Journal on Satisfiability, Boolean Modeling and Computation 4 (2008) 75-97

PicoSAT Essentials

Armin Biere biere@jku.at
Johannes Kepler University,
Linz,
Austria

Abstract

In this article we describe and evaluate optimized compact data structures for watching literals. Experiments with our SAT solver PicoSAT show that this low-level optimization not only saves memory, but also turns out to speed up the SAT solver considerably. We also discuss how to store proof traces compactly in memory and further unique features of PicoSAT including an aggressive restart schedule.

KEYWORDS: *SAT solver, watched literals, occurrence lists, proof traces, restarts*

Submitted October 2007; revised December 2007; published May 2008

1. Introduction

Recent improvements in SAT solvers are fueled by industrial applications and the SAT Competition. They have their origin in new algorithms and heuristics but also in low-level optimization techniques. In the first part of this paper we present a careful evaluation of one low-level optimization that was implemented in the original mChaff SAT solver [28] but was abandoned in more recent solvers, starting with zChaff and including even the latest versions of MiniSAT [11]. To the best of our knowledge, this optimization has not been described nor evaluated in the literature.[1] A careful experimental analysis conducted with our SAT solver PicoSAT [4] shows that this optimized representation of occurrence lists gives considerable performance gains. It could have made the difference between winning SAT Race'06 [35] and taking 4th place, if these new version of PicoSAT would have taken part in the race. In the second part of the article we discuss additional features of PicoSAT, including decision heuristics, restart schedule, the generation of proof traces, and how proof traces can be stored compactly in memory.

2. Occurrence Lists

The first optimization we discuss, implemented in mChaff and independently in PicoSAT, is based on a real *list representation* of occurrence lists resp. watch lists. The list implementation is intrusive since new link fields are embedded into the linked clauses. In past state-of-the-art solvers, starting with zChaff, watch lists, despite their name, have actually been implemented with arrays, more precisely stacks on which references to clauses

1. The author of this article actually independently developed the same idea without being aware of its implementation in the original mChaff source code.

resp. watched literals are pushed. The operations on these stacks have similar semantics and complexity as those of vectors of the standard template library of C++: vectors are ordered, with constant time access operation, and their size can be changed dynamically, for instance by adding an element at the end. In the following we will use the term "stack" to denote a container with exactly these characteristics.

Our approach is similar to the idea of using specialized data structures for binary [30] or ternary clauses [33]. Carefully implementing this optimization *does not influence heuristics*. Identical search trees can be obtained, which makes experimental evaluation much easier and more robust. It avoids common statistical problems in benchmarking SAT solvers.

SAT solvers maintain an *occurrence list* for each literal. An occurrence list contains references to clauses in which a literal occurs. Recent solvers based on DPLL [9] learn many derived clauses [25]. For efficiency it is crucial to put only watched clauses on to the occurrence lists [38, 28], also called *watch lists* in this case. Since only two literals of each clause have to be watched, the average length of occurrence lists is reduced considerably compared to full occurrence lists, which contain all occurrences. Accordingly the number of clauses that have to be visited during boolean constraint propagation (BCP) decreases. The techniques discussed in this article are applicable to watch lists and full occurrence lists and we will use the more general term "occurrence lists".

In the experiments we ran PicoSAT in the version submitted to the SAT'07 SAT Solver competition, as if it would have entered the SAT-Race'06. More details are found in the experimental section. In the following we will make statements about various statistics obtained during these runs. Since these extended statistics are not available in the default fully optimized version of PicoSAT, we had to use the slightly slower version of PicoSAT, which generates statistical data. We could only take those benchmarks into account, which were solved by the slower version (2 less), since PicoSAT only dumps statistics if it successfully solves[2] an instance.

2.1 zChaff Occurrence Lists

Most of the current state-of-the-art SAT solvers follow zChaff [28] in implementing occurrence lists as *stacks*.[3] These stacks are in turn implemented as arrays of pointers. This design decision makes sense for zChaff, which actually does not store pointers to the clauses in the occurrence list, but more precisely pointers to those literal fields within the clause, which contain watched literals. Using singly linked lists in zChaff is therefore prohibitive, since it requires one additional link field for each single literal field of a clause, and thus doubles memory usage.

An example for the stack based implementation of occurrence lists as implemented in zChaff is shown in Fig. 1. For each literal, a stack is maintained that contains a pointer to the occurrence of this literal in those clauses in which it is watched. As in the DIMACS format we denote literals by integer variables. The literal -2 is watched in three clauses, first in the upper most clause of the figure, where it is watched together with -8, and in one ternary and one binary clause. Typically, for instance in the `vector` container of C++

2. Solving means proving satisfiability or unsatisfiability.
3. Note that the results reported in [28] are based on mChaff, though the source code of zChaff became available earlier and thus became the quasi standard until MiniSAT.

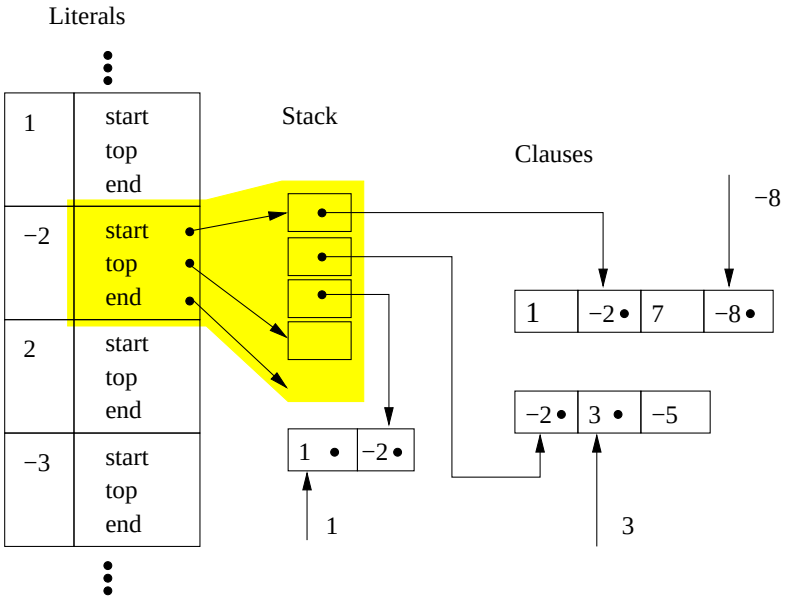

Figure 1. Occurrence lists implemented with stacks as in zChaff.

STL, the stack is implemented as an array, with a start, top and end pointer. The latter is used for efficient resizing of the array if top reaches end.

During BCP, if a literal is assigned, the occurrence list of its negation and the watches referenced in this list have to be updated. We count the process of updating the occurrence list of one literal as one *propagation*. If for instance 2 is assigned to *true*, and still 7 and 5 are unassigned, then 7 becomes watched in the upper clause and -5 in the lower clause. Then the first two watches in the stack have to be removed. The watches are removed while traversing the stack. Thus the removal does not incur any time penalty unless the average ratio of conflicts per propagation is high.[4.] Note, that if a conflict occurs while traversing the stack the rest of the stack still has to be copied if during the traversal of this stack a watch has been removed earlier.

The number of propagations during the run of a SAT solver can be smaller than the number of assignments, if, as most SAT solvers do, variables are assigned eagerly [6]. The number of clauses visited during this process, or just short the number of *visits*, is also a good indication of the amount of work performed by the SAT solver. The *average* number of visits per propagation is usually small, 4.16 with 2.93 as median in our experiments.

An important statistic accounts for the amount of time spent in searching for a new watch while visiting a clause during BCP. The number of literals which the inner most loop of the BCP procedure has to read from the clause during a visit is called the number of *traversals*. Surprisingly, this number is even smaller, 1.67 on average with 1.56 as median. This is in contrast to the large average length of learned clauses which is 31.75 literals on average with 27.80 literals as median. Note that these numbers are calculated after clause

4. In our experiments the maximum ratio was less than 0.7 percent, and most of the time, the number of conflicts is three orders of magnitude less than the number of propagations

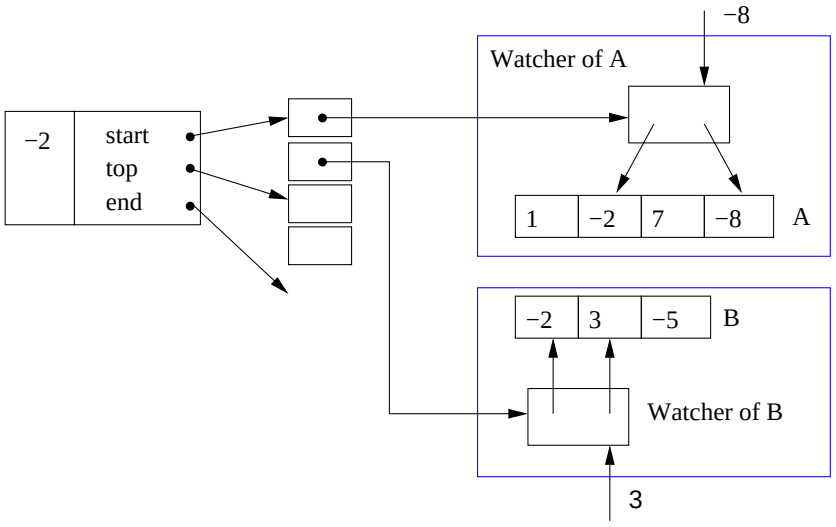

Figure 2. Stack based occurrence lists of watchers as in Limmat and FunEx.

minimization [1], as implemented in MiniSAT [11]. Clause minimization is able to remove 32% literals on average, which means that average clause length increases by almost 50% if clause minimization is disabled.

The cache behavior of modern microprocessors has a non negligible impact on the performance of SAT solvers [39]. In an earlier version of PicoSAT, low-level profiling revealed, that most of the time is spent in one assembler instruction, which corresponds to reading the head of a clause for the first time while it is visited during BCP. This is explained by the random access to clauses during SAT solving and the large amount of learned clauses. Thus the access to clauses can not be localized and the CPU spends a considerable amount of time waiting for clause data to arrive from caches and main memory.

2.2 Separate Watcher Data Structures

While visiting a clause in the original zChaff scheme the position in the clause of the other watched literal can be far away from the position of the propagated literal. However, it seems to be a good heuristic to check, whether the other watched literal already satisfies the clause. In this case the traversal can stop and the watch does not have to be updated.

In order to implement this optimization the clause or at least each element on the occurrence lists, e.g. a "watcher", needs to either know both watched literals or their position within the clause as shown in Fig. 2. This is how occurrence lists in our earlier SAT solvers Limmat and FunEx [6] were implemented. More recently a similar technique has been used to separate watch lists of different threads in the multi-threaded SAT solver MiraXT [24].

The drawback of separate watcher data structures is the additional pointer dereference needed to actually access the literals of the clause, even if the watcher is embedded in the clause. If the two watched literals are far away, another potentially non local memory access is needed to apply the optimization of the first paragraph.

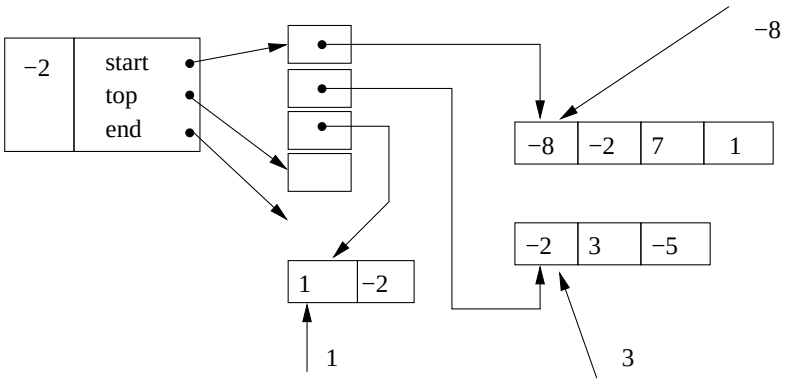

Figure 3. Keeping watched literals at the beginning of clauses as in CompSAT and MiniSAT.

Even though the watchers are small and probably their access can be localized, the additional pointer dereference, precisely at the bottle neck of modern SAT solvers, seems to be too much of an overhead.[5.]

2.3 Keeping Watched Literals as First and Second Literal

The costs of separate watchers can be avoided by following the suggestion made in [15], which is inspired by the implementation of the original mChaff [28], and implemented in CompSAT [6] and MiniSAT [11]. The idea is to move and keep the two watched literals at the beginning of the clause, by exchanging literals instead of moving watches. In contrast to the implementation of occurrence lists in zChaff, keeping the watched literals at the beginning of the clause, allows to put pointers to clauses instead of pointers to literal occurrences within clauses into the watch lists as shown in Fig. 3.

This arrangement avoids additional bits[6.] in the literal fields for marking watched literals and also makes it very efficient to find the "other" watched literal in a clause if a clause is visited during BCP of an assigned literal. The "other" literal can be tested in constant time whether it already satisfies the clause, which actually happens fairly frequently in practice, and avoids updating the watches. In our experiments the "other" literal was *true* in on average 61% of the visited clauses during BCP.

Furthermore, in most SAT solvers the clause memory contains nothing more than the literals or at least the literals are embedded directly into the clause structure, right after a short clause header. Fetching the clause header from memory will also fetch some of its literals, which can then be cheaply accessed by the CPU.

2.4 Real Occurrence Lists in mChaff and PicoSAT

This leads us to the *main observation* of the first part of the paper. Keeping the two watched literals at the beginning of the clause as described in [15] allows to cheaply add two link

5. It is an open question whether this cost can be avoided for a multi-threaded SAT solver.
6. Put into the literal word by bit stuffing techniques – the dots in Fig. 1. Otherwise the search for a new unwatched literal in updating a watch would not know the other watched literal.

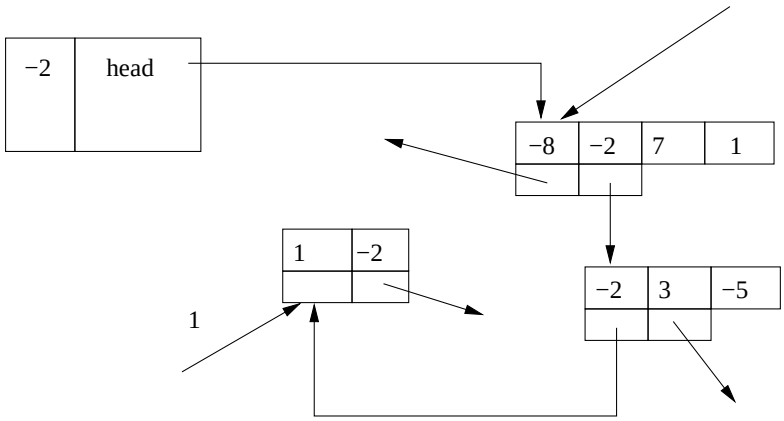

Figure 4. Real Occurrence Lists as in mChaff and PicoSAT.

fields to the clause, one for each watched literal.[7] These link fields contain pointers to the next watched clause of the two watched literals in the clause and thus connect the clause to two *singly linked lists*, one for each watched literal. The anchor of such an occurrence list for a literal is just a pointer to the first clause in which this literal is watched. Figure 4 is an example.

The other clauses can be reached recursively following the link fields. This arrangement saves heap memory for the stacks in the old scheme. It also avoids an additional pointer lookup while visiting a clause during BCP. The link fields are put in front of the literals in the header of the clause. The header of a clause contains beside the two link fields various other flags and includes the size of the clause.

At first sight it seems to be unclear *which link field to use* in the traversal of occurrence lists. In the original mChaff implementation, the position of the literal was encoded in the pointer to the clause by encoding the position as the least significant bit of the pointer. Similar bit stuffing techniques were used in Limmat.

Another simple solution, as implemented in PicoSAT, is to order the pointers stored in the link fields in the same way as the two-watched-literals: the link for the occurrence list of the first watched literal in the clause is stored in the first link field and the link for the second literal in the second field. While traversing the occurrence list of a literal l, and visiting a clause, in which l is watched, we compare l with the first two literals in the clause. If l matches the first literal, then the pointer to the next clause in which l is watched is stored in the first link field, if the second matches, then in the second link field.

2.5 Special Treatment of Binary Clauses

It has been observed in [30] that for binary clauses the concept of two watched literals does not make sense. The position of the watches will never change. Furthermore, if the negation of a literal in a binary clause is propagated, then the other literal in this clause will need

7. This idea was also implemented in mChaff, but never described in the literature before. Apparently, in contrast to Allen Van Gelder [15], the author of PicoSAT was not aware of this particular difference between mChaff and zChaff, and developed the same idea independently.

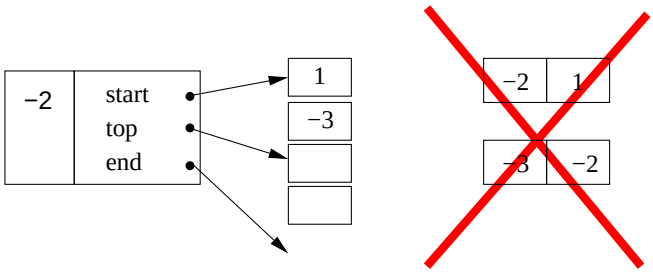

Figure 5. Special watches for binary clauses.

to be assigned to true in any case. Dereferencing the clause is just overhead that can be avoided by saving the other literals in the occurrence lists instead of a pointer to the binary clause. This saves one pointer dereference for every visit to a binary clause.

The basic idea is shown in Fig. 5. Here a stack is much more cache efficient. Mixing binary clause watches with ordinary watches requires bit stuffing as in MiniSAT. In PicoSAT we have separate stacks of watches for binary clauses and for larger clauses. This separation avoids bit stuffing and also allows to visit binary clauses first. Actually, binary clauses of all assigned variables are visited before any larger clause in PicoSAT.

In the Siege SAT solver [33], this concept is extended to ternary clauses, which is a good idea to pursue, but has not been implemented in PicoSAT yet. Also note that handling ternary clauses this way, requires three watches for each ternary clause. It also increases memory usage slightly, and more severely will result in more ternary clauses being visited.

Note, clauses for which watches are "inlined" as just described, do not have to be stored unless we need them for core or proof trace generation. If PicoSAT is compiled with support for proof tracing, then this technique is not used.

3. PicoSAT

Our SAT solver PicoSAT [4] is an attempt to further optimize low-level performance compared to BooleForce [3], which in turn shares many of its key features with MiniSAT 1.14 [11]. The latter is the back end of SATeLiteGTI, the winner in the industrial category of the SAT Competition'05. It uses SATeLite [10] as preprocessor. As it has been shown in [10], most state-of-the-art solvers can benefit from such preprocessing and therefore we focus in this paper on low-level optimizations of the back end. The latest available version 2.0 of MiniSAT, which is also the one that entered the SAT'07 SAT solver competition and performed well in the industrial category, is a reimplementation of SATeLiteGTI, which integrates the algorithms of SATeLite directly into MiniSAT. Therefore, it is probably more instructive to compare PicoSAT with MiniSAT 1.14.

3.1 Decision Heuristics

The decision heuristic of PicoSAT follows ideas implemented in RSAT [32]. The decision variable is selected as in MiniSAT 1.14, which is a more dynamic and adaptive version of the original zChaff decision heuristics. While MiniSAT 1.14 always assigns the decision

variable to *false*, RSAT [32] assigns the decision variable to the same value it has been assigned before. Initially, as long a variable has not been assigned yet, PicoSAT uses the Jeruslow-Wang heuristics [21] for selecting the phase of the decision variable.[8] This seems to be slightly better than using the plain number of occurrences of a literal.

We contribute the success of PicoSAT in the category of satisfiable industrial instances of the SAT'07 SAT Solver Competition to the new restart and phase assignment heuristics, beside the fast and efficient low-level data structure discussed in the previous section.

3.2 Restart Schedule

Beside the list based representation of occurrence lists, as described above, PicoSAT uses an aggressive *nested restart* scheme, inspired by, but simpler than [23], in combination with a more sophisticated *strategy for picking the phase of decision variables*. The restart strategies of RSAT [32] and TiniSAT [20] are similar.

The nested restart scheme, with pseudo-code in Fig. 6, triggers fast restarts with a high frequency. In Fig. 7 an initial prefix of its schedule is shown. The period of fast restarts is increased by 10% after every restart until the end of the outer long period with a slow frequency. Then the long period of the outer restart interval is also increased by 10% and the fast restart interval is reset to its initial period of 100 conflicts.

In addition, to avoid revisiting the same search space over and over again, the last learned clause before a restart is *fixed* and never deleted. Other learned clauses are garbage collected in the reduction phase based on their activity as usual [17]. The implementation follows MiniSAT. At the end of (in our case *outer*) restart interval the limit on the number of live learned clauses is increased by 5%. If this limit is hit, half of the learned clauses are discarded in a garbage collection phase. Our attempts to decouple restarts from increasing the reduce limit were not successful yet.

The 10% increment resp. the *restart factor* of 1.1 has been determined empirically. It produces the best results on our benchmarks. An open question is how to adapt it dynamically.

3.3 Why are Rapid Restarts Beneficial?

Our understanding why aggressive restarts as implemented in PicoSAT can speed up SAT solving considerably, particularly in the context of industrial benchmarks, is incomplete. Nevertheless we try to give some reasons.

3.3.1 Heavy-Tail Behavior

Industrial or structural benchmarks are large but often either easy to satisfy or to refute. Therefore randomizing the search should help the solver not to get lost in those parts of the search space that do not lead to a fast solution or refutation. This kind of heavy-tail behavior was also the motivation in the original publication on restarts [19]. Note, however that rapid restarts combined with phase saving, except for the rare occurrence of random decisions, only change the *order* in which decision variables are picked.

8. In calculating the Jeruslow-Wang heuristics only original clauses are taken into account.

```
int inner = 100, outer = 100;
int restarts = 0, conflicts = 0;

for (;;)
  {
    ... // run SAT core loop for 'inner' conflicts

    restarts++;
    conflicts += inner;

    if (inner >= outer)
      {
        outer *= 1.1;
        inner = 100;
      }
    else
      inner *= 1.1;
  }
```

Figure 6. Pseudo-code for PicoSAT restart schedule.

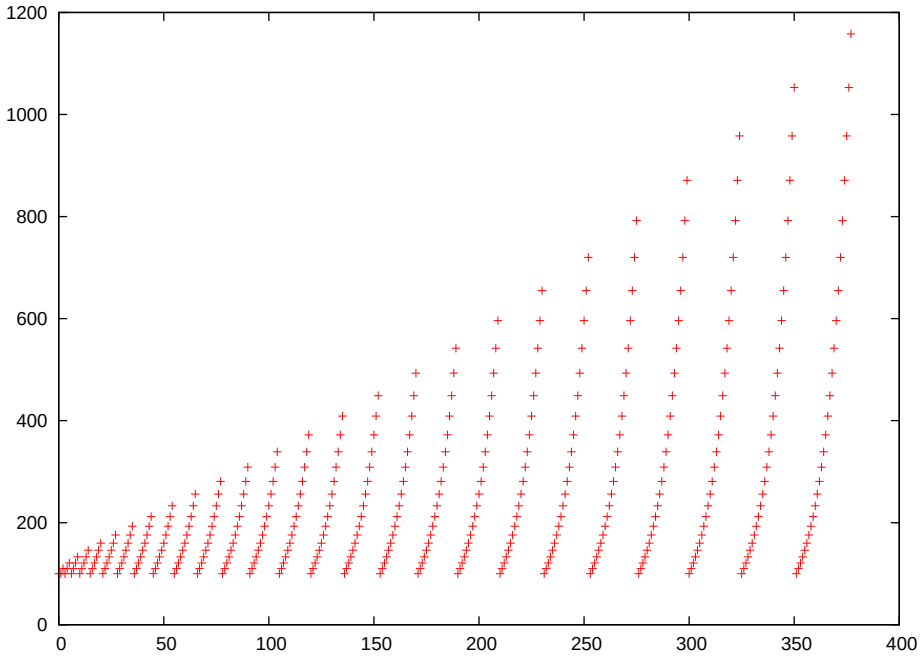

Figure 7. An initial prefix of the PicoSAT restart schedule. On the horizontal axis the number of restarts is shown, where we do not differentiate between outer an inner restarts. The vertical axis denotes the number of conflicts of the restart interval, which more precisely is the number of conflicts that have to occur before the next restart is triggered. The last restart in this schedule occurs after 104408 conflicts.

3.3.2 Bad Decision and Propagation Order

Techniques for *learned clause shrinking* allow to produce shorter learned clauses. Shorter learned clauses not only need less space, but also, if clause shrinking is implemented carefully, will reduce search. Originally, implemented in JeruSAT [29] and in the 2004 version of zChaff [12] clause shrinking was based on unassigning the literals of a learned clause and then reassigning them one after the other until another conflict occurs. The resulting learned clause is often shorter than the original learned clause.

The original clause shrinking technique is expensive and partially subsumed by the new conflict clause shrinking technique of MiniSAT 1.14, also implemented in PicoSAT. This new technique, also referred to as *clause minimization* [1], essentially removes literals from a learned clause, by resolving recursively with clauses of the implication graph.

In any case, the effectiveness of clause shrinking resp. minimization in practice shows, that current SAT solvers are far away from picking good decisions and good propagation paths. Restarts help the SAT solver to recover from these mistakes. After a restart the SAT solver can choose better decision variables, even if they are just picked in a different order, and propagate those decisions and their implied assignments more effectively. Rapid restarts thus simulate conflict clause shrinking.

3.3.3 Locality through Phase Saving

Most state-of-the-art solvers implement a conflict-driven assignment loop [25], which after learning a unit clause, enforces backtracking to the top-level. It was observed in [7, 31] that not only units are learned quite frequently, but also that these units often break the top level formula into disconnected components. If a component becomes satisfied and a new unit is learned the satisfying assignment is erased unless either the component is detected and the assignment explicitly saved as in CompSAT [7] or by simply always assigning a decision variable to its previously assigned value as in RSAT [31] and PicoSAT. In [31] it has been argued that disconnected components are also produced deeper in the search tree. Saving and reusing the phase will also help to speed up search in these cases.

Saving phases turbo charges restarts. First the search will almost continue where it stopped, unless a random decision is triggered. Furthermore, the same set of clauses is visited, which increases cache efficiency. Additionally we conjecture that rapid restarts combined with phase saving increases the likelihood that the other watched literal is true while visiting a clause.

4. Proofs

Another important feature of modern SAT solvers is their ability to produce proof traces [40, 18, 13, 36, 22, 16]. Earlier work on generating more verbose resolution proofs but without experiments can be found in [14].

These proof traces are used in many applications. For instance, in declarative modeling or product configuration [34, 37] an inconsistent specification corresponds to an unsatisfiable problem instance. From a proof trace it is possible to extract a justification why the specification is erroneous. In the context of model checking proofs are used for abstraction refinement [27]. Given an abstract counter example, which can not be concretized, assume

that the failed attempt to concretize the counter example has a refutation proof generated by a SAT solver. Then the refinement consists of adding additional constraints that can either be traced back to clauses that are used in a refutation, the *clausal core*, or to variables contained in these clauses, *the variable core*. In approximative image computation, or more general in over approximation of quantifier elimination, a resolution proof obtained from a proof trace allows to generate interpolants [26].

Proofs are important for certification through proof checking [40, 18, 13, 16] or just simply for testing of SAT solvers. In these two applications, a proof object does not actually have to be generated. In principle, the proof checker can check the proof during the SAT run, by inductively showing that each learned clause is implied by the original clauses and all previously learned clauses. As it has been argued in [18, 16] and also independently implemented in Limmat [6], this check just requires unit propagation as long SAT solvers do not use more sophisticated BCP during SAT solving.

In order to trace a proof we need to save learned clauses together with their lists of antecedents that were used to derive them [40, 18] either on disk or in memory. The result is an acyclic clause graph with original clauses as leafs and the empty clause as one root assuming a refutation has been produced. In this case the clausal core is made of the original clauses reachable from the root. Since this graph traversal can not be started before the empty clause has been learned, in general, all learned clauses together with their antecedents have to be saved.

PicoSAT's predecessor BooleForce [3] has already been able to keep the proof trace in memory, which greatly improves performance in applications, where clausal or variable cores or a proof trace have to be produced frequently.[9] In these applications in-memory proof traces are an order of magnitude more efficient than writing traces to disk and reading them back as it is necessary for proof logging versions of zChaff and MiniSAT.

4.1 Garbage Collecting Learned Clause during Proof Trace Generation

Since proof traces of SAT solvers can grow very large, we employed two techniques to reduce space usage. First, clauses that become satisfied and never were used in deriving a conflict can safely be deleted. In principle, one could even go further and use reference counters for learned clauses. Clauses which are not referenced anymore can also be deleted. In our experience, even just using such a "used" flag as an over approximation of proper reference counting already allows to remove many clauses.

4.2 Compressing Antecedents by Delta Encoding

Another size reduction is achieved by sorting clause indices of antecedents of learned clauses, and then compressing them by just storing the deltas, followed by a simple byte stream encoding. In this encoding the most significant bit of a byte denotes the end of a delta, as in the binary AIGER format [2]. In practice we obtain compression ratios close to one byte per antecedent.

As example consider a learned clause with antecedents 100, 500, and 501. These clause ids are already sorted, the delta which we need to save are 100, 400, and 1. The first delta

9. See for instance [8].

fits in 7 bits and is encoded as one byte with value 100. The second delta needs two bytes

$$244 \;=\; (500 \;\%\; 128) \mid 128 \qquad \text{and} \qquad 3 \;=\; (500 \;/\; 128)$$

similarly the third delta also just needs one byte. Together with a zero sentinel, the antecedents of this clause are thus encoded with the 5-byte sequence 100, 244, 3, 1, 0. Using 4 bytes for each antecedent and the sentinel would lead to 16 bytes. In practice the antecedents of learned clauses tend to contain many recently learned clauses, thus improving locality and increasing efficiency of delta encoding.

In order to use delta encoding we require clause ids, in form of a unique integer identifier for each clause. It could be argued that clause ids are an overhead if one is only interested in cores. These ids have to be mapped to learned clauses and vice versa. However, when dumping proof traces such ids have to be generated anyhow. In addition, our compression techniques save much more space than what these ids consume. Finally, PicoSAT only uses clause ids if during initialization it is instructed to trace proofs.

4.3 Sorting Antecedents for Resolution Proof Generation

The fact that ids are reordered seems to be contra-productive to some applications that actually need to generate a resolution proof, such as certification [14], testing of SAT solvers, and generation of interpolants [26]. The problem is that generating an actual resolution proof requires to order the antecedents of each learned clause in such a way that they can be resolved in sequence to form a regular input resolution proof, called trivial proof in [1], of the learned clause.

We developed a tool TraceCheck [5], which is now part of the BooleForce source distribution, to solve this problem. It reads a proof trace, and reorders the antecedents of all learned clauses, in order to check the trace by resolution steps, or to dump a resolution proof. The first version produced a regular input resolution for each learned clause directly by reverse resolution steps. The current implementation simply uses unit propagation as described in [18] and independently implemented in Limmat [6] and recently refined in [16] to sort the antecedents.

In details, the algorithm in TraceCheck works as follows. First, the antecedent graph is sorted globally and checked for cycles. Then the antecedents of each individual derived clause have to be reordered. Its antecedents are watched and the negation of the literals in the derived clause are assumed. Then these units are propagated until an empty clause under the current assignment is found. If no empty clause is generated, then the derived clause can not be resolved with regular input resolution using the given antecedents as input clauses.

We use a variable trail, on which assigned variables are stored for backtracking purposes. It is also used to record the order in which variables are assigned. The trail is traversed in reverse chronological order. If an assignment of a variable has been forced by an antecedent clause then this clause is put next in the resolution order. Thus the antecedents are sorted in reverse order how they are used as "reasons". This order can be used to resolve the target clause from the antecedents through regular input resolution from the clause that became empty.

Table 1. Overall comparison of the five versions explained in the caption of Tab. 2. The column with the sum of the run-time for all 100 benchmarks includes 900 seconds for each benchmark on which the time limit of 900 seconds was exceeded. The memory usage is shown in the last column and is calculated as the sum of the maximum number of main memory over all runs.

version	solved	unsolved	sum time (seconds)	sum space (MB)
`list2`	78	22	38240	5793
`stack2`	76	24	40334	6768
`list`	72	28	41345	6917
`stack`	67	33	43510	8677
`trace`	67	33	43222	16950

Regularity of the proof, which means that variables are resolved at most once, is the key and naturally follows from the fact that during unit propagation a variable is not assigned twice.

Unused antecedents are simply not resolved at all. This allows the proof generator to use super-sets of those antecedents that are needed. This simplifies proof-generation in more complex situations [36, 22] without sacrifying correctness and the ability to produce resolution proofs out of proof traces. In principle, with a quadratic blow-up, this relaxation also allows to simulate [18], the internal proof checker in Limmat, and the RUP format of [16].

This is the setup we used in the certification track of the SAT'07 SAT solver competition for the version of PicoSAT that was most successful in producing certificates for large industrial formulas. The effectiveness of PicoSAT was of course harmed by the requirement to generate resolutions proofs, consisting of individual resolution steps. These resolution proofs are usually orders of magnitude larger than their proof traces, though admittedly, checking them is much easier.

5. Experiments

In the experiments we used the same setup and the same benchmark set as in the SAT Race'06 [35]. Our cluster has 15 identical computing nodes, with Pentium IV 3 GHz CPUs, 2 GB of main memory, running Ubuntu Linux. The set of 100 benchmarks can be considered as typical structural industrial benchmarks. We used version 535 of PicoSAT which is the same that entered the SAT'07 SAT Solver Competition.

PicoSAT can be compiled to either use a stack based or list based occurrence list implementation. It is also possible to use a more compact representation of binary clauses as in [30]. These two features are independent and together with a version that generates proof traces in memory results in five versions of PicoSAT. These versions and their acronyms are explained in the caption of Tab. 2 and the overall results are documented in Tab. 1.

The performance gain of the list based implementations is in the same order as the speed-up that can be obtained by treating binary clauses as in [30]. A detailed comparison

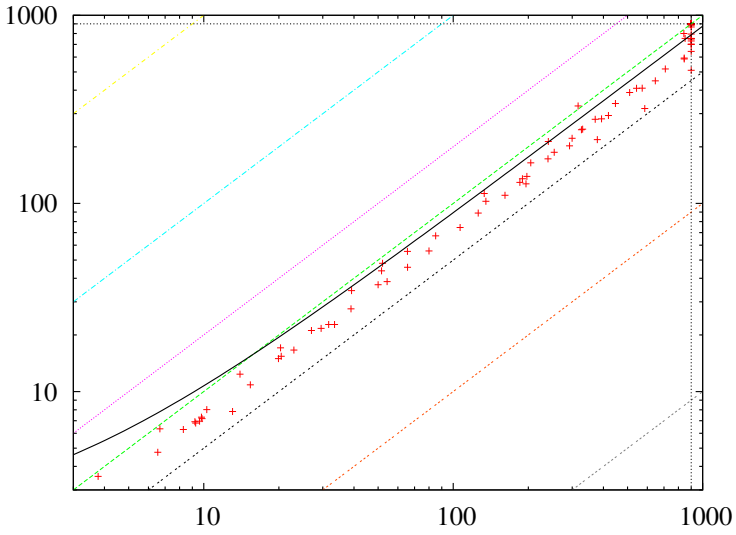

Figure 8. Scatter plot for the run-times of PicoSAT on the SAT-Race'06 instances. The run-times of version `stack` are on the horizontal axis and of `list2` on the vertical axis.

of the run-time of these five versions of PicoSAT is shown in Tab. 2 and Tab. 3. MiniSAT version 2.0, the winner of SAT Race'06, solved 73 benchmarks. PicoSAT, even without preprocessor, turns out to be faster.

In another set of experiments we disabled various features of PicoSAT. The results are reported in Tab. 4 with more details in Tab. 5 and Tab. 6. Note that these runs do *not* have the same search space and thus have to interpreted more carefully than those of Tab. 2 and Tab. 3.

5.1 Discussion

Using optimized data structures for watches gives a considerable benefit. There is a large difference between stack based occurrence lists `stack` and the list based implementation `list2` which in addition also uses special data structures for binary watches. This result is confirmed by the scatter plot in Fig. 8.

To make sure that the search space of all different occurrence list implementations is the same we checked that the number of decisions, conflicts, propagations and visits are the same in all versions.

The experiments also reveal, that optimizing the low-level representation of occurrence lists is not always beneficial, even though identical search trees are traversed. For highly constrained benchmarks of a more combinatorial nature with rather few variables, the classical stack based implementation turns out to be faster. Examples are the benchmarks from the `goldb` and `grieu` suite.

Since we have precisely the same overall behavior with the stack and the list based implementation it made sense to profile one run against the other. Using different seeds for the random number generator would have made such a comparison very difficult. Profiling

Table 2. PicoSAT on the SAT Race'06 benchmarks, with a time limit of 900 seconds, and a space limit of 1.5 GB. Column `list2` uses a list based implementation for watched literal lists as in mChaff and treats binary clauses in a special way [30]. Also `list` uses lists but treats binary clauses as ordinary clauses. Columns `stack2` and `stack` are similar but use stacks instead of lists. The version of PicoSAT used in column `trace` use the same data structures as `list`, but generates a compressed proof trace in memory.

	trace	stack	list	stack2	list2
ibm-2002-05r-k90	35.78	38.93	32.74	30.82	**27.54**
ibm-2002-07r-k100	8.53	9.79	7.45	8.56	**7.31**
ibm-2002-11r1-k45	163.22	185.10	150.55	150.40	**129.40**
ibm-2002-19r-k100	519.70	572.24	477.96	468.00	**410.01**
ibm-2002-21r-k95	351.75	393.35	321.27	325.04	**281.74**
ibm-2002-26r-k45	7.26	6.67	6.42	6.66	**6.34**
ibm-2002-27r-k95	26.86	29.54	24.36	24.75	**21.66**
ibm-2004-03-k70	48.69	54.32	44.74	45.18	**38.43**
ibm-2004-04-k100	221.47	239.96	202.50	195.40	**172.50**
ibm-2004-06-k90	95.99	106.61	89.89	87.36	**74.48**
ibm-2004-19-k90	391.68	419.27	362.31	335.11	**293.49**
ibm-2004-1_11-k25	14.12	15.37	13.20	12.47	**10.86**
ibm-2004-1_31_2-k25	188.50	197.39	164.07	158.66	**139.24**
ibm-2004-26-k25	4.00	3.77	3.62	3.72	**3.54**
ibm-2004-2_02_1-k100	20.76	22.98	19.12	19.09	**16.62**
ibm-2004-2_14-k45	29.74	33.46	27.31	26.84	**22.73**
ibm-2004-3_02_1-k95	2.63	2.79	2.42	2.43	**2.19**
ibm-2004-3_02_3-k95	8.50	9.21	7.85	7.86	**6.91**
ibm-2004-3_11-k60	---	---	---	887.29	**794.18**
ibm-2004-6_02_3-k100	8.45	9.28	7.81	7.86	**6.80**
manol-pipe-c10id_s	7.59	8.28	6.95	7.20	**6.29**
manol-pipe-c10nidw_s	249.48	292.62	236.05	236.14	**202.31**
manol-pipe-c6nidw_i	---	---	822.80	897.06	**752.67**
manol-pipe-c7b	53.89	65.58	47.99	56.86	**45.79**
manol-pipe-c7b_i	65.28	80.01	58.57	69.89	**55.91**
manol-pipe-c7bidw_i	---	---	---	---	---
manol-pipe-c7nidw	---	---	---	---	---
manol-pipe-c9	7.92	9.62	7.15	8.70	**6.96**
manol-pipe-c9nidw_s	137.65	161.21	130.44	128.85	**110.43**
manol-pipe-f10ni	---	---	---	---	---
manol-pipe-f6bi	5.64	6.54	5.14	5.62	**4.76**
manol-pipe-f7idw	733.11	843.60	672.09	690.68	**586.18**
manol-pipe-f9b	---	---	---	---	---
manol-pipe-f9n	---	---	---	---	---
manol-pipe-g10b	109.37	125.93	98.02	105.64	**88.82**
manol-pipe-g10bidw	788.85	---	727.39	758.79	**642.67**
manol-pipe-g10id	166.08	189.55	151.60	156.87	**135.33**
manol-pipe-g10nid	735.80	843.51	669.31	696.59	**592.59**
manol-pipe-g6bi	2.13	2.36	1.89	2.13	**1.84**
manol-pipe-g7nidw	27.37	31.67	24.89	26.92	**22.71**
aloul-chnl11-13	---	---	---	---	---
een-pico-prop01-75	2.82	3.09	2.53	2.94	**2.52**
een-pico-prop05-50	43.17	49.99	38.67	44.71	**36.96**
een-tip-sat-nusmv-t5.B	17.42	19.92	16.18	17.22	**14.96**
een-tip-sat-nusmv-tt5.B	17.84	20.44	16.51	17.66	**15.40**
een-tip-uns-nusmv-t5.B	8.46	9.84	7.75	8.44	**7.18**

Table 3. PicoSAT on the SAT Race'06 instances cont. from Tab. 2.

	trace	stack	list	stack2	list2
goldb-heqc-alu4mul	---	840.46	807.84	**795.29**	800.56
goldb-heqc-dalumul	---	---	---	---	---
goldb-heqc-desmul	74.93	85.04	70.44	77.55	**67.27**
goldb-heqc-frg2mul	193.82	204.57	168.86	188.78	**164.30**
goldb-heqc-i10mul	---	850.51	781.94	784.76	**753.49**
goldb-heqc-i8mul	---	---	---	---	---
goldb-heqc-term1mul	267.26	240.57	218.11	226.77	**213.41**
grieu-vmpc-s05-25	17.99	13.97	14.85	**12.25**	12.38
grieu-vmpc-s05-27	68.38	52.10	57.79	**46.26**	48.10
grieu-vmpc-s05-28	435.21	316.88	373.18	**288.98**	329.56
grieu-vmpc-s05-34	---	---	---	---	---
hoons-vbmc-lucky7	9.51	10.28	**7.95**	9.48	8.02
maris-s03-gripper11	51.76	51.61	46.89	47.26	**43.85**
narain-vpn-clauses-6	---	---	---	854.12	**753.68**
schup-l2s-guid-1-k56	881.58	---	774.41	814.37	**699.22**
schup-l2s-motst-2-k315	---	---	---	---	**885.23**
simon-s02-w08-18	617.17	646.22	584.89	496.08	**448.21**
simon-s02b-dp11u10	---	---	---	---	---
simon-s02b-k2f-gr-rcs-w8	---	---	---	---	---
simon-s02b-r4b1k1.1	29.09	27.03	23.71	23.17	**21.11**
simon-s03-fifo8-300	307.96	330.42	291.54	271.20	**247.99**
simon-s03-fifo8-400	508.96	543.01	483.26	446.35	**409.51**
vange-col-abb313GPIA-9-c	---	---	---	---	---
vange-col-inithx.i.1-cn-54	18.27	20.32	17.22	20.01	**17.08**
mizh-md5-47-3	---	---	---	---	---
mizh-md5-47-4	861.74	---	734.97	849.87	**703.03**
mizh-md5-47-5	---	---	785.55	---	**747.32**
mizh-md5-48-2	---	---	---	---	---
mizh-md5-48-5	---	---	---	---	---
mizh-sha0-35-2	250.12	299.80	225.70	272.94	**221.90**
mizh-sha0-35-3	391.71	447.09	348.67	406.48	**339.62**
mizh-sha0-35-4	212.33	253.97	191.24	230.44	**187.10**
mizh-sha0-35-5	321.00	371.02	287.28	339.75	**280.16**
mizh-sha0-36-2	447.82	509.72	398.25	463.50	**388.62**
velev-engi-uns-1.0-4nd	225.03	195.81	183.93	143.33	**127.01**
velev-engi-uns-1.0-5c1	14.49	13.04	12.74	9.04	**7.84**
velev-fvp-sat-3.0-b18	136.44	135.24	126.94	112.67	**102.62**
velev-live-uns-2.0-ebuf	---	708.71	734.76	539.14	**518.45**
velev-npe-1.0-9dlx-b71	---	---	---	526.42	**510.05**
velev-pipe-o-uns-1.0-7	---	---	---	---	---
velev-pipe-o-uns-1.1-6	---	---	---	---	---
velev-pipe-sat-1.0-b10	737.52	585.93	669.79	337.18	**319.61**
velev-pipe-sat-1.0-b7	488.09	378.05	440.44	230.08	**218.38**
velev-pipe-sat-1.0-b9	72.97	65.55	65.67	56.62	**55.59**
velev-pipe-sat-1.1-b7	43.34	39.13	39.05	35.81	**34.41**
velev-pipe-uns-1.0-8	---	---	---	---	---
velev-pipe-uns-1.0-9	---	---	---	---	---
velev-pipe-uns-1.1-7	---	---	---	---	---
velev-vliw-sat-2.0-b6	356.51	326.62	329.90	260.00	**246.47**
velev-vliw-sat-4.0-b1	---	---	---	764.64	**730.80**
velev-vliw-sat-4.0-b3	153.81	133.14	139.36	117.92	**112.91**
velev-vliw-sat-4.0-b4	---	---	---	893.40	**867.02**
velev-vliw-uns-2.0-iq4	---	---	---	---	---
velev-vliw-uns-4.0-9C1	---	---	---	---	---

Table 4. Overall results for disabling various features of PicoSAT. See Tab. 5 for more details.

version	solved	unsolved	sum time (seconds)	sum space (MB)
`list2`	78	22	38240	5793
`norandom`	73	27	40717	6036
`restart2`	60	40	49154	7687
`norestart`	49	51	56250	7012

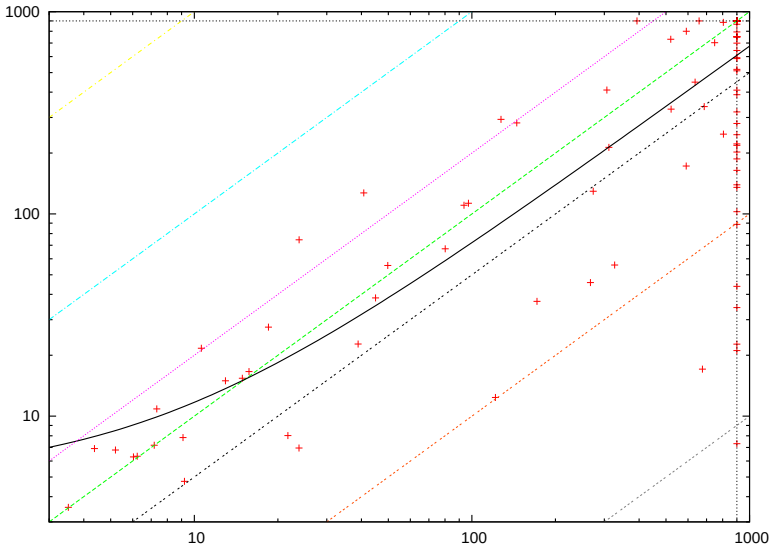

Figure 9. Run-times without restarts are shown on the horizontal axis and are compared to the run-times for the base case `list2` on the vertical axis.

revealed that in these instances the largest amount of time is still spent in BCP. But a non negligible portion of the run time is spent disconnecting satisfied or less active clauses.

Another observation can be made with respect to disabling various features of PicoSAT. As Fig. 9 shows the performance of PicoSAT degrades considerably with less restarts or no restarts at all. Restarts really seems to be a must. This conclusion only applies to industrial, highly structured instances. Our experience with combinatorial instances is the opposite and suggests to adapt the restart schedule dynamically.

However, as a comparison of Fig. 8 with Fig. 9 shows, the results of these experiments have to be interpreted very carefully, because these versions of PicoSAT traverse different parts of the search space and SAT solvers show chaotic behavior if heuristics are only changed slightly. In addition, as explained before, restarts are closely related to the limit on the number of live clauses. If for instance restarts are disabled, then the limit on the number of live learned clauses is kept constant to the initial reduce limit, which in PicoSAT 535 is chosen as one fourth of the number of original clauses of at least length 3.

Table 5. In further experiments we disabled various features of PicoSAT to measure their impact on performance. The setup is the same as in Tab 2. In the base case, the fastest version of the previous experiments, which is `list2`, we target 1% random decisions but always stick to the phase selection heuristics discussed above. The time point, when to pick a random decision, is randomized and also which variable to choose. Picking the phase randomly, even just once in a while, seems to degrade performance. In a second experiment (`norandom`) we did not use any random decisions. In the third experiment (`restart2`) we increased the inner and outer restart interval less aggressively by a factor of 2 as in [20]. Finally restarts are disabled (`norestart`).

	list2	norandom	restart2	norestart
ibm-2002-05r-k90	27.54	22.84	23.84	**18.49**
ibm-2002-07r-k100	**7.31**	7.48	29.96	---
ibm-2002-11r1-k45	129.40	104.76	**32.89**	273.50
ibm-2002-19r-k100	410.01	626.19	533.57	**306.16**
ibm-2002-21r-k95	281.74	459.57	388.09	**145.10**
ibm-2002-26r-k45	6.34	**6.11**	7.16	6.23
ibm-2002-27r-k95	21.66	14.71	19.58	**10.60**
ibm-2004-03-k70	38.43	24.19	**15.01**	44.99
ibm-2004-04-k100	**172.50**	288.87	262.02	591.18
ibm-2004-06-k90	74.48	31.27	**15.32**	23.86
ibm-2004-19-k90	293.49	369.22	217.64	**127.28**
ibm-2004-1_11-k25	10.86	7.70	8.34	**7.33**
ibm-2004-1_31_2-k25	**139.24**	172.37	---	---
ibm-2004-26-k25	3.54	**3.34**	3.94	3.52
ibm-2004-2_02_1-k100	16.62	13.22	**11.57**	15.75
ibm-2004-2_14-k45	**22.73**	29.62	34.82	38.91
ibm-2004-3_02_1-k95	**2.19**	8.47	4.55	12.45
ibm-2004-3_02_3-k95	6.91	8.38	5.52	**4.37**
ibm-2004-3_11-k60	794.18	**724.47**	---	---
ibm-2004-6_02_3-k100	6.80	9.67	**4.90**	5.20
manol-pipe-c10id_s	6.29	**5.76**	7.98	6.04
manol-pipe-c10nidw_s	202.31	**155.20**	214.33	---
manol-pipe-c6nidw_i	752.67	**673.55**	---	---
manol-pipe-c7b	**45.79**	63.51	433.62	267.24
manol-pipe-c7b_i	**55.91**	60.58	369.65	326.76
manol-pipe-c7bidw_i	---	---	---	---
manol-pipe-c7nidw	---	---	---	---
manol-pipe-c9	**6.96**	7.96	14.28	23.83
manol-pipe-c9nidw_s	110.43	95.27	223.71	**93.63**
manol-pipe-f10ni	---	---	---	---
manol-pipe-f6bi	**4.76**	5.70	6.71	9.22
manol-pipe-f7idw	586.18	**351.63**	---	---
manol-pipe-f9b	---	---	---	---
manol-pipe-f9n	---	---	---	---
manol-pipe-g10b	**88.82**	104.70	---	---
manol-pipe-g10bidw	**642.67**	718.07	---	---
manol-pipe-g10id	135.33	**101.71**	---	---
manol-pipe-g10nid	**592.59**	779.63	---	---
manol-pipe-g6bi	**1.84**	1.96	2.83	2.27
manol-pipe-g7nidw	**22.71**	25.62	187.74	---
aloul-chnl11-13	---	---	---	---
een-pico-prop01-75	2.52	**2.05**	2.67	2.80
een-pico-prop05-50	**36.96**	45.97	114.64	171.62
een-tip-sat-nusmv-t5.B	14.96	17.79	13.10	**12.95**
een-tip-sat-nusmv-tt5.B	15.40	13.83	**13.28**	14.89
een-tip-uns-nusmv-t5.B	7.18	11.28	10.14	**7.17**

Table 6. Disabling certain features of PicoSAT on the SAT Race'06 instances cont. from Tab. 5.

	list2	norandom	restart2	norestart
goldb-heqc-alu4mul	800.56	---	775.36	**592.21**
goldb-heqc-dalumul	---	---	---	---
goldb-heqc-desmul	**67.27**	72.01	67.51	80.13
goldb-heqc-frg2mul	**164.30**	183.24	658.05	---
goldb-heqc-i10mul	**753.49**	789.31	---	---
goldb-heqc-i8mul	---	---	---	---
goldb-heqc-term1mul	**213.41**	428.72	337.34	311.29
grieu-vmpc-s05-25	12.38	**1.81**	86.69	121.49
grieu-vmpc-s05-27	48.10	184.25	164.34	**1.52**
grieu-vmpc-s05-28	**329.56**	---	381.22	521.70
grieu-vmpc-s05-34	---	---	---	---
hoons-vbmc-lucky7	8.02	**7.40**	37.20	21.73
maris-s03-gripper11	43.85	**31.39**	150.35	---
narain-vpn-clauses-6	753.68	**457.96**	---	---
schup-l2s-guid-1-k56	**699.22**	749.08	---	---
schup-l2s-motst-2-k315	885.23	871.97	---	**804.53**
simon-s02-w08-18	448.21	**377.50**	760.76	636.65
simon-s02b-dp11u10	---	**572.63**	---	---
simon-s02b-k2f-gr-rcs-w8	---	---	---	---
simon-s02b-r4b1k1.1	**21.11**	---	---	---
simon-s03-fifo8-300	247.99	**233.94**	---	805.22
simon-s03-fifo8-400	**409.51**	495.64	---	---
vange-col-abb313GPIA-9-c	---	---	**400.73**	---
vange-col-inithx.i.1-cn-54	**17.08**	26.76	26.44	676.64
mizh-md5-47-3	---	---	**659.18**	---
mizh-md5-47-4	703.03	**578.73**	---	748.90
mizh-md5-47-5	**747.32**	---	---	---
mizh-md5-48-2	---	---	---	**658.26**
mizh-md5-48-5	---	812.82	830.36	**393.35**
mizh-sha0-35-2	**221.90**	363.07	435.82	---
mizh-sha0-35-3	339.62	**229.42**	404.71	686.43
mizh-sha0-35-4	**187.10**	524.00	665.07	---
mizh-sha0-35-5	**280.16**	---	518.25	---
mizh-sha0-36-2	**388.62**	---	---	---
velev-engi-uns-1.0-4nd	127.01	120.32	99.91	**40.78**
velev-engi-uns-1.0-5c1	7.84	**7.75**	8.80	9.11
velev-fvp-sat-3.0-b18	102.62	70.47	**44.81**	---
velev-live-uns-2.0-ebuf	518.45	**204.90**	669.57	---
velev-npe-1.0-9dlx-b71	510.05	---	**505.82**	---
velev-pipe-o-uns-1.0-7	---	---	---	---
velev-pipe-o-uns-1.1-6	---	---	---	---
velev-pipe-sat-1.0-b10	319.61	**56.45**	---	---
velev-pipe-sat-1.0-b7	218.38	**50.53**	---	---
velev-pipe-sat-1.0-b9	55.59	60.51	198.76	**49.76**
velev-pipe-sat-1.1-b7	**34.41**	58.01	---	---
velev-pipe-uns-1.0-8	---	---	---	---
velev-pipe-uns-1.0-9	---	---	---	---
velev-pipe-uns-1.1-7	---	---	---	---
velev-vliw-sat-2.0-b6	246.47	298.95	**96.94**	---
velev-vliw-sat-4.0-b1	730.80	741.75	**365.25**	520.21
velev-vliw-sat-4.0-b3	112.91	125.96	134.05	**97.16**
velev-vliw-sat-4.0-b4	867.02	455.97	**436.87**	---
velev-vliw-uns-2.0-iq4	---	---	---	---
velev-vliw-uns-4.0-9C1	---	---	---	---

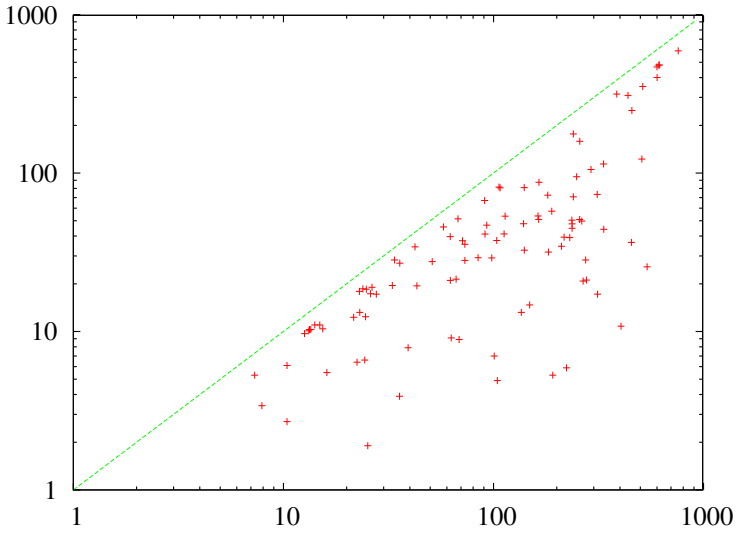

Figure 10. Space usage in MB, if PicoSAT generates an in-memory proof trace, is shown on the horizontal axis (`trace`). The vertical axis lists the space usage without generating and keeping the proof trace, but otherwise having identical data structures for watchers (`list`).

In figure Fig. 10 we show the overhead in memory usage for generating and keeping proof traces in memory. The overhead is on average a factor of 5 with a median of 2.6.

5.2 Deterministic Behavior

In order to measure the impact of low-level optimizations of a SAT solver precisely, it is necessary to enforce deterministic behavior, unless a huge number of experiments can be afforded. More specifically, we instrumented PicoSAT to dump statistics including the number of generated conflicts, decisions and propagations to a log file and checked with a script automatically, that all five versions (`stack`, `list`, `stack2`, `list2`, and `trace`) produce the same numbers.

Making PicoSAT *deterministic* with respect to enabling or disabling low-level optimizations turned out to be rather difficult. On one hand we already implemented our own simple floating point code for handling activity scores before. Using native C floating point numbers, as in MiniSAT, produces different decision trees for different compilers, compiler versions and different levels of optimizations, even on the same machine.

Producing deterministic behavior when switching between stacks and lists was not hard to achieve. Initially, disabling or enabling special treatment of binary clauses, produced quite different search trees. The first necessary adjustment was to base the reduction schedule for garbage collection of learned clauses on the number of large clauses alone and to ignore binary clauses. For instance, delaying reduction by one conflict alone can already change the search tree dramatically. We also had to make sure that during the analysis phase in backtracking the implication graph is traversed in precisely the same order.

Since PicoSAT picks a random decision once in a while, another alternative would have been to repeat the experiments with different seeds for the random number generator. We decided against this alternative, in order to precisely measure the impact of the optimizations by just comparing two runs.

5.3 Simplification: Removing Satisfied Clauses

Originally, we implemented the same algorithm as MiniSAT [11] for disconnecting watched clauses which became satisfied or garbage during a reduction process. This algorithm is a linear search through the whole stack respectively list of clauses for a certain literal and removes individual clauses. It obviously has a quadratic accumulated worst case complexity in the number of clauses to be disconnected.

An improvement would be to use doubly linked lists. However, this only works for our list based implementation, would require two more link fields in the clause header, and would penalize the stack based implementation in an unfair manner in the comparison. The alternative, which we eventually implemented, simply delays disconnecting individual clauses as soon a clause becomes garbage. After all garbage clauses are marked the collection phase is started, which goes through all occurrence lists respectively stacks of all literals only once and removes references to garbage clauses.

Still, as the comparison of the profiles revealed, flushing the references to garbage clauses as in the stack based implementation is much faster than traversing lists in our new implementation. We believe that this effect is due to the fact that in the list based implementation touching the larger headers of the clauses with their two link fields is less cache friendly than just traversing the stack and skipping references to clauses marked garbage. In the latter case only one word of each clause needs to be read while in the former at least three: one containing the garbage flag, at least one literal to determine the correct link field, and the link field.

6. Conclusion

This article describes the SAT solver PicoSAT, including a new restart policy. A detailed experimental analysis of different data structures for occurrence lists is presented. It is shown, that optimizing on this low-level has a large impact on the performance of a SAT solver. The experimental results are more robust compared to previous work since the heuristics do not change. Identical search spaces are traversed in all versions. The usage of real occurrence lists instead of stacks has not been evaluated in the literature before. Another contribution is a technique for compressing proof traces, which allows the SAT solver to keep proof traces in memory.

References

[1] P. Beame, H Kautz, and A Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, **22**, 2004.

[2] A. Biere. The AIGER and-inverter graph (AIG) format. http://fmv.jku.at/aiger.

[3] A. Biere. BooleForce. http://fmv.jku.at/booleforce.

[4] A. Biere. PicoSAT. http://fmv.jku.at/picosat.

[5] A. Biere. TraceCheck. http://fmv.jku.at/tracecheck.

[6] A. Biere. The evolution from Limmat to NanoSAT. Technical Report 444, Dept. of Computer Science, ETH Zürich, 2004.

[7] A. Biere and C. Sinz. Decomposing SAT problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:191–198, 2006.

[8] R. Bryant, D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proc. TACAS'07*.

[9] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Comm. of the ACM*, **5**, 1962.

[10] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT'05*.

[11] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT'03*.

[12] Z. Fu, Y. Mahajan, and S. Malik. New features of the SAT'04 versions of zChaff. In *SAT Competition 2004 – Solver Descriptions*.

[13] A. Van Gelder. Independently checkable proofs from decision procedures: Issues and progress. In *Proc. LPAR'05*.

[14] A. Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Proc. 7th Intl. Symp. on AI and Mathematics*, Ft. Lauderdale, FL, 2002.

[15] A. Van Gelder. Generalizations of watched literals for backtracking search. In *Proc. 7th Intl. Symp. on AI and Mathematics*, Ft. Lauderdale, FL, 2002.

[16] A. Van Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In *Proc. SAT'07*, 2007.

[17] E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust Sat-Solver. In *Proc. DATE'02*.

[18] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proc. DATE'03*.

[19] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. AAAI'98*.

[20] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proc. IJCAI'07*.

[21] R. Jeroslow and J. Wang. Solving propositional satisability problems. *Annals of mathematics and AI*, **1**, 1990.

[22] T. Jussila, C. Sinz, and A. Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *Proc. SAT'06*.

[23] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, **47**, 1993.

[24] M. Lewis, T. Schubert and B. Becker. Multithreaded SAT solving. In *Proc. ASP-DAC'07*.

[25] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers*, **48**(5), 1999.

[26] K. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV'03*, LNCS.

[27] K. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Proc. TACAS'03*.

[28] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC'01*.

[29] A. Nadel. Backtrack search algorithms for propositional logic satisfiability : Review and innovations. Master's thesis, The Hebrew University, Tel Aviv, Israel, 2002.

[30] S. Pilarski and G. Hu. Speeding up SAT for EDA. In *Proc. DATE'02*.

[31] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proc. SAT'07*.

[32] K. Pipatsrisawat and A. Darwiche. RSat 2.0: SAT solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[33] L. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, Burnaby, Canada, 2004.

[34] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proc. ASE'03*.

[35] C. Sinz. SAT-Race'06. http://fmv.jku.at/sat-race-2006.

[36] C. Sinz and A. Biere. Extended resolution proofs for conjoining BDDs. In *Proc. CSR'06*.

[37] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, **17**(1), 2003.

[38] H. Zhang. SATO: An Efficient Propositional Prover. In *Proc. CADE'97*.

[39] L. Zhang and S. Malik. Cache performance of SAT solvers: a case study for efficient implementation of algorithms. In *Proc. SAT'03*.

[40] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proc. DATE'03*.