [22]. These all use a collection of different SAT solvers from several developers. Whereas SATzilla and 3S try to be clever about which solvers to use for solving a particular formula ppfolio is completely naïve. In fact, ppfolio is a very simple program that just executes multiple solvers in parallel. It was meant to illustrate a lower bound on what is achievable using portfolios[1], but it turned out to be one of the strongest solvers at the SAT Competition[2] in 2011.

In several recent cooperating portfolios using homogeneous solver threads, such as Plingeling [6], cooperation is limited to sharing unit clauses only. The relatively weak performance of portfolios sharing more than just unit clauses was the motivation for work presenting a new set of clause sharing heuristics [1]. The solver implementing those heuristics, called PeneLoPe, won a silver medal at the SAT Challenge in 2012 [4]. The winner, solving one instance more, was a non-cooperative portfolio called pfolioUZK [4]. PeneLoPe is based on ManySAT and its performance is remarkable, considering its use of homogeneous solver threads.

2 Definitions

A *literal* l is either a Boolean variable x or its negation $\neg x$. Double negations cancel out, hence $\neg\neg l = l$. An *assignment* ρ is a set of literals such that if $l \in \rho$ then $\neg l \notin \rho$. If $l \in \rho$ we say that literal l is assigned the value **true**. If $\neg l \in \rho$

[1] `www.cril.univ-artois.fr/~roussel/ppfolio`
[2] `www.satcompetition.org`

it is said that l is assigned the value **false**, or equivalently that l is *falsified*. If for some literal l neither l nor $\neg l$ is in the assignment ρ then l is *unassigned*. For an assignment ρ we denote by $\neg\rho$ the set $\{\neg l \mid l \in \rho\}$. A *clause* c is a set of literals $c = \{l_0, l_1, \cdots, l_n\}$ representing the disjunction $\bigvee c = l_0 \vee l_1 \cdots \vee l_n$. Hence, clause c is *satisfied* by assignment ρ iff $c \cap \rho \neq \emptyset$. A clause consisting of exactly one literal is called a *unit clause*.

As typical in work on SAT solvers, we only consider formulas in *conjunctive normal form* (CNF). Such formulas are formed as conjunctions of disjunctions, and hence can be represented as sets of clauses. The formula $\mathcal{F}$ *under the assignment* ρ is denoted $\mathcal{F}^\rho$ as in [5]. It is defined as the formula $\mathcal{F}$ after removing all clauses satisfied by ρ, followed by shrinking the remaining clauses by removing literals that are falsified by ρ. Formally:

$$\mathcal{F}^\rho = \{\ c \setminus \neg\rho \ \mid \ c \in \mathcal{F} \text{ and } c \cap \rho = \emptyset\ \}$$

Let $iup(\mathcal{F}, \rho)$ be the assignment ρ that is the result of executing the following *iterative unit propagation* loop:

$$\textbf{while } \{l\} \in \mathcal{F}^\rho \textbf{ do } \rho = \rho \cup \{l\}$$

Moreover we define $\mathcal{F}|_\rho = \mathcal{F}^{iup(\mathcal{F}, \rho)}$, which is the result of simplifying formula $\mathcal{F}$ under assignment ρ by iterative unit propagation. If $\emptyset \in \mathcal{F}|_\rho$ we say that a *conflict* has been reached. If on the other hand $\mathcal{F}|_\rho = \emptyset$ then assignment ρ satisfies $\mathcal{F}$. The DPLL algorithm [9] is the classical algorithm for determining the satisfiability of CNF formulas. It starts from the formula $\mathcal{F}$ and an empty assignment ρ, and alternates between iterative unit propagation and *branching decisions*. During a branching decision, or simply *decision*, the algorithm picks a *decision variable* x_d that is unassigned by ρ and assigns it to either **true** or **false**. Whenever iterative unit propagation leads to a conflict the algorithm backtracks to the last decision to which it had not backtracked before, and negates the assignment made at that decision. This backtracking search continuous until either all variables of $\mathcal{F}$ are assigned, or all branches of the search tree have been unsuccessfully explored. In the former case ρ satisfies $\mathcal{F}$, in the latter case $\mathcal{F}$ is unsatisfiable.

Most modern SAT solvers are so called *Conflict Driven Clause Learning* (CDCL) solvers [23,24]. Just like the basic DPLL procedure the search for a satisfying assignment proceeds by alternating between iterative unit propagation and branching decisions. The crucial difference is what happens when a conflict is reached. In this case, a CDCL solver will analyze the sequence of decisions and implications that lead to the conflict. During this *conflict analysis* the solver derives a *conflict clause*, which is a clause implied by the input formula that gives a representation of the "cause" of the conflict. By including the conflict clause in the set of clauses on which iterative unit propagation is performed hitting another conflict with the same cause can be avoided.

An important property of the most popular clause learning scheme for CDCL solvers, called *first unique implication point* (1-UIP) [24], is that each conflict clause contains exactly one literal that was falsified by the last decision or the subsequent unit propagation. This literal is called the *asserting literal*.

After conflict analysis the CDCL solver must backtrack. Unlike the DPLL procedure CDCL solvers use non-chronological backtracking, which is driven by the conflict clauses. By definition all literals in a conflict clause are assigned the value **false** by assignment ρ when it is derived. After learning conflict clause c, the solver backtracks until the earliest decision at which all literals of c except the asserting literal l_a are assigned **false**. The literal l_a is then assigned the value **true**, as this is required to satisfy c. Subsequent unit propagation may yield a new conflict which is handled in the same way.

Any conflict clause c, derived by a CDCL solver from the formula $\mathcal{F}$ with the aid of previously derived conflict clauses P, can be derived using a so called *trivial resolution derivation* [5]. This implies that if the value **false** is assigned to all literals of a conflict clause, then the result of simplifying formula $\mathcal{F} \cup P$ under that assignment by iterative unit propagation is guaranteed to reach a conflict, i.e. the following property holds:

$$\emptyset \in \mathcal{F}'|_\rho \text{ for } \mathcal{F}' = \mathcal{F} \cup P \text{ and } \rho = \{\neg l \mid l \in c\} \tag{1}$$

Another important property of conflict clauses derived using the 1-UIP scheme is their *1-empowerment* property [27]. Informally this means that if all literals of a conflict clause c except its asserting literal l_a are assigned to **false**, then iterative unit propagation on the set $\mathcal{F} \cup P$ does not yield the necessary truth assignment **true** to l_a, i.e. the following property holds:

$$l_a \notin iup(\mathcal{F}', \rho) \text{ for } \mathcal{F}' = \mathcal{F} \cup P \text{ and } \rho = \{\neg l \mid l \in c \text{ and } l \neq l_a\} \tag{2}$$

It is said that c is *1-empowering* with respect to $\mathcal{F} \cup P$ via its asserting literal l_a. This property implies that adding the conflict clause c to the learnt clause set P strictly extends the propagation abilities of the solver. In other words, the *deductive power* of the CNF formula $\mathcal{F} \cup P$ is strictly increased [17].

3 The Solver-Reducer Architecture

We propose an architecture using two concurrently executing threads, which are called the SOLVER and the REDUCER. The SOLVER acts like any conventional SAT solver, except for its interaction with the REDUCER. The interaction between the SOLVER and the REDUCER is limited to passing clauses through two shared-memory data structures called the *work set* and the *result queue*. The work set is used to pass clauses from the SOLVER to the REDUCER, the result queue is used for passing clauses in the opposite direction, as illustrated in Fig. 1.

Whenever the SOLVER learns a clause it writes a copy of that clause to the work set. The REDUCER reads clauses from the work set and tries to strengthen them. When the REDUCER successfully reduces the length of a clause, it places the new shorter clause in the result queue. The SOLVER checks frequently whether there are any clauses in the result queue. If this is the case the SOLVER enters the clauses from the result queue in its learnt clause set. It is possible that all of the literals in such a clause are assigned the value **false** in the current assignment

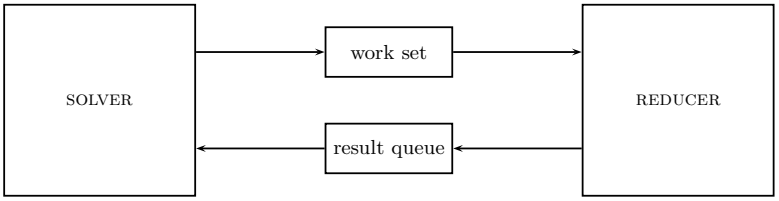


Fig. 1. The solver-reducer architecture

ρ of the SOLVER. In this case the SOLVER must backtrack before entering the clause in the set. Our implementation of *on-the-fly* addition of "foreign" clauses in the SOLVER tries to keep backtracking to a minimum. If the clause is asserting then this is handled in the same way as for normal conflict clauses. We could have chosen to introduce the clauses only when the SOLVER's assignment ρ contains no decision variables (as in e.g. [30]), but a more dynamic approach was considered more appropriate here. There is no mechanism to ensure that the clause c is removed from the SOLVER's learnt clause set when a clause $c' \subset c$ is obtained from the result queue.

Our proposed architecture is conceptually simple, and we will show that it can provide substantial performance improvements. An unfortunate side-effect of our approach is that the behavior of the solver becomes non-deterministic, as the execution order is determined by the operating system's thread scheduling policy. This means that runs of our two threaded solver are not reproducible, and performance may vary drastically in between two different runs on the same formula. The same problem occurs also when using more conventional parallelizations of SAT solvers. In [15] it was shown that a deterministic version of ManySAT, using *synchronization barriers* and a dynamic heuristic for deciding when to perform synchronization, on average performed almost as well as the original version. We believe that a similar technique could be applied successfully in an implementation of the solver-reducer architecture.

3.1 The Reducer

The REDUCER continuously checks the work set for new input clauses, and then runs its reduction algorithm. The algorithm is based on unit propagation and conflict clause learning. Basically, the algorithm assigns the literals of the input clause c_{in} to **false** one by one until iterative unit propagation leads to a conflict, or all variables are assigned. The pseudocode for this algorithm is given in Fig. 2, where P_R represents the set of learnt clauses maintained by the REDUCER.

Consider the case where the algorithm returns c_{out} from Line 7. This case occurs iff there is no trivial resolution derivation of c_{in} from $\mathcal{F} \cup P_R$. This is possible, because the set P_R does not necessarily contain all the clauses P_S that were contained in the SOLVER when c_{in} was derived. The returned clause $c_{out} \subseteq c_{in}$ is obtained by removing with respect to c_{in} any literals $l \in c_{in}$ for

1. $c_{out} = \emptyset$; $\rho = \emptyset$
2. $\mathcal{F}' = \mathcal{F} \cup P_R$
3. for $l \in (c_{in} \setminus c_{out})$ s.t. $\neg l \notin iup(\mathcal{F}', \rho)$
4. if $\emptyset \in \mathcal{F}'|_\rho$ then $(P_R, c_{new}) = \text{ANALYZE}(\mathcal{F}, P_R, \rho)$; return c_{new}
5. $c_{out} = c_{out} \cup \{l\}$; $\rho = \rho \cup \{\neg l\}$
6. $P_R = P_R \cup \{c_{out}\}$
7. return c_{out}

Fig. 2. Pseudocode for the REDUCER's algorithm

which the value **false** of the corresponding literal l was implied rather than assigned. This implements *self-subsumption resolution* [10]. It is sound because the forced falsification of l means that for some $c \subset c_{in}$ it holds that $\mathcal{F} \models c \cup \{\neg l\}$, and by resolution on the clauses c_{in} and $c \cup \{\neg l\}$ it follows that $\mathcal{F} \models c_{in} \setminus \{l\}$. Because c_{out} is 1-empowering with respect to $\mathcal{F} \cup P_R$ it is added to P_R.

Now consider the case were the algorithm returns after calling the function ANALYZE on Line 4 of the pseudocode. The function ANALYZE analysis the conflicting assignment ρ. Until ρ is non-conflicting or $\rho = \emptyset$ it performs backtracking by removing literals from ρ, conflict clause learning by adding clauses to P_R, and iterative unit propagation. Because for each clause added to P_R at least one literal is removed from ρ the number of new conflict clauses is bounded by $|\rho| \leq |c_{in}|$. These conflict clauses are crucial for the performance of the REDUCER, but they are never shared with the SOLVER. The function ANALYZE returns a clause[3] c_{new} such that $c_{new} \subseteq c_{out}$. Consider the assignment ρ after the backtracking performed by ANALYZE. If $\rho = \emptyset$ then $c_{new} = \emptyset$. Else, for some $l \in c_{out}$ and $\rho' \subseteq \rho$ it holds that $l \in iup(\mathcal{F}', \rho')$. In this case $c_{new} = \{l' \mid \neg l' \in \rho'\} \cup \{l\}$.

Our REDUCER's algorithm is very similar to the *vivification algorithm* of [25]. The vivification algorithm aims to find redundant literals in the problem clauses $c \in \mathcal{F}$ by assigning the value **false** for the literals in c, and performing unit propagation on the formula $\mathcal{F} \setminus \{c\}$. In case a conflict arises the algorithm learns only exactly one new conflict clause. The order in which the literals are assigned is heuristically controlled in the vivification algorithm. In our REDUCER implementation the literals are assigned in the order in which they appear in the clause. Due to the organization of the conflict clause analysis procedure of the SOLVER this means that the asserting literal of the conflict clause is always assigned first. Note that the clause $c \cup \{l_a\}$, where l_a is the asserting literal, is equivalent to the implication $(\neg \bigvee c) \rightarrow l_a$. Although the clause can be rewritten as an implication with any one of its literals as the consequent, this implication of l_a is the one that guaranteed the 1-empowering property of the clause in the SOLVER. By starting from l_a, the REDUCER aims to reduce the size of the antecedent $\neg c$ of this deduction power increasing implication.

[3] In the implementation c_{new} is obtained using MiniSAT's ANALYZEFINAL function, where ρ is regarded as the set of *assumptions* [13].

The REDUCER can not do anything that a conventional solver could not also do in the same number of steps. This means that the solver-reducer architecture does not implement a stronger proof system (see, e.g. [5]) than a conventional CDCL solver. In fact, a modern SAT solver using the VSIDS heuristic [24], phase saving [26], and frequent restarts (e.g. [3]) has a tendency to "run towards conflicts" just like the REDUCER does. Consider a SAT solver using phase saving and extreme parameter settings: It restarts after every conflict, and uses a VSIDS activity decay so large that the set of variables involved in the most recent conflict always have larger activities than any other variables. In this case, after every conflict and the subsequent restart, the VSIDS heuristic will pick as decision variables those variables that occur in the most recent conflict clause. Combined with phase saving this will lead to the same sequence of assignments as the REDUCER would make to reduce that conflict clause.

3.2 The Work Set

A set of clauses stored in a shared-memory data structure called the work set is forming the inputs of the REDUCER. It is possible to implement the work set as a simple unbounded FIFO queue. This may be sufficient if the REDUCER has only very few clauses in its learnt clause set, as in this case it can often perform unit propagation fast enough to keep up with the conflict clause generation of the SOLVER. However, the clause learning in the REDUCER is crucial to the strength of the reduction procedure. As the size of REDUCER's learnt clause set increases it is able to provide stronger reduced clauses, but at a lower speed.

If the REDUCER can not keep up with the SOLVER then a work set implemented as a FIFO queue will just keep growing. As the REDUCER lags behind it will only strengthen "old" clauses, that are less likely to be of use to the SOLVER. An unbounded LIFO queue would make the REDUCER focus on reducing recent clauses first, but strong clauses may shift to the back of such a queue quickly if the REDUCER is momentarily busy. Giving preference to strengthening clauses that are likely to be "important" to the SOLVER is natural. The "quality" of a conflict clause can be crudely approximated by its length, or alternatively by its *Literal Blocks Distance* (LBD) [2]. Hence, an alternative work set implementation could keep an unbounded set of clauses sorted by their length. However, as the average conflict clause length changes during the search, a clause that was relatively long ("bad") at the time it was learned may seem relatively short ("good") after some time has passed. Thus, this unbounded sorted set also leads to reducing outdated clauses. The same argument holds when the LBD is used for sorting the set, as the LBD measure of a clause is bounded by its length.

We achieved the best performance using work set with a limited capacity. If the SOLVER enters a clause into a full work set then this clause will replace the oldest clause in the set. If the REDUCER requests a clause from the work set it is given the "best" clause from the set. In this way, the REDUCER's inputs are kept both "fresh" and "good".

3.3 Implementation

We have implemented our solver-reducer architecture on top of two well established existing SAT solvers, MiniSAT 2.2.0[4] [12] and Glucose 2.1[5] [2]. MiniSAT is often used as a basis for the development of new solving techniques, as witnessed by the existence of a "MiniSAT hack track" at the SAT competitions. Glucose won the SAT Challenge in 2012 [4]. Older Glucose versions won at the applications tracks of the SAT competitions in 2009 (for unsatisfiable benchmarks) and in 2011 (for mixed benchmarks). Because Glucose is based on MiniSAT the solver-reducer architecture was easy to port to Glucose once it had been developed inside MiniSAT. We will refer to the two open-source[6] solver-reducer implementations we created as respectively MiniRed and GlucoRed.

Both the SOLVER and the REDUCER of MiniRed are build as extensions to the MiniSAT solver. All the default settings and heuristics of MiniSAT were maintained in the SOLVER and the REDUCER. Similarly, the SOLVER and the REDUCER of GlucoRed maintain the default settings of the Glucose solver. An example of a heuristic that concerns both the SOLVER and the REDUCER is the heuristic for deciding when to reduce the size of their learnt clause sets. GlucoRed uses the LBD measure of a clause as a sorting metric for the work set, i.e. when the REDUCER requests a clause from the work set it is given the clause with the smallest LBD. Because MiniSAT does not compute LBD values MiniRed uses clause length as a sorting metric instead. The result queue is implemented as an unbounded FIFO queue.

The two threads interact solely by passing clauses, or more precisely pointers to clauses, through the work set and the result queue. Exclusive access to those datastructures is achieved by the use of a single *lock*. To reduce the number of times the lock must be acquired the SOLVER and REDUCER always combine read and write operations. In the REDUCER this is straightforward: If the length of a clause is reduced, then the new shorter clause is written to the result queue once the lock has been obtained to read a new input clause from the work set. The SOLVER combines reading and writing by checking the result queue for new reduced clauses whenever it has acquired the lock to write a new clause to the work set, i.e. whenever it derives a conflict clause. The SOLVER always postpones the addition of reduced clauses from the result queue to its learnt clause database until just before its next branching decision.

4 Experimental Evaluation

All experiments in this work were performed in a computing cluster, using machines that each have two six core Intel Xeon X5650 processors[7]. A memory limit of 7GB was enforced.

[4] `http://www.minisat.se`
[5] `http://www.lri.fr/~simon`
[6] `http://users.ics.aalto.fi/swiering/solver_reducer`
[7] These resources were provided by the Aalto Science-IT project.

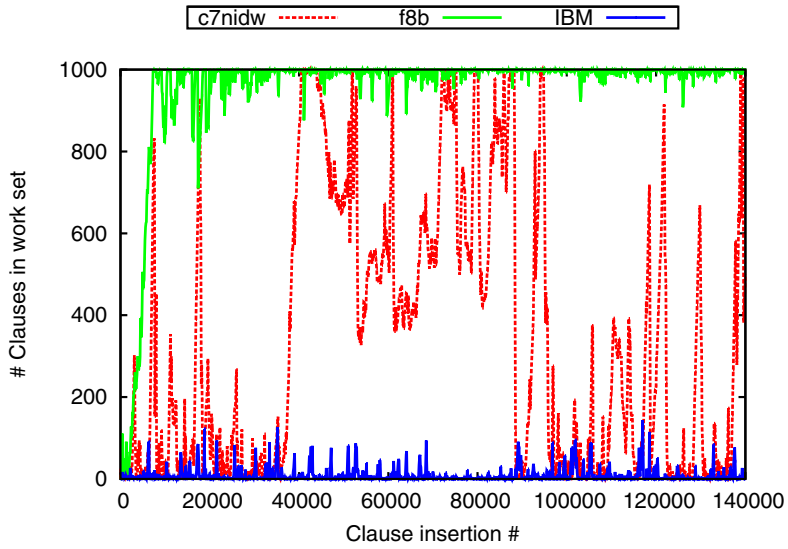


Fig. 3. Erratic use of the work set during three different runs

The MiniSAT [12] distribution provides a version of the solver with an integrated preprocessor, which is similar to SatELite [10]. During the SAT Challenge [4] the developers of Glucose [2] did not use such an internal preprocessor. Instead, they provided a script that first ran SatELite, and then ran Glucose on the resulting formula. For a fair comparison of the strength of the solvers we chose to run all solvers without enabling their integrated preprocessors, and provide them with both original and simplified benchmarks.

The first set of benchmarks we used is named *Competition*, and contains in total 547 benchmarks. The set combines 300 benchmarks from the application track of the SAT competition held in 2011, and the 247 application track benchmarks from the SAT Challenge 2012 that were marked as unused in previous competitions [4]. The set *Simplified* contains 501 benchmarks that are the result of running SatELite on the set *Competition*. The difference in size between the *Competition* set and the *Simplified* set is caused by leaving out benchmarks that were proven unsatisfiable by SatELite, and benchmarks that could not be simplified in 15 minutes.

4.1 Capacity of the Work Set

All experiments used a work set with a capacity of 1000 clauses. The average performance of our implementation is not particularly sensitive to this setting. It is however hard to make any general statements about the typical use of the work set. We illustrate this using a small experiment for which we solved three unsatisfiable benchmarks from the *Simplified* set using MiniRed. Each of these benchmarks takes just over thirty seconds to solve using conventional MiniSAT.

In Fig. 3 the number of clauses in the work set just before a new clause is inserted is plotted for the SOLVER's first 140 000 conflict clauses[8]. The graph shows that the use of the work set is very different for the three benchmarks. For `f8b` the set fills up almost immediately and remains full afterwards, whereas for `c7nidw` the size keeps varying dramatically. For benchmark `IBM`[9] the REDUCER easily keeps up with the supply of clauses, as the work set never fills. Interestingly, `IBM` was also the only one of the three benchmarks for which the added REDUCER did not seem to be beneficial for the solver's performance.

4.2 Clause Length

The numbers in Fig. 4 were obtained using MiniRed and the benchmarks from the *Simplified* set. MiniRed was run twice for every benchmark, once with the default settings, and once with the standard MiniSAT conflict minimization procedure [11] disabled in the SOLVER. In total 367 benchmarks were solved within 1800 seconds of CPU time during both runs.

Let us first focus on the numbers printed in a bold font, which represent the results for MiniRed's default settings. The numbers on the arrows indicate the average length of all the clauses that passed it during the 367 runs. Note that the absolute values of these numbers are meaningless, as they are averaged over a large set of independent and very different runs. The relation between these numbers nevertheless provides some insight in the operation of our architecture.

The arrow that points up out of the work set represents the clauses that are deleted from the work set because of its limited capacity, as described in Sec. 3.2. During this experiment on average 34.6% of the clauses placed in the work set were deleted. The average length of those clauses is large (91.3) compared to the average length of the clauses that are entering the work set (56.8). This was expected, as the work set delivers the shortest clauses first to the REDUCER. The average length of the clauses passing through the REDUCER dropped from 38.1 to 27.6 literals. On average 30.2% of the clauses remain the exact same length after passing through the REDUCER. These clauses are not placed in the result queue, as represented by the arrow that points down at the bottom of Fig. 4. Unsurprisingly the average length of those clauses is rather short (15.3).

The results for the experiment in which MiniRed was run with the SOLVER's conflict clause minimization disabled are printed in an italic font in the figure. The total number of conflict clauses generated by the SOLVER over the 367 runs grew by 17%, and those clauses were on average 2.5 times longer. However, the clauses that are actually delivered from the work set to the REDUCER are not much longer than those in the first experiment, and after reduction they are even slightly shorter. Surprisingly, the average overall performance of MiniRed was almost identical in both experiments. Note that disabling the conflict clause minimization in conventional MiniSAT results in a substantial degradation of

[8] For the benchmarks `c7nidw` and `IBM` the total number of conflict clauses generated by the SOLVER was slightly over 300 000 clauses, for `f8b` the total was around 150 000.

[9] `IBM` abbreviates `IBM_FV_2004_rule_batch_26_SAT_dat.k95`.

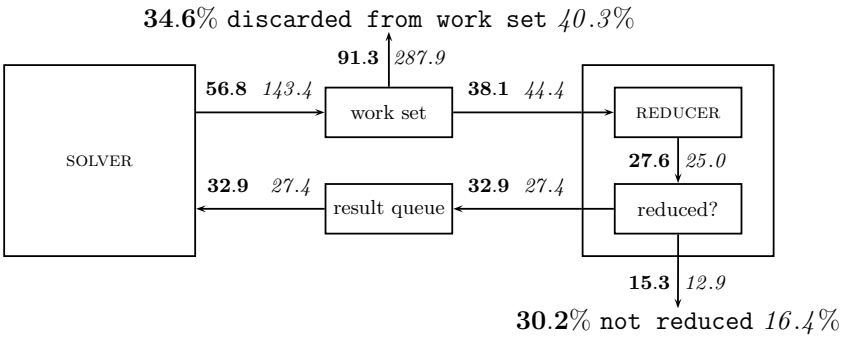


Fig. 4. Average clause lengths over 367 benchmarks

the performance [11], hence MiniRed's consistent good performance is made possible by the REDUCER. It apparently did not harm the SOLVER that the length of the longest clauses was never reduced at all, not even using the conflict clause minimization routine. In the remainder of this work we will only use the default settings, in which the SOLVER's conflict clause minimization is enabled.

4.3 Performance

Table 1 contains the number of benchmarks solved by the four different solvers within 900 seconds of wall clock time. The numbers in the table that are printed in a smaller font inside brackets represent the number of benchmarks solved within 1800 seconds of CPU time. The column VBS (*Virtual Best Solver*) provides the total number of benchmarks solved by at least one of the four solvers. The columns labelled Δ underline the difference between the number of benchmarks solved with- and without REDUCER.

Note the impressive performance improvement the solver-reducer architecture provides for unsatisfiable benchmarks. MiniRed improves the number of unsatisfiable benchmarks solved for the *Simplified* set by 58 benchmarks, and even regarding CPU time still provides a 31 benchmark improvement over MiniSAT. The gaps are smaller but still significant for the Glucose based implementation. The results for the unsatisfiable benchmarks from the *Competition* set are presented as cactus plots in Fig. 5. Comparison is made based on wall clock time in Fig. 5a and based on CPU time in Fig. 5b. The same is done for the unsatisfiable benchmarks from the *Simplified* set in Fig. 6a and Fig. 6b. The logarithmic-scale scatter plots in Fig. 7 and Fig. 8 provide another presentation of the wall clock time performance of MiniSAT versus MiniRed, and Glucose versus GlucoRed. The remarkable consistency of the improvement for unsatisfiable benchmarks can be clearly seen.

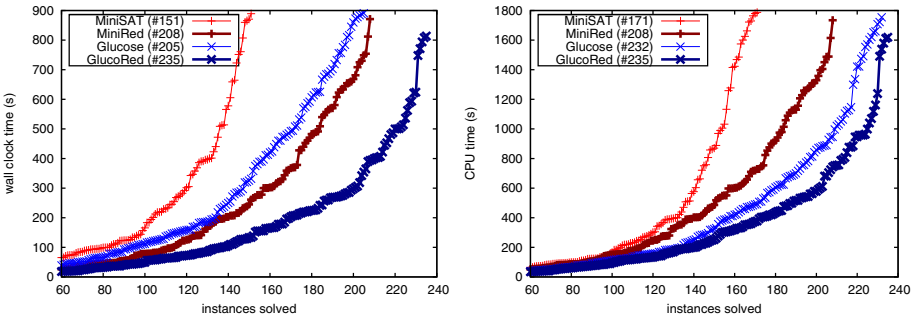
It is not surprising that the REDUCER does not contribute much to the average performance for *satisfiable* benchmarks, and that thus for such benchmarks the addition of the REDUCER results in worse performance regarding the CPU

Table 1. Number of benchmarks solved

Set		VBS	MiniSAT	MiniRed	Δ	Glucose	GlucoRed	Δ
Competition	UNSAT	239 (251)	151 (171)	208 (208)	57 (37)	207 (234)	235 (235)	28 (1)
	SAT	177 (179)	166 (174)	168 (168)	2 (-6)	158 (167)	155 (157)	−3 (-10)
Simplified	UNSAT	246 (249)	164 (191)	222 (222)	58 (31)	220 (232)	237 (237)	17 (5)
	SAT	166 (168)	150 (157)	159 (159)	9 (2)	155 (157)	147 (149)	−8 (-8)

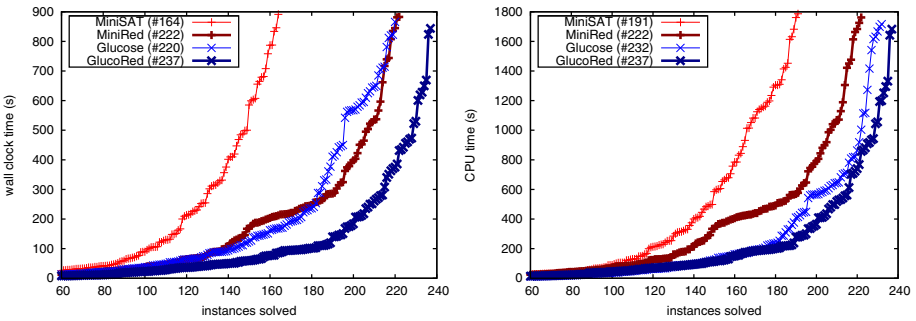
Table 2. Number of benchmarks in the *Simplified* set solved by PeneLoPe

	GlucoRed	PeneLoPe-2	PeneLoPe-4	PeneLoPe-8
UNSAT	237 (237)	227 (227)	231 (221)	247 (217)
SAT	147 (149)	142 (142)	160 (154)	164 (149)



(a) (b)

Fig. 5. Results for unsatisfiable benchmarks from the *Competition* set



(a) (b)

Fig. 6. Results for unsatisfiable benchmarks from the *Simplified* set

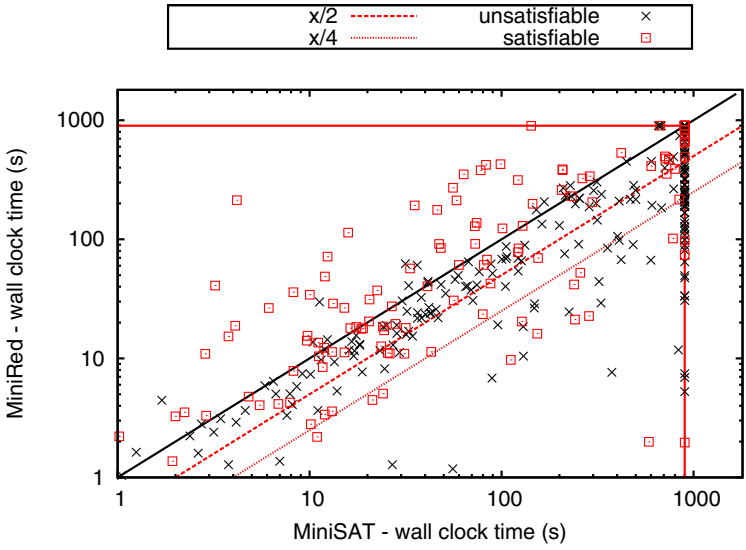


Fig. 7. MiniSat versus MiniRed on benchmarks from the *Simplified* set

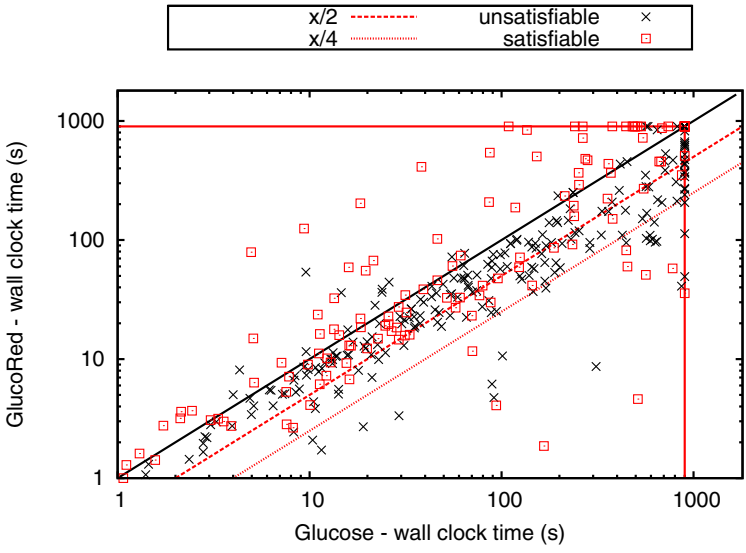


Fig. 8. Glucose versus GlucoRed on benchmarks from the *Simplified* set

time. Between Glucose and GlucoRed, the number of satisfiable benchmarks from the *Simplified* set degrades even regarding wall clock time. Glucose uses many tunable heuristics that we left untouched when creating GlucoRed. Some of these, such as the *restart blocking heuristic* [3], may be negatively affected by the on-the-fly introduction of REDUCER result clauses. An important measure for the heuristics inside Glucose is the average LBD of its conflict clauses. GlucoRed does not incorporate the clauses provided by the REDUCER in this average. Moreover, Glucose and GlucoRed use the LBD measure for deciding which clauses to remove when reducing their learnt clause sets. MiniSAT and MiniRed use an activity based heuristic for this purpose. We expect that for implementations of the solver-reducer architecture the latter is better because it has a natural tendency to delete subsumed clauses. This is important as the REDUCER provides clauses to the SOLVER that are subsumed by clauses that are (or were) already in its learnt clause set. Adaptation of the heuristics from PeneLoPe [1] may also improve GlucoRed's performance.

It would be interesting to study the performance of a PeneLoPe style portfolio of solver-reducer implementations such as GlucoRed. Table 2 presents the number of benchmarks in the *Simplified* set solved by PeneLoPe using 2, 4 and 8 cores. Recall that PeneLoPe is a portfolio solver using homogeneous solver threads and clause sharing. This type of portfolio is expected to perform best on formulas that are satisfiable, as compared to unsatisfiable formulas the run time deviations between multiple runs of a similar solver are larger [19]. PeneLoPe witnesses this by solving more satisfiable benchmarks using four threads than it does using two threads, given the same amount of CPU time. Clearly, GlucoRed and PeneLoPe have orthogonal strengths. Given the same amount of CPU time GlucoRed can prove more benchmarks unsatisfiable than PeneLoPe, regardless of whether 2, 4 or 8 threads are used for PeneLoPe. For unsatisfiable benchmarks the two threaded solver GlucoRed is so much more efficient that in 900 seconds of wall clock time it solves six unsatisfiable benchmarks more than PeneLoPe does using four threads.

5 Conclusions

This work presents the solver-reducer architecture, which employs strengthening of conflict clauses in parallel with CDCL search in a modern SAT solver. An extensive empirical evaluation showed the good performance of this conceptually simple idea, which can be combined with conventional parallelization strategies.

The use of concurrency to aid conventional sequential CDCL search, rather than to parallelize that search, has not been suggested before. This simple but novel idea can be exploited in many different ways. For example, a logical next step would be to consider concurrent formula simplification. This would be a natural way of employing concurrency in recent solvers that use *inprocessing*, i.e. the sequential interleaving of solving and simplifying procedures.

References

1. Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.-M., Piette, C.: Revisiting clause exchange in parallel SAT solving. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 200–213. Springer, Heidelberg (2012)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI, pp. 399–404 (2009)
3. Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 118–126. Springer, Heidelberg (2012)
4. Balint, A., Belov, A., Diepold, D., Gerber, S., Järvisalo, M., Sinz, C. (eds.): Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2012-2. University of Helsinki, Helsinki (2012)
5. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artif. Intell. Res. (JAIR) 22, 319–351 (2004)
6. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Technical Report 10/1, Johannes Kepler University, Linz, Austria (2010)
7. Böhm, M., Speckenmeyer, E.: A fast parallel SAT-solver - efficient workload balancing. Ann. Math. Artif. Intell. 17(3-4), 381–400 (1996)
8. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC, pp. 151–158. ACM (1971)
9. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
11. Eén, N., Sörensson, N.: MiniSat v1.13 - a SAT solver with conflict-clause minimization. Poster for SAT 2005 (2005), `http://www.minisat.se`
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
14. Van Gelder, A.: Generalized conflict-clause strengthening for satisfiability solvers. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 329–342. Springer, Heidelberg (2011)
15. Hamadi, Y., Jabbour, S., Piette, C., Sais, L.: Deterministic parallel DPLL. JSAT 7(4), 127–132 (2011)
16. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: A parallel SAT solver. JSAT 6(4), 245–262 (2009)
17. Han, H., Somenzi, F.: Alembic: An efficient algorithm for CNF preprocessing. In: DAC, pp. 582–587. IEEE (2007)
18. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 50–65. Springer, Heidelberg (2012)
19. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning search spaces of a randomized search. Fundam. Inform. 107(2-3), 289–311 (2011)
20. Hyvärinen, A.E.J., Manthey, N.: Designing scalable parallel SAT solvers. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 214–227. Springer, Heidelberg (2012)

21. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)
22. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Parallel SAT solver selection and scheduling. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 512–526. Springer, Heidelberg (2012)
23. Marques-Silva, J.P., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD, pp. 220–227 (1996)
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC, pp. 530–535. ACM (2001)
25. Piette, C., Hamadi, Y., Sais, L.: Vivifying propositional clausal formulae. In: Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N.M. (eds.) ECAI. Frontiers in Artificial Intelligence and Applications, vol. 178, pp. 525–529. IOS Press (2008)
26. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
27. Pipatsrisawat, K., Darwiche, A.: A new clause learning scheme for efficient unsatisfiability proofs. In: Fox, D., Gomes, C.P. (eds.) AAAI, pp. 1481–1484. AAAI Press (2008)
28. Schubert, T., Lewis, M.D.T., Becker, B.: PaMiraXT: Parallel SAT solving with threads and message passing. JSAT 6(4), 203–222 (2009)
29. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 237–243. Springer, Heidelberg (2009)
30. Wieringa, S., Heljanko, K.: Asynchronous multi-core incremental SAT solving. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 139–153. Springer, Heidelberg (2013)
31. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR) 32, 565–606 (2008)
32. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. 21(4), 543–560 (1996)