Searching for Autarkies to Trim Unsatisfiable Clause Sets

Mark Liffiton and Karem Sakallah

Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor 48109-2121
{liffiton,karem}@eecs.umich.edu

Abstract. An *autarky* is a partial assignment to the variables of a Boolean CNF formula that satisfies every clause containing an assigned variable. For an unsatisfiable formula, an autarky provides information about those clauses that are essentially independent from the infeasibility; clauses satisfied by an autarky are not contained in any minimal unsatisfiable subset (MUS) or minimal correction subset (MCS) of clauses. This suggests a preprocessing step of detecting autarkies and trimming such independent clauses from an instance prior to running an algorithm for finding MUSes or MCSes. With little existing work on algorithms for finding autarkies or experimental evaluations thereof, there is room for further research in this area. Here, we present a novel algorithm that searches for autarkies directly using a standard satisfiability solver. We investigate the autarkies of several industrial benchmark suites, and experimental results show that our algorithm compares favorably to an existing approach for discovering autarkies. Finally, we explore the potential of trimming autarkies in MCS- or MUS-extraction flows.

1 Introduction

Analysis of the infeasibility of unsatisfiable Boolean satisfiability problems has recently received increasing attention, though still little when compared to the efforts directed toward solutions to the problem of deciding the satisfiability of a Boolean formula (SAT). In many cases, the answer returned by a SAT solver given an infeasible formula, "UNSAT," is not sufficient information, and tools for further analysis are necessary.

Two such tools are the related concepts of minimal unsatisfiable subsets (MUSes) and minimal correction subsets (MCSes). Both MUSes and MCSes are irreducible portions of a formula that contain information relevant to understanding and correcting the formula's infeasibility while ignoring unrelated information. Several algorithms have been developed for computing MUSes and MCSes, including algorithms for finding a single, often approximate MUS (e.g., [6, 9, 16, 21]); finding a smallest MUS (SMUS, also called a minimum unsatisfiable core) [14]; and finding both all MCSes and all MUSes [13]. The work in [13], a focus of this paper, has found applications in finding all MCSes for circuit error diagnosis [17] and all MUSes as part of an abstraction refinement flow [1].

Autarkies provide another tool for looking into the structure of an unsatisfiable formula; they essentially provide information about portions of the formula that can be considered independent of the infeasibility. Autarkies have recently been linked to MUSes in [12], where Kullmann, et al., develop a classification of clauses in Boolean formulas based on their involvement in MUSes, autarkies, and resolution refutations. They use CAMUS ("Compute All Minimal Unsatisfiable Subsets") [13], a tool for computing all MUSes of a Boolean formula, and the only existing full approach for finding autarkies of which we are aware (first introduced in [11]) to investigate the complete set of MUSes and the autarkies, respectively, of a set of industrial benchmarks. They do not report runtime results, and we are not aware of any other experimental research on algorithms for finding the largest, or maximum, autarky of an instance.

In [12], the authors suggest two directions of research that we have undertaken in this paper:

1. An algorithm that directly searches for autarkies could be developed and compared to their algorithm, which makes use of a "duality" between autarkies and resolution refutations to find autarkies indirectly.
2. As clauses involved in autarkies are never contained in any MUS, such clauses can be removed as a preprocessing step for computing MUSes of a formula. (This also holds for MCSes, as they are comprised of the same clauses as MUSes.)

We have developed a novel algorithm, named Sifter, that directly performs a complete search for maximum autarkies, and we compare it to the existing approach based on resolution proofs. We also investigate the use of this algorithm as a preprocessing step to trim autarkies from unsatisfiable instances before searching for MUSes or MCSes.

This paper is organized as follows. Section 2 lays out formal definitions and concepts used throughout the paper. We review previous work related to autarkies in Section 3, and in Section 4, we introduce our new algorithm for finding maximum autarkies, Sifter. Experimental results comparing Sifter to the previous approach and investigating its use as a preprocessing step for two algorithms that operate on unsatisfiable formulas are shown and discussed in Section 5. Finally, Section 6 concludes with a brief overview of the paper and potential future work.

2 Preliminaries

Boolean Satisfiability and Conjunctive Normal Form. Formally, a Boolean formula C in conjunctive normal form (CNF) is defined as follows:

$$C = \bigwedge_{i=1\ldots n} C_i$$

$$C_i = \bigvee_{j=1\ldots k_i} a_{ij}$$

where each *literal* a_{ij} is either a positive or negative instance of some Boolean variable (e.g., x_3 or $\neg x_3$, where the domain of x_j is $\{0, 1\}$), the value k_i is the number of literals in the *clause* C_i (a disjunction of literals), and n is the number of clauses in the formula. In more general terms, each clause is a *constraint* of the constraint system C. A CNF instance is said to be *satisfiable* (SAT) if there exists some assignment to its variables that makes the formula evaluate to 1 or TRUE; otherwise, we call it *unsatisfiable* (UNSAT). A *SAT solver* evaluates the satisfiability of a given CNF formula and returns a satisfying assignment of its variables if it is satisfiable, and some produce *resolution refutations* (or *resolution proofs*) for unsatisfiable instances, directed acyclic graphs containing the resolution steps used to prove unsatisfiability.

The following unsatisfiable CNF instance C will be used as an example in this paper. We will refer to individual clauses as C_i, where i refers to the position of the clause in the formula (e.g., $C_3 = (\neg x_1 \vee \neg x_2)$).

$$C = (x_1)(\neg x_1 \vee x_2)(\neg x_1 \vee \neg x_2)(\neg x_2 \vee x_3)(x_4 \vee x_5)(\neg x_4 \vee \neg x_5)$$

AtMost Constraints. Our algorithm employs *AtMost* constraints, a type of counting constraint that can be constructed from Boolean CNF constraints or added to a SAT solver with few modifications. Given a set of n literals $\{l_1, l_2, \ldots, l_n\}$ and a positive integer k, s.t. $k < n$, an AtMost constraint is defined as

$$\text{AtMost}(\{l_1, l_2, \ldots, l_n\}, k) \equiv \sum_{i=1}^{n} \text{val}(l_i) \leq k$$

where $\text{val}(l_i)$ is 1 if l_i is assigned TRUE and 0 otherwise. This constraint places an upper bound on the number of literals in the set assigned TRUE.

This constraint can be encoded into Boolean CNF using encodings such as in [18], or it can be implemented efficiently in a SAT solver that employs watched variables (such as MiniSAT [7], which we use in this work). An implementation of an AtMost constraint can watch the assignments to the variables in the constraint and immediately propagate the negation of each remaining literal once k of them have been assigned TRUE. On a closed SAT solver that does not allow for a built-in implementation of the AtMost constraint, the CNF encoding can still be used.

Minimal Unsatisfiable Subsets / Minimal Correction Subsets. The definitions of *Minimal Unsatisfiable Subsets* (MUSes) and *Minimal Correction Subsets* (MCSes) of clauses are important to this work, as we are looking at the use of autarkies in preprocessing steps for algorithms that find MUSes and MCSes. An MUS is a subset of the clauses of an unsatisfiable formula that is unsatisfiable and cannot be made smaller without becoming satisfiable. An MCS is a subset of the clauses of an unsatisfiable formula whose removal from that formula results in a satisfiable formula ("correcting" the infeasibility) and that is minimal in the same sense that any proper subset does not have that defining property. Any unsatisfiable formula can have multiple MUSes and MCSes, potentially exponential in the number of clauses.

As proven in [3], there is a duality between MUSes and MCSes such that for a given instance, the complete set of MUSes (resp. MCSes) can be generated by finding all minimal hitting sets of the complete set of MCSes (resp. MUSes). This fact is used in [13] as the foundation for CAMUS, a set of two algorithms that computes all MUSes of an instance by way of first computing all MCSes. A corollary of this is that the union of all MUSes is equivalent to the union of all MCSes.

Our example contains one MUS, $\{C_1, C_2, C_3\}$, and its MCSes are the single-clause sets $\{C_1\}$, $\{C_2\}$, and $\{C_3\}$.

Autarkies. An *autarky* (or *autark assignment*) is an assignment to a subset of a formula's variables that satisfies every clause containing one of the assigned variables. Following the meaning of the term in other fields, it is a *self-sufficient* partial assignment. Because we are interested in trimming clause sets in this work, we will refer to autarkies in terms of the clauses they satisfy. Thus, the maximum autarky for us is the largest set of clauses satisfiable by an autarky, as opposed to the largest partial assignment. The maximum autarky for our example formula C is $\{C_4, C_5, C_6\}$, which in this case is the complement of the one MUS, and it is satisfied by the partial assignment $\{x_3 = \text{TRUE}, x_4 = \text{TRUE}, x_5 = \text{FALSE}\}$.

As explained in [10], any clause satisfied by some autarky can not be contained in any MUS (nor in any MCS, as they are comprised of the same clauses). This motivates the idea of preprocessing unsatisfiable formulas by removing their maximum autarkies before searching for MUSes or MCSes.

Pure Literals. One simple form of autarky arises from *pure literals.* A pure literal is a variable that occurs in only one polarity (either always positive or always negated) in a CNF formula. In our example formula, x_3 is a pure literal, because $\neg x_3$ is not present. An assignment of TRUE to a pure literal will trivially satisfy any clause containing the corresponding variable, thus any such assignment is an autarky.

Pure literals can be found in a linear time scan of a formula. Removing the clauses satisfied by pure literals may cause other literals to become pure in the formula, so repeatedly detecting, recording, and removing pure literals is a simple first step for any algorithm that finds autarkies. The process terminates when the formula no longer contains pure literals.

3 Previous Work

Monien and Speckenmeyer [15] first introduced the concept of an autark assignment or autarky, using autarkies in a modification of the DPLL satisfiability algorithm [4, 5] that reduced its complexity upper bound below 2^n splitting steps (for a formula with n variables). Autarkies were later used in another satisfiability algorithm by Van Gelder [20] named Modoc. Modoc integrates autarky pruning, removing those clauses satisfied by autarkies, into a resolution-based

model elimination approach to satisfiability. Both Monien and Speckenmeyer's algorithm and Modoc find autarkies as side-effects of their operation, but neither is aiming to find the maximum autarky. Additionally, both find many more "conditional autarkies," i.e., autarkies that appear after propagating a partial assignment through the formula, than "top-level autarkies" for the entire formula.

More recently, Kullmann has investigated autarkies in several papers. In [10], he introduces the idea of *lean clause-sets*, sets of clauses that have no autarkies. The largest lean clause-set is the complement of the maximum autarky of a formula; all clauses can be partitioned into one or the other. Kullmann investigates a special case of autarky that can be found in polynomial time using linear programming, though this does not generalize to finding all autarkies. He also proves, with Theorem 3.16, that a set of clauses F is lean "if and only if every clause of F can be used by some resolution refutation of F." Conversely, a set of clauses $A \subseteq F$ is an autarky if and only if each clause in A can *not* be used in any resolution refutation of F. Later, in [11], Kullmann uses this fact to develop an algorithm for computing the maximum autarky. Using a SAT solver that provides a resolution refutation for unsatisfiable instances, one can iteratively remove the variables included in some resolution proof. When the reduced formula becomes satisfiable, the satisfying assignment is an autarky of the original formula. This is the algorithm to which we compare ours in Section 5.

Finally, Kullmann, et al. [12] use both autarkies and MUSes as tools to describe and examine unsatisfiable formulas. They characterize clauses in such formulas into several classes based on each clause's involvement in MUSes, resolution refutations, and autarkies. Clauses contained in every MUS are called "necessary"; those in any MUS are "potentially necessary"; "usable" indicates a clause is in some resolution refutation; and thus "unusable" refers to clauses in an autarky. Complements and intersections of these classes are defined as well. They experimentally evaluate a set of industrial benchmarks from an automotive product configuration domain [19], reporting on the MUSes and clauses in the different levels of "necessity" in each instance. To compute all MUSes of the instances, they use CAMUS [13], and they found maximum autarkies using the algorithm described in [11], implemented using the ZChaff SAT solver's ability to produce resolution refutations [21].

4 Searching for Autarkies

Our approach to the problem of finding the maximum autarky for a formula treats it as an optimization problem. We search for the largest partial assignment that satisfies the clauses it touches, i.e., the largest autarky, by explicitly searching in the space of all partial assignments and maximizing the size of the result (in terms of the number of satisfied clauses). Specifically, we "instrument" the formula to give a standard SAT solver the ability to enable and disable individual clauses and variables within its normal search, and we use AtMost

constraints to perform a sliding objective maximization of the autarky size. This draws inspiration from a similar technique we employed in [13] that uses a less-involved instrumentation and the same optimization technique to allow a SAT solver to search for maximal satisfiable subsets of clauses. We directly exploit the efficiency gains made in SAT solvers in recent years by using an "off-the-shelf" solver; our algorithm works with any solver[1], so it can benefit from future improvements as well.

4.1 Instrumentation

To give a SAT solver the ability to search for autarkies, we instrument a formula C with the following modifications:

1. We replace every literal in the formula with a *literal-substitute*; x_j in the formula becomes x_j^1, while $\neg x_j$ is replaced with x_j^0.
2. Each clause C_i is augmented with a *clause-selector* y_i to form a new clause $C'_i = (y_i \rightarrow C_i) = (\neg y_i \vee C_i)$.
3. We create a *variable-selector* x_j^+ for every variable x_j. When x_j^+ is TRUE, x_j will be enabled, and it is disabled otherwise. For every variable x_j, we add clauses to relate its variable-selector x_j^+, its two literal-substitutes x_j^0 and x_j^1, and the value of the variable itself, x_j. In short, we want each literal-substitute to be TRUE when the variable is enabled (x_j^+ is TRUE) and x_j has the corresponding value. This leads to new clauses encoding the following: $(x_j^1 = x_j^+ \wedge x_j)$ and $(x_j^0 = x_j^+ \wedge \neg x_j)$.
4. Finally, we add clauses to require that a clause be enabled (y_i = TRUE) if any one of its variables is enabled. Thus, for any x_j present in clause C_i, we add a clause $(x_j^+ \rightarrow y_i) = (\neg x_j^+ \vee y_i)$.

This is not the only option for instrumenting the formula; other encodings have the same effect. However, while preliminary experiments showed that similar encodings yield slightly different runtimes, the differences in efficiency were not substantial.

The complete instrumented formula for our example is too large to be useful here, but here we show the constraints produced from a single clause of the example, C_2:

$$C_2, (\neg x_1 \vee x_2) \Longrightarrow \left\{ \begin{array}{rl} 1 \;\&\; 2\text{:} & (\neg y_2 \vee x_1^0 \vee x_2^1) \\ 3\text{:} & \begin{array}{l} (x_1^1 = x_1^+ \wedge x_1)(x_1^0 = x_1^+ \wedge \neg x_1) \\ (x_2^1 = x_2^+ \wedge x_2)(x_2^0 = x_2^+ \wedge \neg x_2) \end{array} \\ 4\text{:} & (\neg x_1^+ \vee y_2)(\neg x_2^+ \vee y_2) \end{array} \right\}$$

The clause derived from modifications 1 and 2 replaces the original clause, while the rest are additions. The clauses from modification 3 (presented in shorthand as equalities; each is three clauses in CNF) are specific to variables, and

[1] SAT solvers that implement AtMost constraints internally will likely perform better than those that require using a CNF encoding of them, but all will work.

the complete formula will only contain each set once per variable. The final two clauses, resulting from modification 4, are specific to C_2.

With the formula instrumented in this way, any satisfying assignment will indicate an autarky of the original formula. The x_j^+ variables indicate which variables are "activated," i.e., included in the autarky; the original variables contain the autarky assignment; and the clauses satisfied by the autarky are represented by those y_i variables set to TRUE. One such assignment is the trivial solution in which all variables and all clauses are disabled. To find the maximum autarky, we must maximize the number of enabled clauses.

4.2 Our Algorithm

We maximize the number of enabled clauses (y_i variables assigned TRUE) by way of an iterative optimization approach. We use AtMost constraints to bound the number of disabled clauses, tightening the bound as solutions are found. If an autarky is found that leaves n clauses disabled, we start the search for a larger autarky by bounding the disabled clauses to $n - 1$. Eventually, if the instance is unsatisfiable, we will reach a bound k for which no solution can be found. At this point, we have proven that there exists an autarky of size $k - 1$ and none with size k, thus the previously found autarky is the maximum autarky.

```
Sifter(C)
 1. (C, autarky) ← PureLits(C)
 2. C' ← Instrument(C)
 3. bound ← |C| − 1
 4. loop
 5.     C'_b ← C' ∧ AtMost({¬y_1, ¬y_2, ..., ¬y_n}, bound)
 6.     (isSAT, model) ← Solve(C'_b)
 7.     if not isSAT
 8.         return autarky
 9.     autarky ← autarky ∪ SatisfiedClauses(model)
10.     bound ← |C| − |autarky| − 1
```

Fig. 1. Sifter finds the maximum autarky of a CNF formula C by "instrumenting" the instance and using a SAT solver to search for satisfying partial assignments.

Figure 1 contains pseudocode for the complete algorithm, which we call Sifter. First, we repeatedly scan for pure literals, recording and removing them as described in Section 2: the call to **PureLits** returns 1) C with any clauses containing pure literals removed and 2) the set of such clauses as an initial autarky. We then instrument the formula and use the sliding objective method described above to find the rest of the maximum autarky or to prove that the pure literal approach found it in its entirety. The **Instrument** subroutine produces instrumented clauses via the modifications described in Section 4.1. The bound on the

number of disabled clauses is set initially to $|C| - 1$ to begin the search by looking for an autarky that satisfies at least one clause, and the loop then proceeds by searching for a satisfying assignment, `model`, of the instrumented, bounded formula, C'_b. If none is found (`isSAT` is false), the algorithm returns `autarky`, which must be the maximum autarky. Otherwise, the satisfied clauses are added to `autarky`, the bound is set to search for an autarky that satisfies at least one more clause, and the loop repeats.

5 Experimental Results

Our two experimental goals are 1) to compare and contrast Sifter, our direct search-based approach for finding the maximum autarky, with the earlier iterative technique using resolution refutation trees [11], and 2) to investigate the value of trimming autarkies as a preprocessing step for finding MUSes and MCSes.

5.1 Comparing Search to an Iterated Resolution Proof Approach

We implemented Sifter in C++ using MiniSAT [7] version 1.12b (the last version containing support for AtMost constraints). We wrote the iterative approach [11], which we will call Scraper, as a Perl script. First, Scraper uses the pure literal elimination written for Sifter, making that phase equivalent in both implementations. Then, it employs the tools `zchaff` and `zverify_df` [21] from the ZChaff distribution `zchaff.64bit.2007.3.12` to repeatedly produce resolution refutations and eliminate the involved variables until the instance becomes satisfiable. We compiled all executables for the x86-64 instruction set using GCC 4.1.2 with standard optimizations, and all experiments were run under Linux (Fedora 7) on a 3.0GHz Intel Core 2 Duo E6850 with 4GB of RAM.

Figure 2 contains a log-log scatterplot comparing the runtimes of Sifter and Scraper on a variety of industrial benchmarks. Runtimes for Sifter are represented on the y-axis, so points lying below the diagonal indicate instances in which Sifter outperforms Scraper. A timeout of 600 seconds was used for every run, indicated by the dashed lines on the extremes of the chart; points on these lines indicate that a timeout was reached by the corresponding algorithms. The reported runtimes are processor time, which for Sifter are essentially equivalent to wall-clock time. Our implementation of Scraper, however, stores several intermediate results to disk; we ignore this I/O time in these results to estimate the runtime of a more efficient approach that retains everything in memory.

To provide a more complete understanding of these results, Table 1 lists some overall characteristics of each benchmark family. We list the minimum and maximum number of variables, number of clauses, and size of the maximum autarky (in clauses) for the instances in each family. The Benz benchmarks[2] are the automotive product configuration instances from [19] used in the experiments in [12]. The Miter family[3] contains equivalence checking instances from João Marques-

[2] `http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/`
[3] `http://sat.inesc.pt/benchmarks/cnf/equiv-checking/instances/`

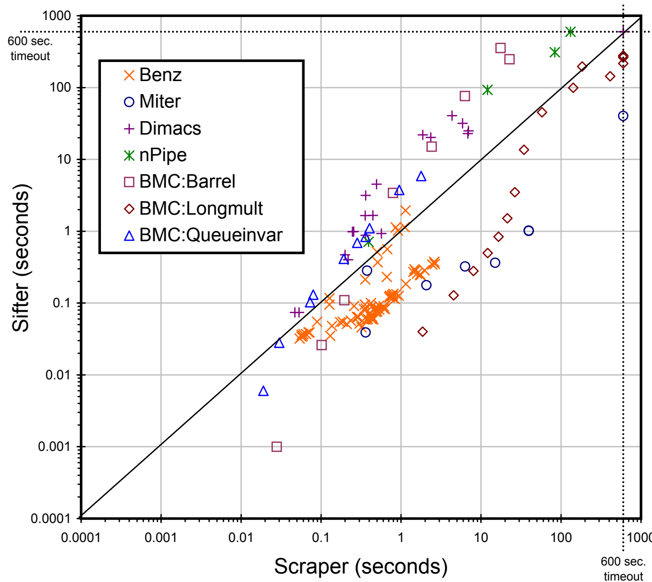

Fig. 2. Comparing the performance of Sifter and Scraper on a variety of benchmarks

Silva. The Dimacs instances are circuit benchmarks from the DIMACS set. The nPipe instances are from Miroslav Velev's FVP-UNSAT-2.0 benchmarks[4], generated for the formal verification of microprocessors, with redundant variables removed. The BMC:[] instances[5] are formulas used in bounded model checking (BMC) as described in [2].

From Figure 2 and Table 1, we can draw several conclusions:

1. Across all of the benchmarks, neither Sifter nor Scraper dominates the other in terms of runtime. In some benchmarks, Scraper is faster, up to 20x, while in others, Sifter is faster, up to 46x.
2. In just those benchmarks with non-trivial autarkies, however, our Sifter algorithm is faster in nearly every instance. Specifically, looking at the Benz and Miter families (the autarkies covering 2 clauses in each BMC:Longmult instance are all found by pure-literal elimination alone), we see that Sifter outperforms Scraper by approximately one order of magnitude.
3. The presence and size of autarkies is fairly consistent *within* benchmark families. Each particular family in Dimacs, nPipe, and BMC:[] has either no autarkies in any instance or an autarky that covers 2 clauses in each. The Benz family consistently has autarkies that cover a large portion (between 32 and 98 percent) of each instance's clauses. Every instance in the Miter family has a non-empty autarky, though the autarky sizes vary more than they do in the Benz instances.

[4] `http://www.miroslav-velev.com/sat_benchmarks.html`
[5] `http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html`

Family	Variables		Clauses		|autarky|	
	min	max	min	max	min	max
Benz	1,513	1,891	4,013	9,957	2,097	7,025
Miter	1,266	17,303	1,027	34,238	1	1,831
Dimacs	389	7,767	1,115	20,812	0	0
nPipe	861	15,469	6,695	394,739	0	0
BMC:Barrel	50	8,903	159	36,606	0	0
BMC:Longmult	437	7,807	1,206	24,351	2	2
BMC:Queueinvar	116	2,435	399	20,671	0	0

Table 1. Benchmark Characteristics

Overall, these conclusions imply a strategy for exploiting autarkies in practice. First, by searching for autarkies on a small representative set of instances from a particular application, one can determine whether the instances in that domain have autarkies at all. If none of the test set have autarkies of any appreciable size, then it is likely that none generated in the application will, in which case autarkies will be of no use. This is likely in applications such as bounded model checking, where performing a cone of influence reduction of the circuit will likely eliminate all autarkies. In these applications, checking for autarkies could be a simple test of the sanity of the CNF encoding. In the other case, in which instances do contain autarkies, it is probable that most if not all instances will have autarkies, and Sifter is likely the more efficient algorithm to use.

5.2 Trimming Autarkies to Boost Searching for MUSes and MCSes

Trimming autarkies holds the most promise for boosting algorithms that have a high complexity and are affected heavily by the number of clauses in an instance. An algorithm for finding any single unsatisfiable subformula, such as that developed in ZChaff [21], is unlikely to benefit from such boosting, as the time taken to find the maximum autarky will likely dwarf the runtime of the unboosted algorithm.

We identified two algorithms that *are* good candidates for this boosting. One is the first phase of CAMUS [13], which finds all MCSes of a formula as the first step of solving several related problems such as computing all MUSes or finding the smallest MUS (SMUS) of a CNF formula. In addition to computing MUSes, this algorithm has been applied (without the second phase) in a circuit error diagnosis system [17], in which the MCSes were used directly. A second algorithm with potential for boosting by trimming autarkies is that by Mneimneh, et. al. [14] for computing an SMUS directly, which we will refer to as SMUS. Both of these candidate algorithms use clause-selector variables (as used in Sifter and described in Section 4.1) and use a SAT solver to implicitly search through subsets of clauses. Therefore, both can benefit from the reduced search space produced by a reduction in the number of input clauses.

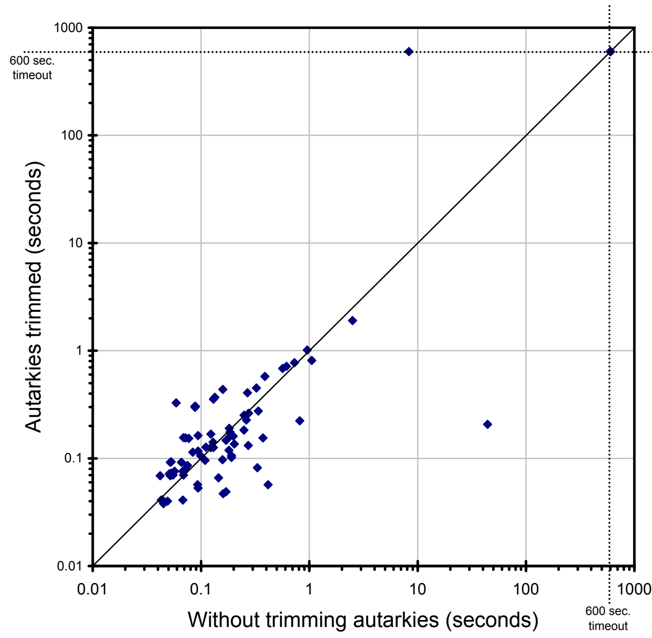

Fig. 3. Boosting SMUS by trimming autarkies for the Benz benchmarks

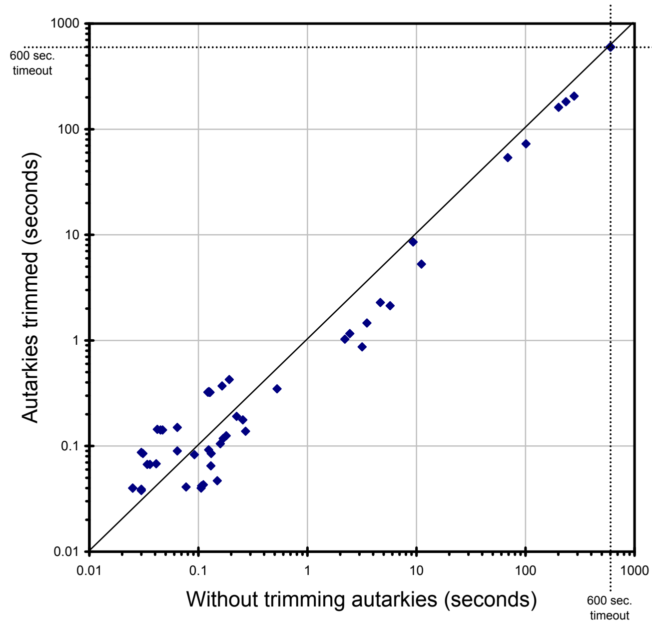

Fig. 4. Boosting CAMUS (first phase) by trimming autarkies for the Benz benchmarks

We investigated the impact of trimming autarkies on both of these algorithms for the Benz benchmarks, which have the largest autarkies, and the results are displayed in Figures 3 and 4. Each figure is a log-log scatterplot that charts the runtime of the specified algorithm alone on the x-axis against the runtime of the boosted version on the y-axis. The runtime reported for the boosted version is the sum of finding an instance's maximum autarky with Sifter and running the algorithm on the trimmed instance. A point below the diagonal indicates an instance for which the boosting produced a net decrease in runtime.

The results are mixed. In Figure 3, we see that the boosting does not produce markedly better or worse results overall for finding SMUSes with SMUS. While the runtimes for SMUS alone (not shown) do improve in nearly all cases when it is run on the trimmed instances, the runtime of Sifter outweighs this gain in many cases. There are two outliers: one in which SMUS's runtime improves by over two orders of magnitude when run on the trimmed instance, and another that takes less than 10 seconds on the untrimmed instance yet times out at 600 seconds on the trimmed version. These are artifacts of the susceptibility of combinatorial search algorithms like SMUS to variations in runtime due to minor ordering changes and similar effects.

The results for boosting the first phase of CAMUS, shown in Figure 4, show that the boosting does have value in some cases. For this algorithm, the runtime of Sifter can outweigh the decrease in runtime due to the boosting in cases with *small* runtimes (below 1 second in these benchmarks), but the boosted algorithm always outperforms the original algorithm in cases with longer runtimes. Taken as a whole, this is a net benefit, because the runtime increases in some "small" instances are far outweighed by the gains in the "large" instances. The total runtime, over all instances that did not time out in both techniques, decreased from 931 seconds on untrimmed instances to 704 seconds for the boosted algorithm, a 24% decrease in total runtime.

6 Conclusions and Future Work

We are aware of only one existing algorithm for computing maximum autarkies, presented in [11], and no experimental research investigating the runtime of finding maximum autarkies has been published prior to this work. Furthermore, little research has been conducted in the area of autarkies for Boolean satisfiability, and no "industrial" application has previously been identified for them.

In this paper, we have presented a new algorithm, Sifter, for finding maximum autarkies that searches for them directly with a standard SAT solver and an "instrumented" formula. We have evaluated it experimentally, comparing it to the existing approach based on iterated construction of resolution refutations, on a variety of industrial benchmarks. In our results, Sifter outperforms the other algorithm on benchmarks with autarkies, though the results are mixed on those with none.

We have also performed an initial exploration of the use of autarky trimming as a preprocessing step for complex algorithms for finding MUSes and MCSes.

We used Sifter to boost two different algorithms by trimming maximum autarkies from instances before searching for MUSes or MCSes. While the boosted version of an algorithm for finding the smallest MUS of a formula was not (overall) faster or slower than the normal version, the boosted version of the first phase of CAMUS [13], which finds all MCSes of an instance, was noticeably faster. The overhead of the trimming often outweighed runtime gains on instances that completed in under one second, but the trimming was beneficial on long-running instances; we obtained a total runtime reduction of 24% over all instances that did not time out.

As future work, we can look into improving the efficiency of Sifter, possibly by using a new encoding to instrument formulas or by employing a different optimization method. Also, more work can be done to explore the structure and characteristics of autarkies in real-world Boolean formulas; the results here show that there is much variation among benchmark families, with some containing no autarkies at all. The use of conditional autarkies (autarkies that arise following the assignment of some variables) in algorithms for analyzing unsatisfiable instances is worth investigating as well. In the area of boosting MUS algorithms, it would be interesting to compare autarky-trimming with the local search used to boost the first phase of CAMUS in [8] by quickly identifying candidate MCSes before the complete search begins.

Acknowledgments

This material is based upon work supported by the National Science Foundation under ITR Grant No. 0205288. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation (NSF).

References

1. Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of the 2006 conference on Asia South Pacific design automation (ASP-DAC'06)*, pages 19–24, 2006.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207, 1999.
3. E. Birnbaum and E. L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. *Journal of Experimental and Theoretical Artificial Intelligence*, 15:25–46, 2003.
4. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
5. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
6. N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Proceedings of the 9th International Conference on Theory*

and Applications of Satisfiability Testing (SAT-2006), volume 4304 of *LNCS*, pages 36–41, 2006.
7. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, volume 2919 of *LNCS*, pages 502–518, 2003.
8. É. Grégoire, B. Mazure, and C. Piette. Boosting a complete technique to find MSSes and MUSes thanks to a local search oracle. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, volume 2, pages 2300–2305, January 2007.
9. É. Grégoire, B. Mazure, and C. Piette. Local-search extraction of MUSes. *Constraints*, 12(3):325–344, 2007.
10. O. Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.
11. O. Kullmann. On the use of autarkies for satisfiability decision. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT-2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, pages 231–253, 2001.
12. O. Kullmann, I. Lynce, and J. Marques-Silva. Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-2006)*, volume 4121 of *LNCS*, pages 22–35, 2006.
13. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, January 2008.
14. M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. M. Silva, and K. A. Sakallah. A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005)*, volume 3569 of *LNCS*, pages 467–474, 2005.
15. B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, March 1985.
16. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*, pages 518–523, 2004.
17. S. Safarpour, M. Liffiton, H. Mangassarian, A. Veneris, and K. Sakallah. Improved design debugging using maximum satisfiability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 13–19, November 2007.
18. C. Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 827–831, 2005.
19. C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, 2003.
20. A. Van Gelder. Autarky pruning in propositional model elimination reduces failure redundancy. *Journal of Automated Reasoning*, 23(2):137–193, 1999.
21. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *The 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, 2003.