On-The-Fly Resolve Trace Minimization

Ohad Shacham
IBM Haifa Research Laboratory
ohads@il.ibm.com

Karen Yorav
IBM Haifa Research Laboratory
yorav@il.ibm.com

ABSTRACT

The ability of modern SAT solvers to produce proofs of unsatisfiability for Boolean formulas has become a powerful tool for EDA applications. Proofs are generated from a *resolve trace* that captures information about the creation of all conflict clauses. Due to their sizes, resolve traces are kept in files. The sizes of these files makes the use of proofs of unsatisfiability impractical for industrial tools. Although only a small part of the resolve trace is eventually used, until now it was not known how to filter out unnecessary information.

We propose a simple algorithm for on-the-fly resolve trace minimization in which we identify clauses that are guaranteed not to take part in the proof of unsatisfiability, and delete all of their associated information. This algorithm dramatically decreases the size of the resolve trace, to the point where it can be stored in the main memory. Our experiments reveal that the minimized trace is typically 3 to 6 times smaller. This makes the use of proofs of unsatisfiability and the computation of unsat cores more practical and will enable future applications to take advantage of it.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*

General Terms

Verification

Keywords

SAT-based verification, SAT solvers, unsatisfiable core

1. INTRODUCTION

Profound advancements in SAT solving technology have led to a proliferation of SAT-based EDA applications. Many of these applications leverage the ability of modern SAT solvers to produce an *unsat core*. Given an unsatisfiable formula in conjunctive normal form, the unsat core is a subset of the original formula that is unsatisfiable in itself. An analysis of the unsat core enables formal verification tools to over-approximate reachability steps [9], to compute better abstractions [10], to boost the performance of subsequent SAT calls [13], and more. Unsat cores are also used in Artificial Intelligence, FPGA routing, and many additional applications that utilize its efficiency in some way. The unsat core is derived from a *proof of unsatisfiability*. In this paper, we deal with the price of generating proofs of unsatisfiability.

In general, SAT-based verification tools use DPLL-style SAT solvers [7, 11] based on a branch and backtrack algorithm with various improvements. To be able to compute the proof of unsatisfiability, the SAT solver must generate information – during its run – recording all resolution operations that generate conflict clauses. This information is usually referred to as a *resolve trace*. From this information, the proof of unsatisfiability (and the unsat core) can be generated in linear time [16]. During an average run, a modern SAT solver may generate hundreds of thousands of conflict clauses, and resolve trace information is produced for each one. This amounts to hundreds of megabytes of data in average runs. Therefore, in most cases the resolve trace is written to a file. This has two major disadvantages: disk space consumption, and a degradation of run time performance due to expensive I/O operations.

Industrial strength tools are expected to be able to handle large designs, which implies extremely large resolve trace files, running over 2GB in some cases. Although disk space is cheap, the large files pose a practical problem when programs fail due to exceeded disk quotas, or when the files go over the file system limitation. Some of these tools are also able to exploit multiple-CPU architectures and computer farms to run many tasks in parallel, generating many trace files at the same time and in the same disk partition. The problem is further aggravated when incremental SAT is used. In incremental SAT solvers [12, 14], information is shared between consecutive SAT executions to boost performance. Sharing is achieved by reusing conflict clauses. Integrating incremental SAT with algorithms that require proofs of unsatisfiability [9, 13] is problematic since a large part of the resolve traces of all the previous runs needs to be maintained, making the resulting resolve traces even larger. In many algorithms, the use of incremental SAT was abandoned precisely because of this. These problems pose a limitation on the adoption of algorithms based on the analysis of proofs of unsatisfiability in commercial tools.

The proof of unsatisfiability typically involves only a fraction of the conflict clauses that are recorded in the resolve trace. The prevailing point of view is that the size of the re-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2007, June 4–8, 2007, San Diego, California, USA.
Copyright 2007 ACM 6978-1-59593-627-1/07/0006 ...$5.00.

solve trace cannot be reduced during the run, because when a clause is generated it is impossible to predict whether it will take part in the proof of unsatisfiability. Therefore, all information regarding resolutions is maintained.

Our work provides a simple and very efficient technique for minimizing the resolve trace on-the-fly (during the SAT solver's run) while guaranteeing that enough information is available for generating a proof of unsatisfiability. The technique uses a variation of a reference counting scheme that tracks potential uses of a clause. The counting scheme enables us to detect clauses that are guaranteed not to be needed for the proof, enabling the deletion of their corresponding resolve trace information. Each such deletion may cause a cascade of deletions of related clauses, thus removing large amounts of dispensable information. Reference counting is, of course, not new. Our contribution is in showing how to properly use reference counting so that needed information is maintained, in the context of unsat core computation.

Our technique filters out clauses that are guaranteed as not needed for *any* future proof of unsatisfiability. We leave intact the resolve source information of exactly those clauses that can potentially be involved in some future proof of unsatisfiability. This allows us to integrate incremental SAT into SAT-based algorithms that require proofs of unsatisfiability, without having to add extra computations or disk space. A significant boost in performance is observed for these applications, making incremental SAT practical.

We present experimental results that show how our technique significantly decreases the size of the resolve trace, enabling us to store it in the main memory. Our method becomes more effective as the trace sizes grow. For most non-trivial examples, the reduction ratio (non-reduced divided by the reduced size) is above 3, and goes over 16 on the largest examples. The increase in the memory consumption of the program itself, caused by storing the whole (minimized) resolve trace at all times, is not significantly more than the eventual size of the resolve trace. We also found that integrating our technique in the SAT solver slightly reduces the solver's runtime because the small overhead of maintaining the reference count is more than compensated for by avoiding costly I/O operations. Finally, we note that being able to integrate incremental SAT into an algorithm can reduce its run time by more than a half.

The rest of this paper is organized as follows: Section 2 explains the basic DPLL SAT solving algorithm with conflict clause learning. Section 3 explains the generation of unsat cores and the required resolve trace. Section 4 describes our algorithm and proves its correctness. Section 5 presents our experimental results and Section 6 concludes.

2. SAT ALGORITHM

Given a Boolean formula F over a set of variables V, the task of the SAT solver is to find a satisfying valuation to all variables in V. The formula F is given in *Conjunctive Normal Form* (CNF), which is a conjunction of *clauses*, where each clause is a disjunction of *literals*. Given a variable $v \in V$, its corresponding literals are v and its negation $\overline{v}$. For a literal l, let $var(l)$ denote the corresponding variable. For simplification, we refer to clauses as sets of literals.

We briefly describe DPLL [6] with conflict clause learning and non-chronological backtracking [3, 8]. For a more thorough discussion, see [15]. Figure 1 gives a bird's eye view of a DPLL algorithm with learning. The algorithm searches for a satisfying assignment by iteratively choosing a value for a variable using the `decide_next_branch()` function. If all variables are assigned, the algorithm halts and outputs a satisfying assignment. Otherwise, the implications of each assignment are carried through by the `deduce()` function. If `deduce()` reveals a conflict, the reason for the conflict is analyzed and a conflict clause is added to the database. The conflict clause summarizes the combination of values that led to the conflict and prevents this combination from being assigned a second time. The `analyze_conflicts()` function returns a decision level to which the algorithm backtracks. If this level is -1, it means that a conflict exists even without a single decision, in which case the formula is unsatisfiable. Otherwise, the algorithm backtracks to level `blevel` and continues the search.

```
while(1) {
  if (decide_next_branch())
    while (deduce() == CONFLICT) {
      blevel = analyze_conflicts();
      if (blevel < 0) return UNSAT;
      else bactrack(blevel);
    }
  else
    return SAT;
}
```

Figure 1: Basic DPLL algorithm with learning

2.1 Decisions

During the run of a solver, a literal may have one of three values: *true*, *false*, or *undefined*. The `decide_next_branch()` function chooses a variable that is undefined and assigns it with either *true* or *false*. The choice of the variable to decide upon is controlled by a special purpose heuristic. Each decision is associated with a number, called the *decision level*. All the implications that result from one decision are associated with the same decision level. When constant values are revealed, i.e., literals that must be true regardless of previous decisions, they are associated with decision level zero. We refer to these as *level zero constants*.

2.2 Boolean Constraint Propagation

A clause in which one literal is undefined and all the rest are *false* is called a *unit clause*. Such a clause forces an assignment of *true* to the undefined literal, as this is the only way to satisfy the formula. Boolean Constraint Propagation (BCP) is the process of propagating the effect of an assignment. This is the task of the `deduce()` function in Figure 1. Each unit clause implies a new assignment, which may, in turn, result in more unit clauses. During BCP, this process iterates until either no further assignments can be implied, or a conflict is encountered.

For each assignment, the clause that forced this value is called the *antecedent*. The SAT solver must keep antecedent information because it is needed for conflict analysis. When backtracking occurs, assigned values are removed, along with antecedent information. However, values that are added to decision level zero are never removed, and their antecedent clause information is retained throughout the run.

2.3 Clause Learning

A conflict happens when BCP discovers a clause with all

its literals set to *false*. We call this clause a *conflicting clause*. During conflict analysis, the chain of implications that were involved in the conflict is analyzed, and the reason for the conflict is summarized in a *conflict clause*. This clause describes a combination of assignments that should not be repeated as they conflict. The conflict clause is added to the clause database, thus pruning the search space that remains to be traversed. A conflict clause is generated by a series of *resolution* steps, involving many clauses [16].

Definition 1. Let $c = \{l_1, \ldots, l_n\}$ and $c' = \{l'_1, \ldots, l'_m\}$ be clauses and $i \in 1 \ldots n$ and $j \in 1 \ldots m$ indices such that $l_i = \neg l'_j$. Then

$$resolve(c, c', var(l_i)) = \{l_p | p \neq i\} \cup \{l'_p | p \neq j\}$$

For example, for clauses $c_1 = (x_1 \vee x_2)$ and $c_2 = (\neg x_2 \vee x_3)$ the resulting clause is $c_3 = (x_1 \vee x_3)$. Resolution is a powerful operation since it guarantees that the resulting clause is implied by the generating clauses.

Figure 2 displays the pseudocode for `analyze_conflicts()` (from Figure 1). This function is called when a conflicting clause is encountered. The algorithm starts by checking if the current decision level is 0 and if this is the case returns (-1), declaring that the given formula is unsatisfiable due to the fact that a conflict occurs without making any decisions. Otherwise, `cl` is initialized by the conflicting clause and an iterative loop for building a conflict clause starts. The `choose_literal(cl)` function chooses a literal from `cl` such that its corresponding variable was assigned last amongst all the corresponding variables of `cl`'s literals. The `antecedent(var)` function returns the antecedent of this value (the clause that became unit and thus forced this value assignment). The resolution between `var`'s antecedent and `cl` is computed by `resolve(cl, ante, var)`. This process continues until the halting criterion is met. Finally, the generated conflict clause is stored in the clause database and the decision level to which the algorithm must backtrack is returned.

While generating resolutions, `analyze_conflicts()` uses a queue to store the ids of all clauses that were involved (lines 4 and 10). These clauses are called the *resolve sources*, because from this list the conflict clause can be generated by a series of resolutions. If a clause c' appears in the resolve sources list of a clause c we say that c' is a *source* of c, and that c is a *descendant* of c'. The list of resolve sources is dumped to a file (line 13). This file, called the *resolve trace*, is not required for the SAT algorithm itself, but is used later to compute the proof of unsatisfiability. The resolve trace file includes the list of resolve sources for every clause generated during the run, along with the list of constant values in decision level zero and their antecedents.

2.4 Clause Deletion

As mentioned earlier, conflict clause learning causes an accumulation of a huge number of clauses. The role of these clauses is to make the search more efficient by pruning the search space that remains to be explored. However, these clauses also increase the overhead of doing BCP and, therefore, every once in a while, a clause deletion mechanism is employed. This mechanism heuristically chooses which clauses to remove. While the clauses themselves are removed from the database, their ID's and resolve sources lists are maintained in the resolve trace file.

```
analyze_conflicts() {
1  if (decision_level == 0)
2      return -1;
3  cl = find_conflicting_clause();
4  resolve_sources.enqueue(cl.id);
5  while (!stop_criterion_met()) {
6      lit = choose_literal(cl);
7      var = var(lit);
8      ante = antecedent (var);
9      cl = resolve(cl, ante, var);
10     resolve_sources.enqueue(ante.id);
11 }
12 id = add_clause_to_database(cl);
13 print_resolve_sources_to_resolve_trace(id, resolve_sources);
14 back_dl = clause_asserting_level(cl);
15 return back_dl;
}
```

Figure 2: Conflict clause generation algorithm

3. UNSAT CORE

Given an unsatisfiable CNF formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge .. \wedge \varphi_n$, an *unsat core* of φ is a set of clauses ϕ s.t. $\phi \subseteq \{\varphi_i | i = 1..n\}$ and the conjunction of the clauses of ϕ is unsatisfiable. There can be many unsat cores for a given formula, and finding the minimum is too expensive for most purposes. However, a small core is a summary of the cause for the unsatisfiability of the formula, which makes it very useful for many applications.

The unsat core is generated from the resolve trace in linear time, for example, using the algorithm of Zhang and Malik [16]. The core idea is to create a proof of unsatisfiability in the form of a *resolution graph*, building on information from the resolve trace. Our minimization algorithm removes large amounts of data from the resolve trace. To show that our algorithm is correct, we need to show that no necessary information is removed, which requires some understanding of how the unsat core is created. Due to space limitations, we cannot give full details of the unsat core creation algorithm. We take advantage of the fact that the unsat core creation algorithm only requires the resolve trace information for *clauses that actually take part in the proof of unsatisfiability*. Hence, it is sufficient for us to describe the proof of unsatisfiability and omit the details of the algorithm that creates it.

A resolution graph `RG` is a directed acyclic graph representing a series of resolutions. Each node in the graph represents a clause; the leaves represent the original clauses of the formula and the inner nodes represent conflict clauses. For every two clauses c_1 and c_2 s.t. c_1 is one of c_2's resolve sources (i.e., c_2 is a descendant of c_1), there exists an edge in `RG` from c_1 to c_2. Thus, the set of resolve sources of a conflict clause c is represented by the set of incoming edges into c's node.

Since each resolution creates a clause that is implied from the parent clauses, a resolution graph that generates an empty clause shows that the set of clauses involved imply a conflict and hence are unsatisfiable. The transitive fan-in of the empty clause in `RG` forms a proof of unsatisfiability, and the leaves in this graph form the unsat core. Note that the transitive fan-in of the empty clause in `RG` is typically a small fraction of the entire graph. When constructed from the resolve trace produced by a SAT solver, the proof of un-

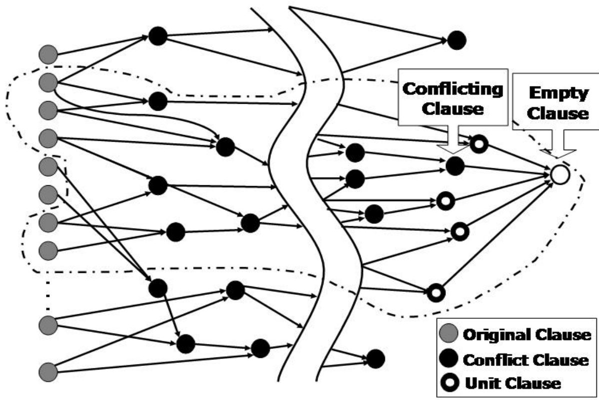


Figure 3: A resolution graph

satisfiability has a distinct structure:

1. The empty clause is generated from resolving the final conflicting clause $(l_1, \ldots, l_n)$ with unit clauses $(\neg l_1)$, $\ldots$, $(\neg l_n)$. The final conflicting clause is the conflicting clause corresponding to the conflict at decision level zero (which is what terminates the SAT solver's run on an unsatisfiable instance). Each unit clause (l_i) is a representation of a level zero constant l_i.

2. A unit clause (l_i) of a level zero constant is generated from its antecedent. If the antecedent is a unit clause, then this clause is used. Otherwise, the required unit clause is generated by resolution of the antecedent with other unit clauses of level zero constants.

3. All conflict clauses, including antecedents and the final conflicting clause, are generated by a resolution of the clauses that appear in their resolve sources lists.

Figure 3 shows a general resolution graph in which the empty clause is one of the roots. The transitive fan-in of this node, encircled with a dotted line, forms the proof of unsatisfiability. The leaves (the gray circles that fall inside the dotted line) form the unsat core.

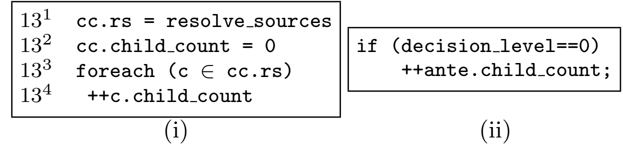
4. RESOLVE TRACE MINIMIZATION

As mentioned before, the resolve trace file is not minimized on-the-fly because while generating conflict clauses it is not possible to predict which clauses will eventually take part in the proof of unsatisfiability. Even when clauses are deleted, it does not mean that they will not take part in the proof, so all resolve sources are kept, both for live and deleted conflict clauses.

Our work provides a technique for deleting resolve sources lists on-the-fly (during the SAT solver's run) while guaranteeing that any information that is deleted is in fact useless. We never delete resolve source information for antecedents of level zero constants. Our technique is based on the following observation regarding clauses that are *not* antecedents of level zero constants:

If there is a deleted clause for which all the descendants are not going to take part in the proof, then the clause itself will not take part in the proof.

This is because on the one hand, this clause cannot be in the transitive fan-in of the empty clause through any of its current descendants; on the other hand, it has been deleted so



```
13^1  cc.rs = resolve_sources
13^2  cc.child_count = 0
13^3  foreach (c ∈ cc.rs)
13^4   ++c.child_count
```

(i)

```
if (decision_level==0)
    ++ante.child_count;
```

(ii)

Figure 4: (i) Updating the `rs` and `child_count` fields for new clauses (ii) Updating the `child_count` for antecedents of constant values

it will never have new descendants. This observation allows us to build a variation of a reference counting algorithm that keeps track of whether a clause has a chance of ever appearing in the proof. We first present our algorithm, and then show that it is correct.

To efficiently minimize the resolve trace on-the-fly, we keep resolve source information inside the solver. For each conflict clause `cc`, we keep two fields: (1) the list of resolve sources of the clause (`cc.rs`), and (2) a counter that tracks the number of descendants of *cc* that still have a chance of appearing in the proof (`cc.child_count`).

We make a few small changes to the SAT-solving algorithm, and add a new function. Figure 4(i) gives a code fragment that replaces line 13 in `analyze_conflicts()` from Figure 2. Here, we store the resolve trace information in the program's main memory. Figure 4(ii) gives a code fragment that is added to the `deduce()` function. This code is executed for each implication, and `ante` is the clause that generated the implication, i.e., the antecedent of the newly found constant. This part guarantees that we never delete the resolve sources of clauses that are antecedents of level zero constants. We also add a call to a new function, called `DeleteResolveSources`, presented in Figure 5. This function is called for every clause that is deleted from the SAT solver's database.

Our algorithm (Figure 5) works as follows: at any point in time `cc.child_count` holds the number of descendants of `cc` that still have a chance to make it into the proof. When a clause is created this count is zero. A newly generated clause can potentially take part in the proof, so the `child_count` of each of its resolve sources is incremented. When a clause `cc` is deleted and `cc.child_count` $\geq$ `1`, we keep `cc.rs` because we cannot know if a descendant of `cc` will take part in the proof or not. If `cc` has no descendants then `cc.rs` is deleted and the `child_count` for each of its resolve sources is decremented. These counters may become zero, so a recursive call to `DeleteResolveSources` tries to delete each of the resolve sources. It is plausible that upon deletion, the `child_count` of `cc` is not zero, but the `rs` information of all its descendants is deleted later on and the `child_count` becomes zero. At this point *cc* itself is not in the database, but its `child_count` and `rs` fields are. When the last of the descendants' `rs` field is removed there will be a recursive call to `DeleteResolveSources(cc)`, which will delete `cc.rs`. When a clause becomes an antecedent of a level zero constant, its `child_count` is incremented. This prevents the `child_count` from ever becoming zero, so the resolve sources for this clause are never deleted.

To show our algorithm is correct, we need to show that resolve trace information that is needed for the proof of unsatisfiability is never deleted. First, we claim that for every clause c, if the resolve sources information of c has not been

```
DeleteResolveSources(c) {
1  if (c.child_count == 0) {
2     if (¬ empty(c.rs) ∧ c.deleted) {
3        foreach (cl ∈ c.rs) {
4           --cl.child_count
5           DeleteResolveSources(cl)
6        }
7        delete c.rs
8     }
9  }
}
```

Figure 5: Recursive deletion of redundant resolve sources

deleted (i.e., `c.rs` $\neq \emptyset$), then for every clause c' that is a resolve source of c the resolve sources list of c' has not been deleted. This is easy to see by examining the code. The `child_count` of c' is incremented exactly once for each of its descendants, including c. The `child_count` is decremented when the resolve sources list of a descendant is deleted, at which point the descendant's `rs` field is also deleted. This implies that for each descendant, the `child_count` is decremented only once. Thus, the `child_count` of c' cannot become zero unless *all* descendants are removed, including c. So, as long as the resolve sources list of c is not deleted, the resolve sources lists of all its ancestors are maintained.

From here it is easy to conclude that the resolve sources information of all clauses in the transitive fan-in of the empty clause in RG is never going to be deleted. The proof is structured according to the structure of the proof of unsatisfiability, as described in Section 3:

1. The final conflicting clause exists in the SAT solver's database when the run ends and resolve sources lists are deleted only when clauses are deleted. Thus, the resolve sources list for the final conflicting clause exists at the end of the run.

2. For each level zero constant the algorithm increments the `child_count` of the corresponding antecedent clause. Once this happens, the `child_count` of this clause can never become zero, even if all its descendants are deleted. As a result, the resolve sources list of antecedents of level zero constants are never deleted.

3. If any other conflict clause appears in the proof of unsatisfiability, then it is either an ancestor of the final conflicting clause, or an ancestor of an antecedent of a level zero constant. We have already shown that these do not lose their resolve sources information, and thus all of their ancestors do not lose their resolve sources information.

4.1 Incremental SAT

In many SAT-based algorithms the use of incremental SAT [12, 14] enhances performance considerably. The basic idea is to share information learned by the SAT solver between consecutive SAT executions. This is achieved by sharing conflict clauses, so that later executions avoid repeating large amounts of computation.

When incremental SAT is put together with algorithms that require the generation of unsat cores we find that large parts of the resolve traces need to be maintained. Because source-descendant relationships are not maintained, there is no way to know which parts of the trace are going to be needed, without actually building the resolve graph. This has largely prevented the integration of these two techniques.

Our minimization algorithm leaves in the database the resolve source information for exactly those clauses that have a chance of appearing in some future proof of unsatisfiability. If a clause has not been deleted it may still participate in future proofs, and indeed the resolve sources lists for existing clauses and all of their ancestors are maintained. If a clause has been deleted but a descendant of this clause exists in the database, we are guaranteed that its information is not deleted. If a clause has been deleted, and all its descendants' information has been deleted, then there is no chance for it to be needed in any future proof, and indeed its information is deleted.

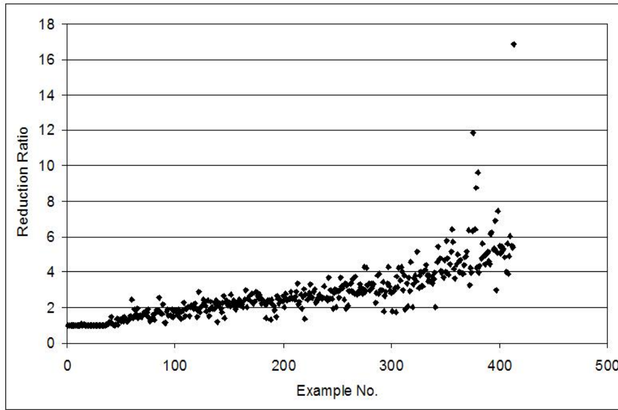
With our minimization algorithm the integration of incremental SAT is trivial and does not require any extra computation or disk space. This allows using incremental SAT in a wide range of SAT-based applications.

5. EXPERIMENTAL RESULTS

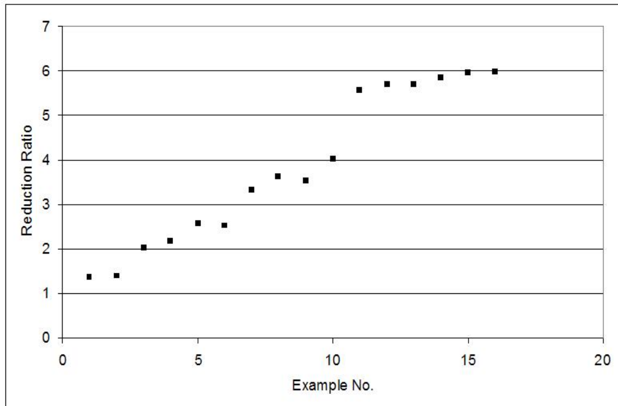
We implemented our technique in the Mage solver and collected results for two sets of examples. Mage is a Chaff [11] based SAT solver that is part of IBM's RuleBase PE [2] model checker. Mage uses Chaff's clause deletion mechanism. The first benchmark suite is created from running SAT-based Bounded Model Checking [5] and McMillan's Interpolation algorithm [9] on IBM designs. This suite includes designs from IBM's public benchmark suite [1] as well as new and confidential IBM designs, some of which are extremely large. The second suite is Grieu05, taken from the SAT competition of 2005 [4]. Of the hundreds of trace files produced by these examples 91 were over 200MB, and 43 were over 0.5GB. All our experimental results were obtained on an Intel Xeon dual CPU 2.4GHz, 2.5GB RAM running Linux.

When using the minimization algorithm, the resolve trace is kept in the memory in the form of the `rs` fields attached to conflict clauses. To measure the impact of the algorithm we dump these lists to a trace file at the end of each run, and compute the ratio between the size of the trace file without minimization, and the size after minimization. Figure 6 presents our experimental results in the form of ratio graphs. The examples are sorted on the X axis according to the size of the resolve trace *before minimization*, while the Y axis shows the minimization ratio obtained. Graph (i) shows results for our examples (both internal and public domain), and Graph (ii) gives results for the Grieu05 benchmarks set.

The experimental results are conclusive and show that our technique substantially decreases the sizes of resolve traces. This was achieved with only a minor change to the SAT solver's memory consumption. It is easy to see that the minimization ratio increases significantly with the size of the resolve trace. For large examples, the minimization ratio is greater than 3 for our examples and greater than 5 for Grieu05. In the largest examples, the ratio increases up to almost 17. In some cases, the resolve trace size was larger than 2GB and the SAT solver crashed due to a limitation of the file system, which is incapable of storing such large files. In these examples, our technique enabled the continuation of the SAT solver's run and eventually generated a resolve



(i) Public + Internal IBM Benchmarks



(ii) Grieu05

Figure 6: Resolve trace size reduction ratio

trace that was smaller than 400MB. We omitted these examples from the graphs since the minimization ratio can not be computed when the size of the resolve trace using the traditional technique is unknown.

While experimenting with the minimization algorithm we also measured the runtime of the solver and the maximum memory consumption during the run. We found out that in all 681 runs the maximum memory consumption with the minimization algorithm never went over the maximum memory consumption of the original algorithm plus the size of the (reduced) resolve trace. Since the reduced resolve trace is so much smaller than the original, keeping it in the memory of the process did not pose a problem. Moreover, using the minimization algorithm slightly decreased the runtime of the solver due to the removal of the costly I/O operations that the traditional technique requires. This reduction in runtime is small overall: no more than 1.1 speedup for examples that run over 5 minutes, but it still saves a few hundreds of seconds in runtime on the larger examples.

6. CONCLUSION

We presented a method that significantly reduces the size of a resolve trace produced by a SAT solver, and enables storing the resolve trace in the program's main memory. This alleviates the overhead of using proofs of unsatisfiability and unsat cores. We believe that this work will not only be integrated in most algorithms that use unsat cores, but will also encourage the development of new algorithms that leverage the power of unsat cores without paying a large overhead. In addition, this work enables integration of incremental SAT in existing and future algorithms, to boost their performance without suffering from huge resolve traces.

Our technique can be viewed as a reference counting technique, keeping information on whether or not there could be a sequence of pointers from the empty clause to each individual conflict clause. Unlike reference counting in garbage collection, where at every point during the computation the information regarding reachable memory elements is available, in our technique the information regarding the empty clause is known only after the SAT solver terminates.

This work was motivated by many reports of IBM verification engineers who suffer from serious disk quota problems. The minimization scheme is now implemented in our model checking tool RuleBase PE. It completely solved the disk quota problems and enabled verifying large blocks taken from new and complex IBM designs. Moreover, we implemented incremental SAT in all our algorithms and gain a tremendous speedup without suffering from any memory related problems.

7. REFERENCES

[1] IBM Formal Verification Benchmarks Library.

[2] IBM RuleBase Parallel Edition. 2004.

[3] R. J. Jr. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *IICAI*, 1997.

[4] D. Le Berre and L. Simon. SAT competition. 2005.

[5] A. Biere, A. Cimatti, E. Clark, and Y. Zhu. Symbolic model checking Without BDDs. In *TACAS*, 1999.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7), 1962.

[7] N. Een and N. Sorensson. An Extensible SAT-solver. In *SAT*, 2003.

[8] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5), 1999.

[9] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, 2003.

[10] K. L. McMillan and N. Amla. Automatic Abstraction without Counterexamples. In *TACAS*, 2003.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC*, 2001.

[12] O. Strichman. Pruning Techniques for the SAT-Based Bounded Model Checking Problem. In *CHARME*, 2001.

[13] C. Wang, H. Jin, G. D. Hachtel, and F. Somenzi. Refining the SAT Decision Ordering for Bounded Model Checking. In *DAC*, 2004.

[14] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: a new incremental satisfiability engine. In *DAC*, 2001.

[15] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *ICCAD*, 2001.

[16] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formulas. In *SAT*, 2003.