SAT Solver Descriptions: CMUSAT-Base and CMUSAT

Himanshu Jain
CMU SCS, Pittsburgh, PA 15213

Edmund Clarke
CMU SCS, Pittsburgh, PA 15213

ABSTRACT

CMUSAT-Base is a satisfiability (SAT) solver for formulas expressed in conjunctive normal form (CNF). It uses the DPLL algorithm to decide the satisfiability of CNF formulas. The basic DPLL algorithm is enhanced using various standard techniques such as watch literal scheme for efficient Boolean Constraint Propagation, conflict driven learning, non-chronological backtracking, restarts, conflict clause minimization, and variable activity based decision. The new features of CMUSAT-Base are: 1) an optimization to the watch literal scheme which leads to consistent improvement, 2) simplified data structures based on standard template library (STL), and 3) efficient usage of STL to achieve a high performance SAT solver.

Modern SAT solvers employ pre-processing techniques in order to simplify a given CNF formula before the actual SAT solving starts. CMUSAT solver combines a pre-processing frontend with CMUSAT-Base.

1. WATCH LITERAL SCHEME

Current SAT solvers spend 80%-90% of their runtime during Boolean Constraint propagation (BCP). The aim of this step is to find out all possible implications (variable assignments) due to a given decision by using the *unit literal* rule or report a clause which is falsified due to the current assignment. A major technique to make BCP efficient is the watch literal scheme first proposed by the zChaff [5] SAT solver.

For simplicity let us focus only on clauses with size greater than one in the description below. In the watch literal scheme two literals l_1, l_2 are associated with each clause C, where $l_1 \in C, l_2 \in C$. We refer to literals l_1, l_2 as the *watches* for a clause C. As long as both l_1 and l_2 are unassigned there is no need to examine clause C during BCP. This avoids bringing C from the main memory to the caches which can require numerous clock cycles. By minimizing the number of times a clause is brought in the (L1/L2) cache the watch literal scheme improves the performance of BCP significantly.

Most of the existing SAT solvers examine a clause C (or atleast a part of it) when one of its watches becomes false. Let us assume that C is stored as an array and $C[i]$ denotes the literal at position i, where $0 \leq i < N$ and N is the number of literals in C. Also without loss of generality we assume that watches l_1, l_2 for C are stored in the first two positions, that is, $l_1 = C[0]$ and $l_2 = C[1]$. Now suppose that l_2 becomes false. Then the following cases arise:

1. l_1 is already true and C is satisfied under current assignment.

2. l_1 is not true and we are able to find another literal l_3 in C which is different from l_1, l_2 and is not false. In this case l_3 replaces l_2 as one of the watches for C.

3. l_1 is not assigned and all the other literals in C are false. In this case l_1 is implied by C

4. All literals in C are false. In this case C is a conflict clause.

Observe that the four cases above are total and mutually exclusive (when l_2 is false). We profiled the number of times each of the above cases occur on a large number of industrial benchmarks and found that case 1 occurs most often. Table 1 shows the results on a few industrial benchmarks. The column "#Case 1" counts the number of times case 1 holds during the SAT solving, and the column "#Cases 2,3,4" reports the total number of times other cases hold. It is easy to see that case 1 occurs most often than all the other cases combined.

In order to detect when case 1 holds we only need to look at the watches for a clause and not the entire clause. That is, we can completely eliminate the need of getting C (or a part of it) into the cache when case 1 holds.

Benchmark	#Case 1	#Cases 2,3,4	Result
`goldb-heqc-desmul`	3.8×10^8	1.5×10^8	UNSAT
`cache-ibm-q-full`	6×10^8	1.5×10^8	UNSAT
`aloul-chnl11-13`	1.6×10^9	5.6×10^8	UNSAT
`ibm-2002-11r1-k45`	7.2×10^8	4.1×10^8	SAT
`ibm-2004-04-k100`	6.8×10^8	3.6×10^8	SAT
`manol-pipe-c9nidw-s`	8.2×10^8	4.5×10^8	UNSAT
`velev-vliw-sat-4.0-b1`	1.5×10^9	3×10^8	SAT
`velev-vliw-sat-4.0-b3`	8.9×10^8	1.8×10^8	SAT

Table 1: Frequency of various possibilities when one of the watches for a clause becomes false.

When case 1 holds current state-of-the-art solvers (for example, MiniSat [1]) looks only at $C[0], C[1]$ as the watches are stored as the first two literals in the clause. However, since the first two elements of C are accessed the hardware prefetching mechanism may also bring other elements of C into the cache which are not needed whenever case 1 holds.

In CMUSAT-Base C is not touched (read/written) when the case 1 holds. This is done by *separating* the watches and the clauses as described in the next section.

2. DATA STRUCTURES FOR ENHANCED WATCH LITERAL SCHEME

Each clause is assigned a number called *clause index*. All the clauses are stored in an array called *clauses*, and *clauses*$[i]$ stores the clause with index i. Another array called *warray* stores the watches corresponding to each clause, that is, *warray*$[i]$ stores the two literals that are watches for *clauses*$[i]$.

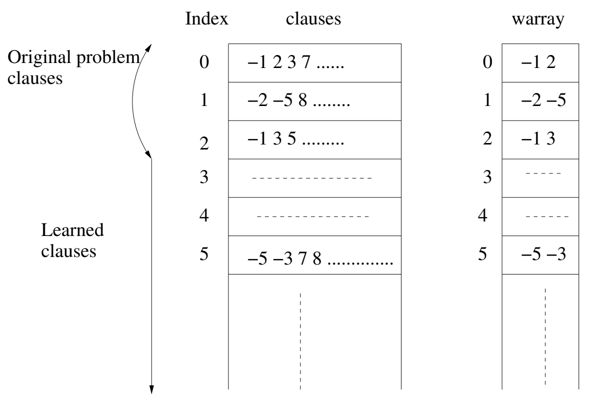

Figure 1: Organization of clauses and watches.

For each literal *l* the *watch-list* for *l* contains a list of clause numbers where *l* is being watched. This is in contrast to many existing SAT solvers such as MiniSat which store the pointers to the actual clauses in the watch lists.

If *l* becomes false the watch list for *l* is scanned. Suppose the watch list contains a clause number *j*. Then in order to check whether the other watch for $clauses[j]$ is satisfied (case 1 holds) we look at $warray[j]$. Note that $clauses[j]$ is examined only when case 1 does not hold.

Example: Figure 1 shows the *clauses* and *warray* data structures. The *clauses* array stores the original problem clauses in the beginning followed by the learned clauses. For each clause the corresponding watches are present in the watches array (*warray*). The watches for clause number 1 are literal -2 and literal -5.

The watch list for literal -1 will contain clause numbers 0, 2. If literal -1 is set to false, then the watch literal scheme examines clause number 0 and clause number 2. In our implementation it is first checked whether the other watch in these clauses is true or not by accessing warray[0] and warray[2]. By examining warray[0] we know that the other watch for clause number 0 is 2. If literal 2 is true, then we know that clause number 0 (*clauses*[0]) is satisfied. There is no need to get *clauses*[0] in the cache.

MiniSat 2.0 also check for case 1 when one of the watches becomes false. However, MiniSat does that by accessing the first two elements of the actual clause. This might cause the hardware or software pre-fetching mechanism to bring other elements of the clause in the cache as well. As Table 1 shows the prefetched elements are not needed more than half of the time.

3. STANDARD TEMPLATE LIBRARY (STL) BASED DATATYPES

In CMUSAT-Base most of the data structures are standard STL containers such as *vectors* (dynamic arrays). Many existing SAT solvers contain their own implementation of containers such as vectors. We argue that this should be avoided whenever possible because the same level of performance can be achieved by using the STL containers.

In our current implementation a literal is defined to have a type *litt* which is simply an integer[1]. A clause has a type `clauset` which is an STL vector containing literals.

```
typedef int litt;
typedef std::vector<litt> clauset;
```

[1]In retrospect `litt` should have been a class.

The clause database has a type `clausest`. It is stored in form of an STL vector of clauses.

```
typedef std::vector<clauset> clausest;
```

The watch list for a literal has a type *watchlistt*. It stores the clause numbers where a literal is being watched in form of an STL vector.

```
typedef std::vector<int> watchlistt;
```

By relying on STL containers the amount of pointer manipulation is reduced in our implementation. This makes it easier to debug the SAT solver and also argue about its correctness. Reducing the amount of pointer manipulation should also improve the results of compiler optimizations.

4. CMUSAT SOLVER

CMUSAT-Base is a pure SAT solver and does not perform any pre-processing on the input formula. In the past SAT competitions pre-processing has turned out to be an important component of the fast SAT solvers. Thus, we added a pre-processing frontend to CMUSAT-Base. We refer to the resulting SAT solver as CMUSAT.

We use MiniSat 2.0 as the pre-processing frontend in CMUSAT. It implements various ideas such as variable elimination, clause subsumption [4].

We experimented with the various ways of integrating MiniSat 2.0 pre-processing with CMUSAT-Base. The simplest possible integration is to first run MiniSat 2.0 with only pre-processing options enabled and write the pre-processed file to the disk. Then check the satisfiability of the pre-processed file using CMUSAT-Base. The main problem with this integration is that it requires the use of a temporary file which might not be possible if there is no disk space left. Also it is much cleaner if a solver does not use a temporary file during a SAT competition.

We report an integration of MiniSat 2.0 pre-processing with CMUSAT-Base which does not use any temporary file. It consists of following steps:

1. Given a CNF formula ϕ. First, perform pre-processing using MiniSat. It is possible that during pre-processing itself the problem is reported to be unsatisfiable. In this case there is nothing else to be done and CMUSAT exits with an unsatisfiable answer. Otherwise, let ϕ' refer to an equi-satisfiable formula left after pre-processing.

2. Copy ϕ' from MiniSat data structures to the data structures of CMUSAT-Base. After the copying is done there is no longer any need of MiniSat data structures. In our implementation, MiniSat related objects go out of scope after ϕ' is obtained. It is very important to not keep pre-processing data structures or even eliminated clauses lying around as it impacts the performance negatively.

3. Check satisfiability of ϕ' using CMUSAT-Base. If ϕ' is unsatisfiable CMUSAT exits with an unsatisfiable answer. Otherwise, an assignment $\mathcal{V}''$ is obtained which satisfies ϕ'. Note that $\mathcal{V}''$ is not a satisfying assignment to ϕ and it needs to be extended to obtain an assignment $\mathcal{V}$ that satisfies ϕ.

4. Once it is known that ϕ' is satisfiable and $\mathcal{V}''$ is obtained ($\mathcal{V}'' \models \phi'$) we no longer require the CMUSAT-Base data structures (they can go out of scope). In order to extend $\mathcal{V}''$,

CMUSAT reads ϕ again. CMUSAT then checks the satisfiability of ϕ using CMUSAT-Base and it uses the truth assignment provided by $\mathcal{V}''$ as an assumption. We know that ϕ must turn out to be satisfiable. However, the satisfiability checking of ϕ (with $\mathcal{V}''$) gives us a satisfying assignment $\mathcal{V}$ to ϕ. By providing $\mathcal{V}''$ as an assumption the satisfiability checking of ϕ is usually very fast.

5. EXPERIMENTAL IMPACT OF ENHANCED WATCH LITERAL SCHEME

Table 2 summarizes the performance of CMUSAT, MiniSat 2.0, and RSAT [2] on 422 industrial benchmarks. These benchmarks are drawn from the industrial category of SAT competition 2005, Satrace 2006, and 54 problems were generated by the UCLID tool [3] [2]. Both CMUSAT and MiniSat 2.0 use pre-processing while the RSAT version we used does not perform pre-processing[3]. The experiments were conducted on a 1526MHz cpu, AMD Athlon processor with 256 KB cache, and 3GB of main memory. There was a timeout of 30 minutes per problem.

In order to quantify the impact of the enhanced watch literal scheme (Section 2) we use two different configurations of CMUSAT: 1) CMUSAT-test : does not use the enhanced watch literal scheme. It uses the watch literal scheme similar to that in MiniSat 2.0. 2) CMUSAT-sub : this is the version submitted to the competition. It uses the enhanced watch literal scheme and was described in the previous section.

Solver	Solved (out of 422)	Total time (in seconds)
MiniSat 2.0	338	212504
RSAT_1_03_LINUX	346	197854
CMUSAT-test	343	193360
CMUSAT-sub	352	181580

Table 2: Summary of results on 422 industrial benchmarks.

It can be seen that the enhancement to the watch literal scheme described in Section 2 does improve the performance of CMUSAT. It can solve 352 problems and disabling it reduces the number of problems solved to 343.

6. REFERENCES

[1] MiniSat, www.cs.chalmers.se/cs/research/formalmethods/minisat/.

[2] RSAT, http://reasoning.cs.ucla.edu/rsat/.

[3] UCLID, http://www.cs.cmu.edu/∼uclid/.

[4] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[5] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, June 2001.

[2]We thank Sanjit Seshia for providing the UCLID benchmarks.

[3]Personal communication with the authors.