Final Project- CS 264A

Elmalaki, Salma 404006510

Millar, Matt 603813149

June 7, 2015

1 Introduction

In this report, we present some of the technical details and algorithms used to implement our SAT library. In particular, we give details of the main three algorithms in our implementation named; (1) Two-literal watch, (2) Non chronological backtracking and Unique Implication point, and (3) Variable State Independent Decaying Sum. We then present the runtime performance over multiple test-benches. We report performance when the developed SAT primitives are used as (1) stand alone SAT solver and (2) part of the C2D compiler.

Finally, we compare the performance of the developed SAT library against the original library that was originally supplied. In the stand alone mode, we notice that in all the test-benches, our library was able to return a correct model assignment for the variables. In the C2D mode, we notice that within the test-benches—which passed the entailment test— our library outperformed the original library in some of the test-benches while recorded slightly worse total execution time in others with an average loss of execution time equal to 2%.

2 Unit resolution using two-literal watch data structure

We used the two literal watch data structure for the unit resolution. Each clause maintains two references on two of its literals. These references have no order and are allowed to move backward and forward through the list of clause's literals. We use the way the unit resolution is done using the *two literal watch* data structure as described in the lecture notes. We added some modification to the way this algorithm works in order to facilitate the detection of the conflict-clause later on when contradiction happens during the unit resolution. These additions are described in the rest of this section and in section 3.

2.1 Data structure

In this part we explain how we maintain the watching literals through the process of unit resolution.

Each clause maintains a 'List' of literals that it contains and a reference to the two literals it currently watches. These two references are updated during the unit resolution. For each literal we keep a 'list' that contains the indices of the clauses that are currently watching over this literal. At each run of the unit resolution we check only the list of the watching clauses of this particular literal that we resolve on.

```
struct clause{
 ....
Lit** literals;
Lit* L1; //first watched literal
Lit* L2; //second watched literal
.....
}
struct literal{
...
unsigned long* list_of_watched_clauses;
unsigned long* list_of_dirty_watched_clauses;
...
}
```

2.2 Improvement on the actual algorithm *"Dirty List of watching clauses"*

Maintaining the *"list of watched clauses"* requires addition of new clause index and removal of new clause index during the process of unit resolution. To avoid this overhead of maintaining the original list, we create a new 'list' called *"list of dirty watched clauses"*. This 'list' keeps track of flags of the clauses. This flags are cleared if the clause is no more watching this clause, while this flag is set if the corresponding clause is currently watching the literal.
Example:
If a clause of index 3 has the following literals {[-1] , [-5], -8} and is watching over -1 and -5 (represented by brackets) then the list of watching literals for literal -1 and literal -5 will have the following format {x, x, 3} which is a set that has the current indices of the clauses that is watching over literal -1 and -5 (x represents the indices of other clauses that are watching over these literals as well) and will have a corresponding flag in the dirty list of watching clauses that has this format {1, 1, 1} which means that the clause at this corresponding flag is valid as a watching clause
Now suppose during the unit resolution process clause 3 now is watching over -5 and -8 so now it has this format { -1, [-5], [-8]} then literal -1 should not have clause 3 in its list of watching clause. Hence, the list of watching clause over literal -1 will still have this format { x, x, 3} but now the dirty list of watching clause will have this format {1, 1, 0 } which means that the corresponding watching clause at the flag = 0 (which is clause 3) is now not a valid watching clause. Hence, it is not considered in the process of checking all the watching clauses if we were to resolve over the literal -1.

2.3 At conflict and at learning a new clause

When a conflict happens during unit resolution, no backtracking is needed for the two literals references which is one of the biggest advantages of the two-literal watch algorithm. However, on adding a new clause, its references to its watched literals are updated; in our algorithm we just initialize the references to watching literals to be over the first two literals, since the order of which literal to watch first will not affect the completeness of the algorithm. The list of watching clauses and the dirty list of watching clauses for each literal contained in the new learnt clause is updated according to the same structure explained above.

2.4 Differentiate between three cases in unit resolution

The API for the unit resolution has the signature:

```
1 BOOLEAN two_literal_watch(SatState* sat_state,
2                 Lit** literals_list,
3                 unsigned long num_elements,
4                 unsigned long type);
```

This API takes as an argument the list of literals *"literals_list"* from which the unit resolution takes the literal on which the unit resolution will perform.

We pass the whole decision list as the "*literal_list*" in this API along side with a *"type" argument*. The type argument is used to differentiate between three cases of unit resolution explained in the Section 2.4.1, 2.4.2, and 2.4.3

To improve the performance of the unit resolution, the differentiation between three different cases on which how the unit resolution will behave is necessary.

2.4.1 Case 1: At deciding a new literal

When deciding a new literal, this literal is put a added to a decision list kept by the current sat state. This list keeps track of pointers to the literals that are decided or implied in the current state.

```
1 struct sat_state{
2  ....
3  Lit** decisions;
4 .....
5 }
```

When running the unit resolution, we only take into consideration this added literal and derive implications accordingly.

2.4.2 Case 2: At adding a new learning clause

When adding a new clause (which is a learnt clause after contradiction), the unit resolution must run again. This time we make use of the fact that the learnt clause will be reduced to a unit clause in the current state leading to an implication of a new literal (which is the free literal in the learnt clause). This new implied literal from the learning clause is added to the decision list which then pass to the API *two_literal_watch* of the sat state and then the unit resolution will run over this new implied literal.

This means that once a learnt clause is added to the new set of clauses, a new implication is driven which is the free literal in the learnt clause. Then unit resolution will drive any chance of successive implications due to this new implied literal.

2.4.3 Case 3: At the presence of a unit clause

At the very first beginning, we scan the set of clauses in the CNF for the presence of a unit clause. This unit clause will force an implication. All these implications that were go from the unit clauses are added to the decision list described above. Unit resolution will then take one literal at a time from this decision list and run unit resolution on it to derive more successive implications.

3 Conflict detection - Clause learning: Non chronological backtracking

This section outlines the conflict analysis procedure used by our SAT solver. We used the algorithm described by [1].

3.1 Learning clauses from conflicts

During running the unit resolution, each time a new implication is determined, the antecedent of the this literal is captured. This antecedent is the clause that caused this implication. Accordingly, the corresponding variable of this literal is instantiated, and we assign this variable this antecedent. We used the definition of antecedent as described by [1] as follows:
The clause used for implied literal l_i is said to be the antecedent of l_i, denoted by $\alpha(l_i)$. If the literal l_i is a decided literal then it has no antecedent. Hence, its antecedent is equals to $NULL$.

When contradiction happens in unit resolution, the conflict clause is captured and the unit resolution procedure returns failure. We denote this conflict clause as $\alpha(\kappa)$

Hence, from this step we come up with two main definitions that we use afterwards to derive the learning clause:

1. : The antecedent of each literal $\alpha(l_i)$
2. : The conflict clause (the clause that causes the contradiction) $\alpha(\kappa)$

We add this antecedent as a property to the variable structure as shown below by capturing the value of the index of this antecedent clause in the variable structure.

```
1 struct var{
2  ...
3 unsigned long antecedent;
4 ....
5 }
```

3.2 Clause learning with UIPs

We used the two definitions described in the above section to learn the a new clause using the UIP method. The algorithm for this procedure can be shown in equation:

$$w_L^{d,i} = \begin{cases} \alpha(\kappa) & \text{if } i = 0 \\ resolution(w_L^{d,i-1}, \alpha(l)) & \text{if } i \neq 0 \wedge \zeta(w_L^{d,i-1}, l, d) = 1 \\ w_L^{d,i-1} & \text{if } i \neq 0 \wedge \sigma(w_L^{d,i-1}, d) = 1 \end{cases}$$

This equation is defined as follows:

1. d represents the current decision level
2. $w_L^{d,i}$ is a learned clause at decision d at step i
3. $\sigma(w, d)$ is the number of literals in $w_L^{d,i}$ assigned at decision d.

4. Define a predicate ζ

$$\zeta(w, l, d) = \begin{cases} 1 & \text{if } l \text{ element of } w \wedge \text{ decision of } (l) = d \wedge \alpha(l) \neq NIL \\ 0 & \text{otherwise} \end{cases}$$

5. A clause resolution (*resolution*) is applied on the two clauses $w_L d, i-1$ which is the learnt clause from the previous step with the antecedent clause of the literal l on which the predicate ζ holds to output an intermediate learnt clause $w_L^{d,i}$ at step i.

This algorithm will terminate at the fixed point when $w_L^{d,i} = w_L^{d,i-1}$ and is guaranteed to give a leaning clause w_L that is contain the first the UIP.

3.3 Improvements on the original algorithm

We improved the algorithm explained above by adding a new check. In the middle of the loop when we choose a literal l on which the predicate ζ holds, we choose l that was last falsified in the clause $w_L^{d,i-1}$ This check makes sure we converge to the first UIP faster.

The API used to run the conflict analysis and return the learnt clause using the UIP method as explained above is:

```
1   Clause* CDCL_non_chronological_backtracking_first_UIP(SatState* sat_state)
```

3.4 Uses of clause learning

After learning a new clause at contradiction as explained above, the clause is added to the set of clauses as a new constraint. Unit resolution runs again but with this new clause added. This new clause will have to update the list of watching clauses and list of dirty watching clauses as explained in Section 2.

4 Variable Order

In order to speed up our SAT solver performance, we make use of the "Variable State Independent Decaying Sum" heuristic, or VSIDS [2] to select literals for decision in a more efficient way. This heuristic is an attempt to represent the frequency with which each possible variable assignment is showing up *recently* in execution, with the idea being to exploit locality patterns while making our decisions, that is, once we start making decision assignments on a variable, if we find some contradiction, we probably want to keep examining rather than go somewhere else. The principle is simple: at the beginning of execution, we assign to each literal L a *score* which represents the number of occurrences of L in the CNF. Upon learning a clause, the scores of all literals contained within that clause are incremented. Then most importantly, once in a while we lower every literal's score uniformly (by dividing by a constant). In our implementation, this step is only ever performed upon clause learning, and whether we perform it is decided randomly with 25% probability of performing the division. We use a division constant of 2.

To evaluate the effectiveness of the heuristic, we run tests on the benchmark CNFs and compare the performance under the default variable ordering to VSIDS ordering. Following is a table comparing the performance under the two operation modes. time values are in seconds (using the C clock approximation of process time), and memory values are in pages (VmPeak measurement via /proc filesystem collected on Debian 64 bit with 4096-byte virtual memory pages):

CNF	t_{norm}	Mem_{norm}	t_{VSIDS}	$\text{Mem}_{\text{VSIDS}}$	Speedup ($t_{\text{norm}}/t_{\text{VSIDS}}$)	$\text{Mem}_{\text{norm}}/\text{Mem}_{\text{VSIDS}}$
tire-2	0.006154	1200	0.008706	1200	0.7069	1.0
qg2-07	0.4099	4692	0.4318	4790	0.9493	0.9795
bw_large.b	0.715	1881	0.09193	1815	7.777	1.036
C211_FS	0.005831	1353	0.008643	1353	0.6747	1.0
qg7-09	0.03181	2079	0.02123	2079	1.498	1.0
C171_FR	0.008263	1386	0.01245	1386	0.6635	1.0
tire-3	0.004159	1201	0.002689	1201	1.547	1.0
prob004-log-a	1.585	2077	0.424	2046	3.738	1.015
C210_FVF	0.01114	1419	0.01854	1419	0.6009	1.0
qg1-07	0.5792	4825	0.2505	4657	2.312	1.036
log-2	71.04	6143	0.7919	2278	89.71	2.697
medium	0.00169	1122	0.001439	1122	1.174	1.0
ais10	2.115	3330	1.564	3935	1.353	0.8463
qg6-09	0.02053	2079	0.04194	2079	0.4894	1.0
bw_large.a	0.009308	1287	0.006063	1287	1.535	1.0
C230_FR	0.008734	1452	0.01476	1452	0.5916	1.0
tire-4	0.009639	1254	0.00551	1254	1.749	1.0
2bitcomp_5	0.00437	1122	0.00131	1089	3.336	1.03
cnt06.shuffled	1.512	2081	0.203	1518	7.447	1.371
log-3	380.3	21004	3.612	3069	105.3	6.844
4blocksb	1.153	2442	5.92	6199	0.1948	0.3939
log-1	0.01989	1287	0.007027	1287	2.831	1.0
C163_FW	0.02035	1485	0.02673	1485	0.7614	1.0
C250_FW	0.005743	1287	0.007758	1287	0.7403	1.0
ssa7552-038	0.01398	1353	0.01648	1320	0.8487	1.025
huge	0.02073	1419	0.00885	1386	2.342	1.024
C638_FKA	0.01324	1620	0.02145	1620	0.6173	1.0
ra	0.01837	1684	0.01965	1651	0.935	1.02
C215_FC	0.0114	1452	0.01984	1452	0.5747	1.0
2bitmax_6	0.0515	1260	0.02566	1194	2.007	1.055
C638_FVK	0.01037	1353	0.02114	1353	0.4905	1.0
C169_FW	0.00277	1287	0.003395	1287	0.8159	1.0

We observe that many of the speedup values are less than 1, which seems troubling because the heuristic is supposed to make the system faster. The first remark here is that the clock time technique that used in our performance measurements is not a perfect count (i.e., these tests were run under bare metal Linux which does not provide real-time guarantees), so we will ignore the values that are close to 1. In particular, we note that most of the values substantially less than 1 (and many of those close to 1) belong to CNFs that were solved in very little time by the original solver, whereas the tougher (slower) problems generally saw a performance gain from the heuristic. We suspect the reason for this discrepancy then is that on these smaller CNFs, the gains provided by the heuristic may not be worth the overhead of computing the actual VSIDS scores. We could probably mitigate this problem substantially with some more precise fine-tuning of the score computation. The one counterexample to our theory comes from the 4blocksb CNF, which of

note has the following interesting property: On most of the runs we have observed our solver make, the contradictions found tend to happen on a set of clauses that is nonuniform but has some spread to it. For example, of the 216 contradictions that occur in solving bw_large.b.cnf, the mode clause for contradictions is clause #4588, upon which occur 25 of the 216 contradictions. In qg1-07.cnf, 215 contradictions occur, and the most contradictory clause is #38, which yields 7 contradictions. However, in 4blocksb.cnf, of the 396 total contradictions, a full 329 of them occur on clause #24548, which makes perfect sense upon looking at the CNF file itself (in VSIDS mode, this behavior reverts back to a more consistent 111 / 1782 contradictions sharing the most-contradictory clause #21370). So we suspect it is possible the performance anomaly for this file may be exacerbated by the CNF's non-uniform properties, that is, if the bulk of the work was going to occur in a small section of the knowledge base in the first place, the process of variable selection becomes less useful. It may also mean that our implementation of the heuristic is not suitable for knowledge bases of this nature.

5 Performance

5.1 SAT solver

Performance of these algorithms are tested over the benchmarks/sampled provided. In order to make sure that we do not have a 'false positive' (in which we detect a SAT wrongly) we make sure that the instantiation of variables at the end of the sat solver is an actual model of the CNF. These results are shown in table 3

5.2 Knowledge compiler

Results with the knowledge compiler is shown in table 4. We run the the model counter with the our library and compare it with the executable c2d sent in the executable folder. We only tested the files that passed in the sat solver as indicated in table 3

5.3 Completeness of our algorithm

We implemented a function called "evaluateDelta()" in which we evaluate the final computed model over the original and learned clauses. As shown in Table 3, our SAT solver is sound and complete in the sense that whenever the algorithm terminates it reveals a correct model as checked by the "evaluateDelta()" function. This function we add in the sat solver so that it tests whether the generated model from the solver is right model or not.

However, when the primitives are integrated with the C2D compiler, some of the benchmarks failed to pass the entailment test. We believe that this behavior is a result of our lack of understanding of how the primitives are used in the C2D compiler and *NOT* as a result of computing incorrect models. Unfortunately, we were not able to debug the interface between the primitives and the C2D compiler.

Test Unsat : To make sure that our algorithm can detect UNSAT as well, we construct the 3-cnf unsat example:

```
1 c  3 UNSAT CNF
2 p cnf 3 8
3 1 2 3 0
```

```
1 2 -3 0
1 -2 3 0
1 -2 -3 0
-1 2 3 0
-1 2 -3 0
-1 -2 3 0
-1 -2 -3 0
```

Our algorithm managed to detect the UNSAT using the both the sat_solver and the C2d when linked with our library. The result from the C2d is shown below.

```
Constructing CNF... DONE
CNF stats:
  Vars=3 / Clauses=8
  CNF Time  0.000s
Constructing vtree (from primal graph)... DONE
Vtree stats:
  Vtree widths: con<=1, c_con=8 v_con=1
  Vtree Time   0.001s
Compiling... DONE
  NNF memory          0.2 KB
  Learned clauses        3
Cache stats:
  hit rate      nan%
  lookups       0
  ent count     0
  ent memory    0.0 KB
  ht  memory    152.6 MB
  clists        nan ave, 0 max
  keys          nanb ave, 0.0b max, 10000000.0b min
  Compile Time 0.000s
Saving compiled NNF to file... DONE
  Save Time           0.000s
NNF stats:
  Nodes               1
  Edges               0
Post compilation
  Loading NNF from file... DONE
  Load Time           0.000s
NNF stats:
  Nodes               1
  Edges               0
  Counting... 0 models / 0.000s
  Checking decomposability... OK / 0.000s
  Checking entailment... OK / 0.000s
Total Time: 0.090s
```

As shown in the results of the C2d on the UNSAT example, it correctly determines 0 model which says that this is an UNSAT cnf.

5.4 Remarks on the results

We compare our results with the benchmarks that passed in the C2D test with our library versus the executable one to get the sense of how well how library is. This compared result is shown in Table 2.

sample cnf	compilation time executable/our library	node count (NNF) executable/our library	edge count(NNF) executable/our library	total time executable/our library
bw_large.a	0.007s/0.032s	917/917	916/916	1.381s/1.521s
bw_large.b	7.091s/10.307	2758/2783	3304/3354	18.811s/23.761s
huge	0.005s/0.021s	917/917	916/916	1.885s/1.910s
log-3	5.066s/26.407s	404274/408654	804312/813072	191.964s/248.255s
medium	0.001s/0.002s	315/319	360/368	0.163/0.154s
qg1-07	1.365s/5.822s	3043/3043	5092/5092	238.880s/242.428s
qg2-07	1.504s/7.168s	4976/4976	8918/8918	269.797s/271.037s
qg3-08	0.878s/6.867s	8449/8673	15384/15832	12.175s/17.74s
qg6-09	0.055s/0.794s	3739/3745	5572/5584	60.918s/60.461s
ra	12.950s/12.516	479722/505985	954500/1007026	100.728s/104.864

Table 2: comparison between our library and the given executable for the passed tests

We note that out of these 10 test-benches, our library outperform in some of them while perform slightly worse in the rest with a degradation of 2% in the total execution time.

A Literal data structure

```
struct literal {
   c2dLiteral sindex;
      /* Literal index from 1 to  n for positive literals
       *                   -1 to -n for negative literals  */

   BOOLEAN LitState;
      /* A flag that is raised if this literal has a value or not */

   BOOLEAN LitValue;
      /* Whether this literal has value true, false, or free (not set) */

   unsigned long decision_level;
      /* If it is decided or implied,
       * What is the level of the literal in this setting? */

   Var* variable;
      /* Pointer to the variable that this literal represents */

   /* The following variables are used for the two literal watch
    * data structure */
   unsigned long * list_of_watched_clauses; // lists clause *indices*
   unsigned long num_watched_clauses;
   unsigned long max_size_list_watched_clauses;

   /* The following varibales are used for the dirty flags of watching
    * clause to avoid removing from the original list */
   unsigned long * list_of_dirty_watched_clauses;
   unsigned long num_dirty_watched_clauses;
   unsigned long max_size_list_dirty_watched_clauses;

   /* The following variables help facilitate setting the subsumed
    * clause add a list of clauses containing this literal */
   unsigned long * list_of_containing_clauses;
   unsigned long num_containing_clause;
   unsigned long max_size_list_contatining_clauses;

   /* For use by the variable order algorithm (VSIDS) */
   unsigned long vsids_score;
}
```

B Variable data structure

```
struct var {
   c2dSize index;
      /* The variable index */

   /* The following pointers represent the positive and negative
    * literals corresponding to this variable, respectively */
   Lit * posLit;
   Lit * negLit;

   /* The following variables are used to store a list of which
    * clauses contain this variable inside of them */
   unsigned long * list_clause_of_variables;
```

```
13     unsigned long num_of_clauses_of_variables;
14     unsigned long max_size_list_of_clause_of_variables;
15
16     /* This list never changes. It is set at the beginning and
17      * remains as is throughout program execution.  */
18     unsigned long * list_clause_of_variables_in_cnf;
19     unsigned long num_of_clauses_of_variables_in_cnf;
20     unsigned long max_size_list_of_clause_of_variables_in_cnf;
21
22     unsigned long antecedent;
23        /* This antecedent is used for the non-chronological
24         * backtracking UIP.  The unit clause used for implying
25         * the (variable) is said to be the antecedent of this
26         * literal(variable). */
27     // Clause * antecedent; // (original version, was changed to above)
28
29     SatState* sat_state;
30        /* Have to keep a pointer of the sat state here
31         * due to the requirements of some APIs. */
32
33     BOOLEAN mark; // THIS FIELD MUST STAY AS IS
34 }
```

C Clause data structure

```
1  struct clause
2  {
3     c2dSize cindex;
4        /* The clause index */
5
6     Lit ** literals;
7        /* Array of literals that this clause contains */
8
9     unsigned long num_literals_in_clause;
10       /* Needed for bounding the array above */
11
12    unsigned long max_size_list_literals;
13       /* Needed for memory-managing the array above */
14
15    BOOLEAN is_subsumed;
16       /* Used to mark this clause as subsumed */
17
18    /** The following pointers are used for the two-literal-watch
19     * data structure. They represent the two literals that the
20     * clause is watching. */
21    Lit * L1;
22    Lit * L2;
23
24    BOOLEAN mark; //THIS FIELD MUST STAY AS IS
25 }
```

D Sat state data structure

```
1  struct sat_state_t {
2     /* The following varibales represent the primary contents of the
```

```
 3      * SAT state as defined in the project spec */
 4     Clause * delta;
 5     unsigned long * gamma; // use indices to avoid double-copies
 6     Lit ** decisions;
 7     Lit ** implications;
 8     Clause * alpha;
 9
10     Var * variables;
11        /* This array stores all variables in the entire problem space
12         * (and, within them, pointers to their  positive and negative
13         * literals--see variable definition).  It is allocated in the
14         * beginning when the file is parsed. */
15
16     /* The following are used to bound and manage the arrays above. */
17     unsigned long  num_clauses_in_delta;
18        /* This will have a value of m + added any new clauses */
19     unsigned long  num_clauses_in_gamma; // This is not used
20     unsigned long  num_literals_in_decision;
21     unsigned long  num_literals_in_implications;
22     unsigned long  num_variables_in_cnf;
23        /* This is the value n, from the project spec */
24     unsigned long  num_clauses_in_cnf;
25     unsigned long max_size_list_gamma;
26     unsigned long max_size_list_delta;
27
28     unsigned long current_decision_level;
29        /* This represents the current decision level as the algorithm
30         * runs through the variables */
31
32     Clause * conflict_clause;
33        /* If a contradiction happens at the current state, then this
34         * clause points to the cause of contradiction */
35 }
```

E Correctness and Performance Results

E.1 SAT solver mode

sample cnf	sat/unsat	Correct model(not a false positive)
2bitcomp_5	sat	✓
2bitmax_6	sat	✓
4blocksb	sat	✓
C163_FW	sat	✓
C169_FW	sat	✓
C171_FR	sat	✓
C210_FVF	sat	✓
C211_FS	sat	✓
C215_FC	sat	✓
C230_FR	sat	✓
C250_FW	sat	✓
C638_FKA	sat	✓
C638_FVK	sat	✓

ais10	sat	✓
bw_large.a	sat	✓
bw_large.b	sat	✓
cnt06.shuffled	sat	✓
huge	sat	✓
log_1	sat	✓
log_2	sat	✓
log_3	sat	✓
medium	sat	✓
par16_1_c	—	CNF file is not in the right format (end of clause is a new line with 0)
par16_2_c	—	CNF file is not in the right format (end of clause is a new line with 0)
par16_2	—	CNF file is not in the right format (end of clause is a new line with 0)
par16_3	—	CNF file is not in the right format (end of clause is a new line with 0)
par16_5_c	—	CNF file is not in the right format (end of clause is a new line with 0)
par16_5	—	CNF file is not in the right format (end of clause is a new line with 0)
prob004_log_a	sat	✓
qg1_07	sat	✓
qg2_07	sat	✓
qg3_08	sat	✓
qg6_09	sat	✓
qg7_09	sat	✓
ra	sat	✓
ssa7552_038	sat	✓
tire_2	sat	✓
tire_3	sat	✓
tire_4	sat	✓
uf250_017	—	CNF file is not in the right format(extra tab on problem line and spaces and newline)
uf250_026	—	CNF file is not in the right format(extra tab on problem line and spaces and newline)

Table 3: SAT solver results with the new libsat.a library

E.2 C2D compiler mode

sample cnf	compilation time	model count	entailment	node count (NNF)	edge count(NNF)	total time
2bitcomp_5	0.635s	2455928871845 8880 models / 0.004s	Failed!!! / 0.006s	7499	14656	0.751s

2bitmax_6	541.274s	2210801437065 0477719157997 5680 models / 2.605s	Failed!!! / 4.320s	3350822	6700876	553.482s
4blocksb	117.529s	4 models / 0.001s	OK / 0.602s	1870	2808	157.332s
C163_FW	(compilation failed)	—	—	—	—	—
C169_FW	0.004s	3823586928230 400 models / 0.001s	Failed!!! / 0.023s	3051	3268	0.176s
C171_FR	10.500s	12687459941187 75941764618924 08485351426344 68755319382519 29909408939159 48171532851596 09010785934898 10436953575532 78976 models / 0.304s	Failed!!! / 0.742s	317154	630632	12.956s
C210_FVF	(compilation failed)	—	—	—	—	—
C211_FS	0.358s	17335339652871 20330713889420 81332269563352 2344078205198 4369443143680 models / 0.037s	Failed!!! / 0.067s	40173	77036	1.339s
C215_FC	(compilation failed)	—	—	—	—	—
C230_FR	(compilation failed)	—	—	—	—	—
C250_FW	0.010s	1308275436174 1127104423786 071843667968 models / 0.002s	Failed!!! / 0.010s	3675	4388	0.230s
C638_FKA	(compilation failed)	—	—	—	—	—
sample cnf	compilation time	model count	entailment	node count (NNF)	edge count(NNF)	total time

C638_FVK	0.107s	23825888396195 55127000273532 89471299482402 24492211374564 58455631460347 4816234032507 6904921201169 4598603012284 6683922432000 models / 0.006s	Failed!!! / 0.071s	9609	15562	0.660s
ais10	(compilation failed)	—	—	—	—	—
bw_large.a	0.032s	1 models / 0.001s	OK / 0.054s	917	916	1.521s
bw_large.b	10.307s	2 models / 0.002s	OK / 0.534s	2783	3354	23.761s
cnt06.shuffled	(compilation failed)	—	—	—	—	—
huge	0.021s	1 models / 0.000s	OK / 0.080s	917	916	1.910s
log_1	0.046s	74788034318466 3281250 models / 0.003s	Failed!!! / 0.143s	5813	9214	0.513s
log_2	(compilation failed)	—	—	—	—	—
log_3	26.407s	279857462060 models / 0.336s	OK / 148.464s	408654	813072	248.255s
medium	0.002s	2 models / 0.001s	OK / 0.006s	319	368	0.154s
par16_1_c	(CNF not in the correct format)					
par16_2_c	(CNF not in the correct format)					
par16_2	(CNF not in the correct format)					
par16_3	(CNF not in the correct format)					
par16_5_c	(CNF not in the correct format)					
par16_5	(CNF not in the correct format)					
prob004_log_a	(compilation failed)	—	—	—	—	—
qg1_07	5.822s	8 models / 0.002s	OK / 2.862s	3043	5092	242.428s

qg2-07	7.168s	14 models / 0.003s	OK / 4.383s	4976	8918	271.307s
qg3-08	6.867s	18 models / 0.006s	OK / 1.183s	8673	15832	17.064s
qg6-09	0.794s	4 models / 0.002s	OK / 1.161s	3745	5584	60.461s
qg7-09	0.237s	4 models / 0.002s	OK / 1.161s	3726	5546	58.646s
ra	12.516s	18739277038847 93988675401992 03581234243084 69030992781557 96690998321191 09631577636787 26120154469030 85680773058797 18599103790694 62105489708001 87300472379863 33423405217995 60185957916958 40186920710944 33558591235611 56747098129524 43337159646142 48560042278542 41384374972430 82509507328295 0873641 models / 0.507s	OK / 85.102s	505985	1007026	104.864s
ssa7552-038	2.548s	36473748648931 80498147006823 3927315685376 models / 0.029s	Failed!!! / 2.232s	44464	83396	5.175s
tire-2	0.204s	738969640920 models / 0.022s	Failed!!! / 0.318s	29646	57846	0.836s
tire-3	0.384s	172553592456 models / 0.021s	Failed!!! / 0.557s	29277	56994	1.239s
tire-4	3.519s	119042785240500 models / 0.036s	Failed!!! / 0.929s	47185	92212	3.519s
uf250-017	(CNF not in the correct format)					
uf250-026	(CNF not in the correct format)					

Table 4: C2D compiler results with the new libsat.a library

References

[1] J. P. Marques Silva and K. A. Sakallah *Dynamic Search-Space Pruning Techniques in Path Sensitization*, 31st Conference on Design Automation, pp. 705–711, June 1994.

[2] *Chaff: engineering an efficient SAT solver.*, "http://dl.acm.org/citation.cfm?id=379017"