The Logic in Computer Science Column

by

Yuri Gurevich

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
gurevich@microsoft.com

Anatomy and Empirical Evaluation of Modern SAT Solvers

Karem A. Sakallah
EECS, University of Michigan
karem@umich.edu

Joao Marques-Silva
CSI/CASL, University College Dublin
jpms@ucd.ie

Abstract

The Boolean Satisfiability (SAT) decision problem can be deservedly declared a success story of computer science. Although SAT was the first problem to be proved NP-complete, the last decade and a half have seen dramatic improvements in the performance of SAT solvers on many *practical* problem instances. These performance improvements enabled a wide range of real-world applications, several of which have key industrial significance. This article surveys the organization of modern conflict-driven clause learning (CDCL) SAT solvers, focusing on the principal techniques that have contributed to this impressive performance. The article also empirically evaluates these techniques on a comprehensive suite of problem instances taken from a range of representative applications, allowing for a better understanding of their relative contribution.

Bulletin of the EATCS no 103, pp. 96–121, February 2011

1 Introduction

SAT solving technology has advanced significantly over the past 15 years. There are currently dozens of SAT solvers that extend the basic DPLL framework, explained in the next section, with a combination of algorithmic enhancements and optimized data structures. The advances have been spurred in part by regular international competitions and a growing archive of SAT instances from a wide range of real-world applications. Modern SAT solvers differ in many aspects, but most of them include the following four features that have been shown, through extensive empirical evidence, to be critical for scalability and performance:

- Conflict-driven clause learning [33, 34],
- Search restarts [26],
- Boolean constraint propagation based on lazy data structures [37], and
- Conflict-based adaptive branching [37]

The immediate aim of this article is to briefly recount the development of these features and to experimentally characterize their contribution in solving a suite of 1000 benchmarks chosen from 12 diverse application areas. A larger goal is to stir the interest of the theoretical computer science community in developing appropriate theoretical models that can explain the remarkable performance of these modern solvers, and also to explain why they still fail in some cases. To be sure, some efforts along these lines have already been made, but much more needs to be done. Further progress in this field will certainly benefit from a combination of theoretical and experimental/algorithmic developments.

Following a review, in Section 2, of the major developments in SAT technology, Section 3 describes the twelve configurations we instrumented in the MiniSat [19] solver for the purpose of isolating the contribution of the aforementioned features of modern SAT solvers. Section 4 describes the benchmark suite we chose for this study and discusses our reasoning for the choice. The results of the study are presented and analyzed in Section 5; we also provide, in an appendix, detailed run time distributions that compare the various solver configurations for each benchmark family. The paper ends with some concluding thoughts in Section 6.

2 A Brief History of SAT Solvers

The first practical algorithm for solving the SAT problem is usually credited to Martin Davis and his collaborators Hilary Putnam, George Logemann, and Don-

ald Loveland[1]. The context was the development of automated procedures for proving theorems in quantification theory and entailed a) translation of a quantified first-order logic formula into a sequence of propositional formulas in conjunctive normal form (CNF), and b) checking the resulting formulas for satisfiability. The first version of the algorithm appeared in 1960. It was authored by Davis and Putnam [15] and consisted of three basic rules:

Rule I simplifies the CNF formula by eliminating one-literal clauses.

Rule II identifies variables that occur only positively or only negatively in the formula and deletes all clauses in which such variables occur.

Rule III eliminates variables that occur in both polarities.

In modern terminology, Rule I is referred to as the unit-clause rule and its repeated application to simplify a formula is known as Boolean Constraint Propagation (BCP) [54], whereas rule II is commonly referred to as the pure-literal rule. The bulk of the algorithmic complexity is in rule III which replaces all clauses that *mention* a particular variable with the resolvents involving this variable. Quoting from [15],

> III. *Rule for Eliminating Atomic Formulas.* Let the given formula F be put into the form
>
> $$(A \vee p) \,\&\, (B \vee \bar{p}) \,\&\, R$$
>
> where A, B, and R are free of p.
> . . .
> Then F is inconsistent if and only if $(A \vee B) \,\&\, R$ is inconsistent.

Note that $(A \vee B) \,\&\, R$ is what results when p is existentially quantified from F. Note also that $(A \vee B)$ must be "multiplied out" to put it back into CNF[2] and that the resulting CNF clauses represent all p-resolvents in F[3]. The complete algorithm repeatedly cycles through these three rules terminating when the resulting formula evaluates to 1 (resp. 0) indicating satisfiability (resp. unsatisfiability) of the original formula.

[1] Davis and Putnam [15] credit Wang [53] and Gilmore [24] with earlier attempts that "both run into difficulty with some fairly simple examples."

[2] Note that the form $(A \vee p) \,\&\, (B \vee \bar{p}) \,\&\, R$ is shorthand for $(A_1 \vee p) \,\&\, \cdots \,\&\, (A_m \vee p) \,\&\, (B_1 \vee \bar{p}) \,\&\, \cdots \,\&\, (B_n \vee \bar{p}) \,\&\, R$ where each A_i and B_j is a disjunction of literals other that p and $\bar{p}$. Thus, $(A \vee B) = (A_1 \,\&\, \cdots \,\&\, A_m) \vee (B_1 \,\&\, \cdots \,\&\, B_m) = (A_1 \vee B_1) \,\&\, \cdots \,\&\, (A_m \vee B_n)$.

[3] The p-resolvent of $(A_i \vee p)$ and $\left(B_j \vee \bar{p}\right)$ is the clause $\left(A_i \vee B_j\right)$ which is the disjunction of all literals in the two original clauses other that p and $\bar{p}$. Note that $(A_i \vee p) \,\&\, \left(B_j \vee \bar{p}\right) \rightarrow \left(A_i \vee B_j\right)$.

While appearing to solve the problem in a linear number of steps, this algorithm hides the fact that the size of intermediate formulas can grow exponentially, thus severely limiting its scalability. In the second version of the algorithm, published in 1962 [14], Davis, Logemann, and Loveland replaced rule III with a *splitting rule* that they referred to as rule III*. Quoting from [14],

> III*. *Splitting Rule.* Let the given formula F be put into the form
>
> $$(A \vee p) \,\&\, (B \vee \bar{p}) \,\&\, R$$
>
> where A, B, and R are free of p. Then F is inconsistent if and only if $A \,\&\, R$ and $B \,\&\, R$ are both inconsistent.

Thus, rather than eliminating a variable as in rule III, the new variant considers, in sequence, the two subformulas $A \,\&\, R$ (resp. $B \,\&\, R$) which are obtained by fixing the value of the chosen splitting variable to 0 (resp. 1). If the satisfiability of either subformula cannot be immediately determined, splitting is repeated by fixing additional variables. This process amounts to a depth-first exploration of the space of truth assignments and avoids the memory blow-up inherent in the first version of the algorithm. Completeness is insured by maintaining a stack of the subformulas that are created by splitting and by *chronologically backtracking* to them whenever later subformulas are found to be unsatisfiable. These two variants have traditionally been referred to as DP and DLL. Recently, though, DPLL has been increasingly used to refer to the DLL version and we follow this usage in the remainder of the article.

The DPLL algorithm established the basic architecture for all subsequent complete SAT solvers and identified the major computational steps to be:

Branching which is the mechanism for moving forward in the search space,

(Unit) Propagation which is the mechanism for deducing appropriate consequences, also known as *implications*, of branching choices,

Backtracking which is the mechanism for retracting from regions of the search space that do not contain satisfying assignments.

Of course, this description leaves many details unspecified including how to select variables and values for branching and how to detect unit clauses and propagate implications.

Until the mid 1990s, implementations of DPLL followed the original algorithm rather closely, differing primarily in the heuristics used for branching [13, 17, 42, 51, 29, 6, 21, 50]. In particular, these *early solvers* executed chronological backtracking, and their capacity was limited to SAT instances with variable counts

in the low hundreds. The first major enhancement to DPLL came in 1996 with the debut of the GRASP solver [33, 34]. Building on ideas from the Artificial Intelligence (AI) and Constraint Satisfaction Problems (CSP) communities, GRASP introduced a novel procedure for clause learning from conflicts, i.e., from partial assignments that cause the formula to evaluate to 0. Concepts such as *dependency-directed backtracking*, *learning from failures*, and *nogoods* appeared in the mid to late 1970s in connection with the development of truth maintenance systems for planning [20], circuit analysis [49] and medical diagnosis [45]. Similar ideas, such as backjumping and conflict-directed backjumping, and learning were developed later for solving constraint satisfaction problems [16, 43]. The key insight in all of these approaches is the inevitability of failure in search algorithms, i.e., the near-certainty (and in case of unsatisfiable instances the absolute certainty) that regardless of how clever the branching heuristics are, a search procedure cannot avoid visiting parts of the search space that contain no solutions. These failures occur because the set of constraints that characterize the search space are generally incomplete in the sense that they do not explicitly capture all possible interactions among the variables. In other words, failures, which we will henceforth refer to as *conflicts*, signify missing constraints. Thus, by analyzing conflicts we can *learn* additional constraints that will help avoid similar conflicts later on in the search.

Learning from conflicts takes a particularly simple and elegant form in SAT. A conflict is captured as a partial variable assignment, namely the assignment that causes the CNF formula to evaluate to 0. Negating this assignment yields the desired missing constraint in the form of a *conflict-induced clause*. Having the same form as the original formula clauses, these learned clauses do not require any special treatment and can be processed similarly to other clauses by the search algorithm.

A significant feature of clause learning in GRASP is the way it ties together branching, propagation, and backtracking to create effective learned clauses. To appreciate this, note that a naive form of learning is to simply negate the entire set of branching decisions that led to the present conflict. For deep conflicts, the resulting learned clause would be unnecessarily long and not particularly useful for eliminating future conflicts. Effective learning requires identifying a small set of assignments that are sufficient to expose the conflict. GRASP identifies such a set by keeping track of the implication sequences during propagation which it implicitly captures as an *implication graph*. Nodes in this graph represent either decision assignments or implications due to propagation. Each assignment is also labeled with a *decision level* that indicates when it was created. This level is incremented each time a branching decision is made, and all implied assignments caused by that decision inherit that decision's level. Directed edges in this graph capture unit propagation and are labeled with the unit clauses responsible for propagation. Various cuts in this graph correspond to independent sets of assignments

each of which representing a "witness" of the conflict. In particular, GRASP introduced the notion of unique implication points[4] (UIPs) which correspond to the dominators in the implication graph and demonstrated that UIP-based learning yields short effective conflict-induced clauses. This style of clause learning also allowed the algorithm to backtrack non-chronologically to the decision level of the *latest* assigned variable in the learned clause. GRASP also introduced the Dynamic Largest Individual Sum of literals (DLIS) branching heuristic and experimentally demonstrated its advantage over other heuristics in common use at the time [32]. DLIS maintains counts of literals in unresolved clauses and uses for its next branching decision the literal with the largest count. The enhancements introduced in GRASP, particularly conflict analysis and clause learning, enabled, for the first time, the solution of SAT instances with up to tens of thousands of variables and hundreds of thousands of clauses.

The next milestone in the development of modern SAT solvers was triggered by the observation that the run time distributions of DPLL solvers on satisfiable instances tend to have *heavy tails* when branching is randomized [26]. This should not be too surprising since a satisfiable SAT instance can be solved very quickly given a perfect branching sequence (i.e., given the *solution*) but can also take an exponential amount of time with a poor choice of branching decisions. To mitigate against this inherent variability in run times, Gomes et al [26] proposed the use of *rapid randomized restarts* so that a solver can escape from bad regions of the search space. Restarts are typically triggered after the solver performs a given number of backtracks. Run lengths can also be varied between restarts using a variety of policies such as Luby [1] which is based on the sequence $1, 1, 2, 1, 1, 2, 4, \cdots$. Search restarts were later shown to work effectively with clause learning [5, 37], and to be useful even for unsatisfiable instances. In addition, researchers continue to explore various heuristics that can improve the effectiveness of restarts such as adaptive restarts [8] and problem-specific restart heuristics [46].

Modern SAT solvers took their final shape with two further enhancements that were introduced in the Chaff solver [37]. Both enhancements were motivated by the desire to significantly reduce the computational cost of branching and propagation which dominated the overall run time of conflict-driven solvers such as GRASP. The first enhancement replaced counter-based procedures for identifying unit clauses and performing BCP with a clever algorithm that has come to be known as the *two-watched-literals* scheme. In this scheme, which generalizes a similar idea in the SATO solver [55], the status of a clause is maintained by "watching" just two of its literals that are not currently assigned to 0 regardless

[4] A notion inspired by unique sensitization points from the field of automatic test pattern generation [23].

of how many other literals it may have. The status is updated only when one of the watched literals is assigned to 0. In that case, another literal that is not currently assigned to 0 must be found and watched; the clause becomes unit if no such literal can be found except the other watched literal. The key idea in this scheme is to lazily update the status of a clause since we *only* need to know when it becomes unit; assignments to literals that are not watched are irrelevant and do not incur any overhead. Furthermore, assigning 1 to a watched literal, or unassigning it while backtracking, requires no maintenance. The performance benefit of this scheme is most prominent when a CNF instance contains many large original clauses or, as is more likely, the SAT solver adds large learned clauses as the search proceeds.

The second enhancement introduced in Chaff was the *Variable State Independent Decaying Sum*, or VSIDS, branching heuristic [37]. Unlike earlier branching heuristics, VSIDS was designed to leverage clause learning. Specifically, noting that difficult instances tend to generate many conflicts, and corresponding conflict-induced learned clauses, the heuristic leans towards choosing variables that occur with higher frequency in the most recent conflicts. Quoting from [37], VSIDS consists of the following five steps:

(1) Each variable in each polarity has a counter, initialized to 0.

(2) When a clause is added to the database, the counter associated with each literal in the clause is incremented.

(3) The (unassigned) variable and polarity with the highest counter is chosen at each decision.

(4) Ties are broken randomly by default, although this is configurable.

(5) *Periodically, all counters are divided by a constant.*

The adaptation of branching choices to conflict-driven learning has the additional benefit of low overhead since variable count statistics need to be updated only upon the occurrence of conflicts. This is in marked contrast to other branching heuristics, such as DLIS, which require much more frequent updating of literal counts.

The above narrative was not meant to be comprehensive. Other enhancements, such as learned clause minimization [48], learned clause deletion policies [25], search restart strategies [8, 46], formula simplification [18], and literal phase saving [40], etc., have been, and continue to be, suggested. However, it is fair to say that the four enhancements described in this section, namely a) GRASP-like conflict-driven clause learning, b) restarts, c) Chaff-like two-watched-literal propagation, and d) Chaff-like VSIDS branching, have had the most impact in improving the performance and capacity of modern SAT solvers. It is now standard to

refer to solvers that incorporate these features as conflict-driven clause learning, or CDCL, solvers.

3 MiniSat Configurations

MiniSat [19] is one of the best implementations of conflict-driven clause learning. It has a history of winning in SAT competitions, is open-source, and is particularly easy to modify and adapt for various uses. In this study we instrumented MiniSat 2.2.0 [5]to run in the twelve configurations listed in Table 1. The base configuration emulates the DPLL algorithm: it uses the DLIS branching heuristic, implements BCP with counters, and backtracks chronologically. The remaining eleven configurations are characterized by the on/off setting of the following four options:

- Option CL (Clause Learning): When this option is chosen, MiniSat performs conflict analysis, clause learning, and non-chronological backtracking; when CL is turned off, MiniSat reverts to DPLL-style chronological backtracking search.

- Option RST (Restarts): When this option is chosen, MiniSat applies a Luby restart strategy with a default cutoff of 100 conflicts (i.e., run lengths between restarts follow the sequence $100, 100, 200, 100, 100, 200, 400, \cdots$).

- Option 2WL (Two-Watched-Literals): When this option is chosen, MiniSat uses the watched-literals scheme for unit propagation; otherwise, it detects unit clauses by maintaining counters of 0 literals in each clause.

- Option VSIDS (Variable State Independent Decaying Sum): When this option is chosen, MiniSat applies the VSIDS branching heuristic; otherwise, it reverts to the DLIS heuristic.

  Since VSIDS relies on clause learning, it is not clear how to apply it when clause learning is disabled. Specifically, in the absence of clause learning, the application of VSIDS has no effect as all literal counts remain at their initial value (zero) causing branching decisions to become totally random. To retain the spirit of VSIDS in this case, we allow a limited form of clause learning, namely we allow conflict clauses to be created in order to update the literal counts that VSIDS depends on, but immediately delete these clauses to prevent them from helping avoid similar conflicts in the future. For comparison purposes, though, we have implemented the "crippled" version of VSIDS which will be referred to as VSIDS-.

[5] `http://minisat.se/Main.html`

Table 1: MiniSat Configurations

Configuration	CL	RST	2WL	VSIDS-	VSIDS
DPLL	N	N	N	N	N
CL	**Y**	N	N	N/A	N
RST	N	**Y**	N	N	N
2WL	N	N	**Y**	N	N
VSIDS-	N	N	N	**Y**	N
VSIDS	N	N	N	N	**Y**
¬ CL-	**N**	Y	Y	**Y**	N
¬ CL	**N**	Y	Y	N	**Y**
¬ RST	Y	**N**	Y	N/A	Y
¬ 2WL	Y	Y	**N**	N/A	Y
¬ VSIDS	Y	Y	Y	N/A	**N**
CDCL	Y	Y	Y	N/A	Y

Other than turning the above options on and off, we ran MiniSat with default settings for its other parameters, in particular, `opt_ccmin_mode = 2` for deep conflict clause minimization, `opt_phase_saving = 2` for full phase saving, and `opt_rnd_init_act = false` which sets initial literal activity to 0.

4 Benchmarks

To evaluate these twelve configurations of MiniSat we assembled a suite of 1000 CNF instances drawn from twelve diverse application domains. Table 2 lists the benchmark families along with the number of instances selected to represent each family. Columns SAT and UNS indicate, respectively, the number of satisfiable and unsatisfiable instances for each family, whereas column UNK indicates the number of instances whose satisfiability status is unknown[6].

The choice of these particular problem instances was based on a number of factors including:

- Representation of real-world problem domains where SAT had been successfully applied over the last decade and a half.

[6]The status of each instance was determined by consulting publicly-available data at various benchmark archives. We were unable to determine the status of only 28 instances and tagged them with UNK even though they may be known to be SAT or UNSAT.

Table 2: Benchmark Families

Family	Instances	SAT	UNS	UNK	Description
atpg	100	28	72	0	Circuit testing
bioinf	30	8	12	10	Bioinformatics
config	50	15	35	0	Product configuration
crypto	30	26	3	1	Cryptanalysis
equiv	30	5	25	0	Equvalence checking
fpga	50	25	22	3	FPGA routing
hbmc	250	88	146	16	Hardware bounded model checking
hverif	200	125	75	0	Hardware verification
netcfg	10	7	2	1	Network configuration
plan	80	51	24	5	Planning
sverif	120	57	52	11	Software verification
termrw	50	26	22	2	Term rewriting
Total:	1000	461	490	49	

- Representation of benchmark archives that are used to rank solvers in SAT Competitions[7] and SAT Races[8].
- Inclusion of a reasonable number of *easy* problem instances to enable all solver configurations to finish on at least some instances.
- Weighting the participation of each family (in terms of the number of instances representing it) by the relative success of applying SAT solving technology to that family in the recent past.

The atpg, plan, equiv and fpga families represent SAT applications dating from the early and mid 1990s [30, 44, 38]. Of these, atpg is the one domain where SAT solvers have had the most impact. The config family represents automative product configuration benchmarks [47], an area where SAT has been applied over the years [27]. Most of the remaining benchmarks were taken from the SAT competitions, again reflecting the relative impact of each practical application. The most sucessful applications of SAT include hardware bounded model checking (hbmc) [9], hardware verification (hverif) [52, 31], and software verification (sverif) [4]. More recent applications include network configuration problems (netcfg) [39], termination problems in term rewriting systems (termrw) [22], cryptanalysis (crypto) [35], and bioinformatics (bioinf) [10, 12]. Finally, it is

[7] http://www.satcompetition.org/.
[8] http://baldur.iti.uka.de/sat-race-2010/.

worth noting that we are not including in this study randomly-generated benchmarks a) because such benchmarks, especially random 3-SAT instances, have been studied extensively, particularly when the clause-to-variable ratio is approximately 4.25 (the so-called phase-transition region) [36], and b) because real-world benchmarks are rarely random. As we argue later, much work is needed to understand the structure of SAT instances that are derived from actual applications and how such structure affects the behavior of modern SAT solvers.

Figures 1 and 2 provide a variety of statistics that can be useful in differentiating among the benchmark families. Specifically, Figure 1 shows the distribution of the number of variables, number of clauses, and clause-to-variable ratio for the instances in each family. As can be seen, the instances cover a wide range with the smallest instance (50 variables and 159 clauses) coming from the `hbmc` family and the largest (2,270,930 variables and 8,901,845 clauses) representing the `netcfg` family. The clause size distribution (number of clauses of a given size) is diagrammed in Figure 2 and shows that while most instances consist of 2 and 3 literals, there is a sizable number of instances that consist of large clauses. Interestingly, the above-mentioned largest `netcfg` instance has a clause with 865 literals.

5 Evaluation

Before presenting and discussing the results of this study, a couple of caveats are in order. In general, to obtain statistically meaningful run time data requires that a SAT solver be applied to $n > 10$ randomly-ordered (both variables and clauses) versions of each CNF instance. Performing such an extensive evaluation was not feasible, however. Still, we believe that the data we collected provide useful indications of trends and can serve as motivation for further investigations. Secondly, we used a time-out limit of 1000 seconds, which is a little over 15 minutes. A longer time out, say one hour, could potentially, though not likely, affect the conclusions of the study.

The experiments were conducted on a cluster of servers at University College Dublin (UCD) consisting of x86_64 3GHz CPUs with 32GB memory and running the Linux operating system. A total of 12,000 separate runs were performed representing the execution of each of the 12 MiniSat configurations in Table 1 on each of the 1000 CNF instances in our benchmark suite. The results of these experiments are summarized in Figure 3 and Table 3.

The *cactus plot* in Figure 3 is a helpful way of comparing the performance of all twelve solver configurations. The curve corresponding to each configuration represents its run times for each of the 1000 benchmark instances. However, the

Table 3: Number of Instances Solved by Each MiniSat Configuration

Family	# Instances	DPLL	CL	RST	2WL	VSIDS-	VSIDS	¬ CL-	¬ CL	¬ RST	¬ 2WL	¬ VSIDS	CDCL
atpg	100	44	**100**	38	44	13	82	62	**100**	**100**	**100**	94	**100**
bioinf	30	1	4	0	1	0	6	0	7	14	9	3	**15**
config	50	49	**50**	49	49	0	7	2	50	**50**	**50**	29	**50**
crypto	30	0	0	0	0	4	8	7	6	**27**	12	7	26
equiv	30	0	10	1	0	0	5	5	5	**23**	19	15	**23**
fpga	50	25	42	24	25	1	25	12	49	45	44	28	**47**
hbmc	250	40	187	33	40	18	107	71	127	231	227	164	**234**
hverif	200	22	167	21	22	5	119	142	150	197	194	183	**198**
netcfg	10	0	2	0	0	0	2	3	0	7	6	**10**	9
plan	80	17	44	13	17	7	32	26	48	57	57	**65**	**65**
sverif	120	5	66	6	5	6	34	22	73	73	92	82	**100**
termrw	50	1	24	0	1	1	24	12	22	**45**	37	34	43
Totals	1000	204	696	185	204	55	451	364	637	869	847	714	910

data points for each configuration are sorted separately from smallest to largest[9]. In other words, the ordering of the instances along the x-axis is different for different configurations.

Careful study of the data in the table and figure suggests the following conclusions:

1. CDCL, the configuration that includes all four features discussed in this article, provides the best overall performance. It solves 910 out of the 1000 instances within the 1000 second time-out limit.
2. The configurations that represent the absence of a feature suggest that, in terms of effectiveness, the features should be ordered according to CL > VSIDS > 2WL > RST. In other words, fewer instances are solved if CL is missing (637) than if VSIDS is missing (714), etc. Conversely, more instances are solved if that same feature is the only option used by the solver: 696 for CL, 451 for VSIDS, 204 for 2WL, and 185 for RST.
3. The "crippled" VSIDS- option performs extremely poorly, completing on just 55 instances. This is not surprising since, as mentioned earlier, in the absence of clause learning VSIDS degenerates into a random branching heuristic. Option ¬CL− which adds RST and 2WL to VSIDS- improves performance considerably allowing the solution of 364 instances. A possible explanation of this result is that RST is able to overcome the bad branching choices of VSIDS-. This is further corroborated by noting that the average number of restarts under ¬CL− (random branching) is about ten times the average number of restarts under RST (DLIS branching).
4. The RST configuration performs worse than DPLL. This seemed to be anomalous until we realized that the combination of no learning and DLIS branching causes the same decision sequence to be retraced on every restart. Effectively, then, in the RST mode MiniSat is repeatedly executing DPLL until it reaches the conflict cutoff. This also confirms the need to randomize the decision sequence for restarts to be useful. Such randomization occurs naturally when clause learning is enabled since the addition of clauses changes literal activity causing the solver to follow a different decision order on each restart.
5. The performance of 2WL was suspiciously identical to that of DPLL. We expected that 2WL would show some advantage for those instances with large clauses. It turned out, however, that 2WL timed out on those instances and its performance edge on instances with short clauses was not significant enough to outperform counter-based unit propagation. As with search restarts, 2WL is not effective per se, since most problem instances have mostly short clauses.

[9]The data points for instances that timed out are not shown to keep the figure from becoming too cluttered.

However, in the presence of clause learning, the effect of 2WL becomes significant, due to the addition of many large learned clauses.

6. The most significant difference between VSIDS and DLIS is the associated overhead. DLIS requires all literal counters to be updated after unit propagation and after backtracking. Data from the late 1990s indicated that DLIS could account for more than 75% of the run time of GRASP [32]. In contrast, VSIDS has very low overhead. This results in most of the time being spent in the actual search, and so allows solving more problem instances.
7. The most difficult instances come from the `bioinf` family; only 15 out of the 30 instances were solved. Next are the instances from the `equiv` family (23 out of 30 solved), the `plan` family (65 out of 80 solved), and the `sverif` family (100 out of 120 solved). The instances from the remaining eight families are, comparatively, much easier: all instances from the `atpg`, `config`, and `netcfg` families were solved, nearly all instances from the `fpga` and `hverif` families were solved, and over 90% of the instances from the `crypto`, `hbmc`, and `termrw` were solved.
8. An interesting byproduct of this study is that 28 out of the 49 UNK instances were solved. Ironically, 25 of these were solved by the ¬ VSIDS configuration: 14 from the `hbmc` family, 4 each from the `plan` and `sverif` families, and 1 each from the `crypto`, `netcfg`, and `termrw` families. The other 3, all from the `fpga` family were solved by the ¬ CL configuration. It is unclear why disabling VSIDS and CL was helpful in these cases.

Additional insights can be gleaned by examining the run time data for each benchmark family separately. Cactus plots for each of the twelve benchmark families are provided in the appendix.

6 Conclusion

The impressive progress in the capacity and performance of SAT solvers over the last fifteen years has accelerated their adoption as indispensable reasoning engines, particularly in hardware and software verification applications. In a recent paper, SAT-based formal methods were reported to have been successfully used as the primary validation vehicle for the Intel Core i7 Processor Execution Engine [28]. In this article we tried to highlight what we think were the primary algorithmic and implementation advances in SAT solving technology that made this possible. But lingering questions remain. With few exceptions, very little has been done theoretically to *explain* why CDCL solvers work so well. In [7], Beame et al. show that, as a proof system, clause learning is more powerful than regular and DP resolution. Recent work has also shown that a proof system based on clause learning with restarts is as powerful as general resolution (e.g. [41]).

Perhaps an easier question to answer is why CDCL works so well on certain types of problem instances and not so well on others (see observation 8 in Section 5). Studies that characterize the *structure* of CNF instances can help guide the answers to such questions. For example, intractability may be caused by symmetry and approaches that break symmetry have shown some promise [2]. Other structural attributes such as the cut width of graph representations of CNF instances have been proposed as useful metrics [11]. A recently-suggested promising direction is based on "discovering" that industrial SAT instances have a *scale-free* graph structure [3] and that this structure seems to be maintained throughout the search process as assignments are made and erased and as learned clauses are added.

In summary, the experimentalists have done quite well in the last few years. What we need going forward is a little bit more *theory* to explain what has already been done and, perhaps more importantly, to guide the future course of SAT research.

Acknowledgment.

This work was partially supported by SFI grant BEACON (09/PI/I2618) and by the United States National Science Foundation under Grant No. 0705103. The authors wish to also extend their sincere appreciation to Hadi Katebi for his tireless efforts in instrumenting MiniSat and using it to collect and organize the experimental results.

References

[1] M. L. Alistair, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[2] F. Aloul, K. Sakallah, and I. Markov. Efficient symmetry breaking for boolean satisfiability. *Computers, IEEE Transactions on*, 55(5):549 – 558, May 2006.

[3] C. Ansótegui, M. L. Bonet, and J. Levy. On the structure of industrial sat instances. In *CP*, pages 127–141, 2009.

[4] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *International Conference on Software Engineering*, pages 211–220, 2008.

[5] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 489–494, September 2000.

[6] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[7] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[8] A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 28–33, 2008.

[9] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Advances in Computers*, chapter Bounded Model Checking. Academic Press, 2003.

[10] M. L. Bonet and K. S. John. Efficiently calculating evolutionary tree measures using SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 4–17, 2009.

[11] E. Broering and S. V. Lokam. Width-based algorithms for sat and circuit-sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 162–171, 2003.

[12] F. Corblin, L. Bordeaux, E. Fanchon, Y. Hamadi, and L. Trilling. Connections and integration with SAT solvers: A survey and a case study in computational biology. In *Hybrid Optimization: the 10 years of CPAIOR*. Springer, 2010.

[13] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *AAAI*, pages 21–27, 1993.

[14] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.

[15] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, July 1960.

[16] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.

[17] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. Sat versus unsat. In D. S. Johnson and M. A. Trick, editors, *Cliques, coloring, and satisfiaability*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26, pages 415–436. 1993.

[18] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *International Conference in Theory and Applications of Satisfiability Testing*, pages 61–75, 2005.

[19] N. Eén and N. Sörensson. An extensible SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing*, May 2003.

[20] S. E. Fahlman. A planning system for robot construction tasks. Master's thesis, Massachusetts Institute of Technology, 1973.

[21] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1995. UMI Order No. GAX95-32175.

[22] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Sat solving for termination analysis with polynomial interpretations. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 340–354, 2007.

[23] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *Computers, IEEE Transactions on*, C-32(12):1137 –1144, December 1983.

[24] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, 4(1):28 –35, January 1960.

[25] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Design, Automation and Testing in Europe Conference*, pages 142–149, March 2002.

[26] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *National Conference on Artificial Intelligence*, pages 431–437, July 1998.

[27] M. Janota. Do sat solvers make good configurators? In *SPLC (2)*, pages 191–195, 2008.

[28] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in Intel CoreTM i7 processor execution engine validation. In *International Conference on Computer Aided Verification*, pages 414–429, 2009.

[29] S. Kim and S. K. Hantao. Modgen: Theorem proving by model generation. In *in Proc. National Conference of American Association on Artificial Intelligence (AAAI-94*, pages 162–167, 1994.

[30] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, January 1992.

[31] P. Manolios and S. K. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 363–366, 2005.

[32] J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the Portuguese Conference on Artificial Intelligence*, pages 62–74, September 1999.

[33] J. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[34] J. Marques-Silva and K. A. Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[35] I. Mironov and L. Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 102–115, 2006.

[36] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *National Conference on Artificial Intelligence*, pages 459–465, 1992.

[37] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, June 2001.

[38] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: Detailed routing of complex FPGA s via search-based boolean SAT. In *International Symposium on Field-Programmable Gate Arrays*, February 1999.

[39] S. Narain. Network configuration management via model finding. In *Conference on Systems Administration*, pages 155–168, 2005.

[40] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *International Conference in Theory and Applications of Satisfiability Testing*, pages 294–299, 2007.

[41] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers with restarts. In *International Conference on Principles and Practice of Constraint Programming*, pages 654–668, 2009.

[42] D. Pretolani. Efficiency and stability of hypergraph sat algorithms. In D. S. Johnson and M. A. Trick, editors, *Cliques, coloring, and satisfiaability*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26, page 479Ű498. 1993.

[43] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[44] B. Selman and H. Kautz. Planning as satisfiability. In *European Conference on Artificial Intelligence*, pages 359–363, 1992.

[45] E. H. Shortliffe. *Computer-based medical consultations, MYCIN / Edward Hance Shortliffe*. Elsevier, New York :, 1976.

[46] C. Sinz and M. Iser. Problem-sensitive restart heuristics for the DPLL procedure. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 356–362, 2009.

[47] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.

[48] N. Sörensson and A. Biere. Minimizing learned clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243, 2009.

[49] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[50] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(9):1167 –1176, September 1996.

[51] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. A. Trick, editors, *Cliques, coloring, and satisfiaability*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26, page 559Ű586. 1993.

[52] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *J. Symb. Comput.*, 35(2):73–106, 2003.

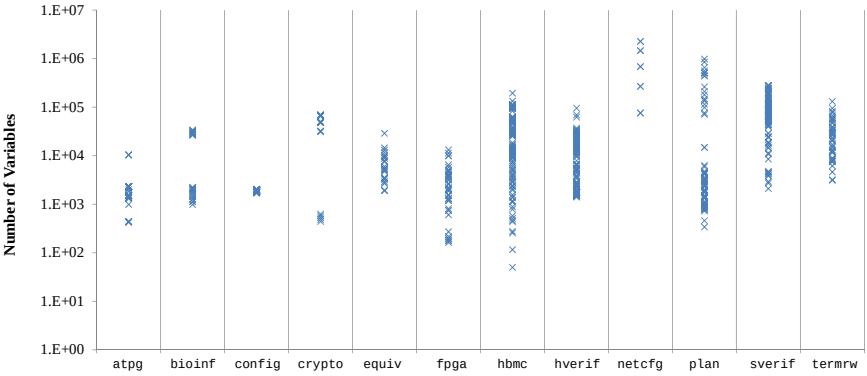
[53] H. Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4(1):2 –22, January 1960.

[54] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *National Conference on Artificial Intelligence*, pages 155–160, July 1988.

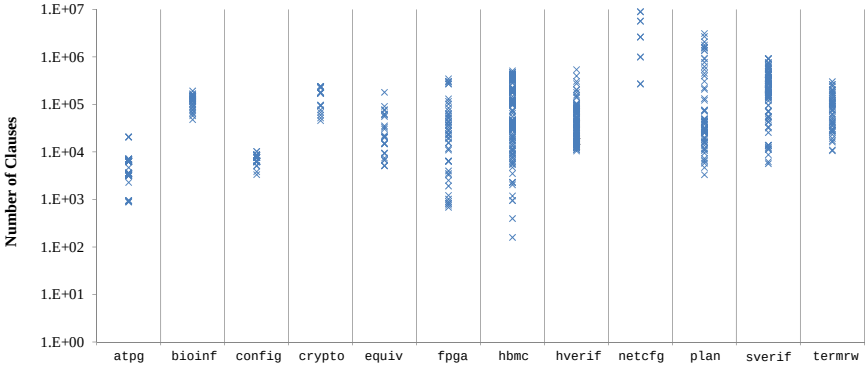
[55] H. Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, July 1997.

A Detailed Run Time Distributions

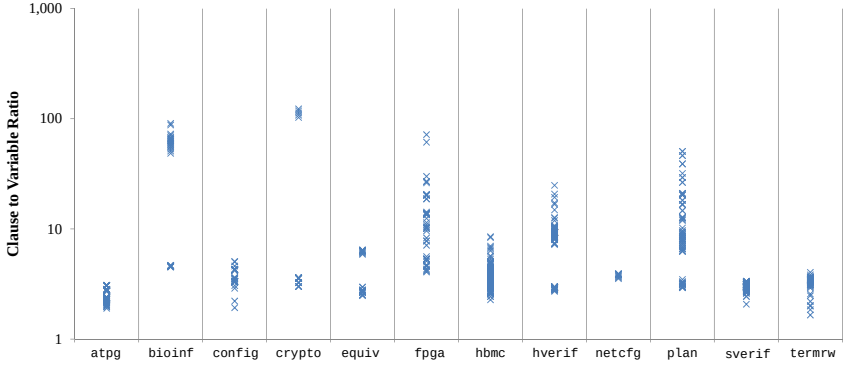
Figures 4(a) through 4(l) show, separately, the run time distributions of the twelve MiniSat configurations for each of the benchmark families.



(a) Number of variables



(b) Number of Clauses



(c) Clause-to-Variable Ratio

Figure 1: Benchmark Statistics

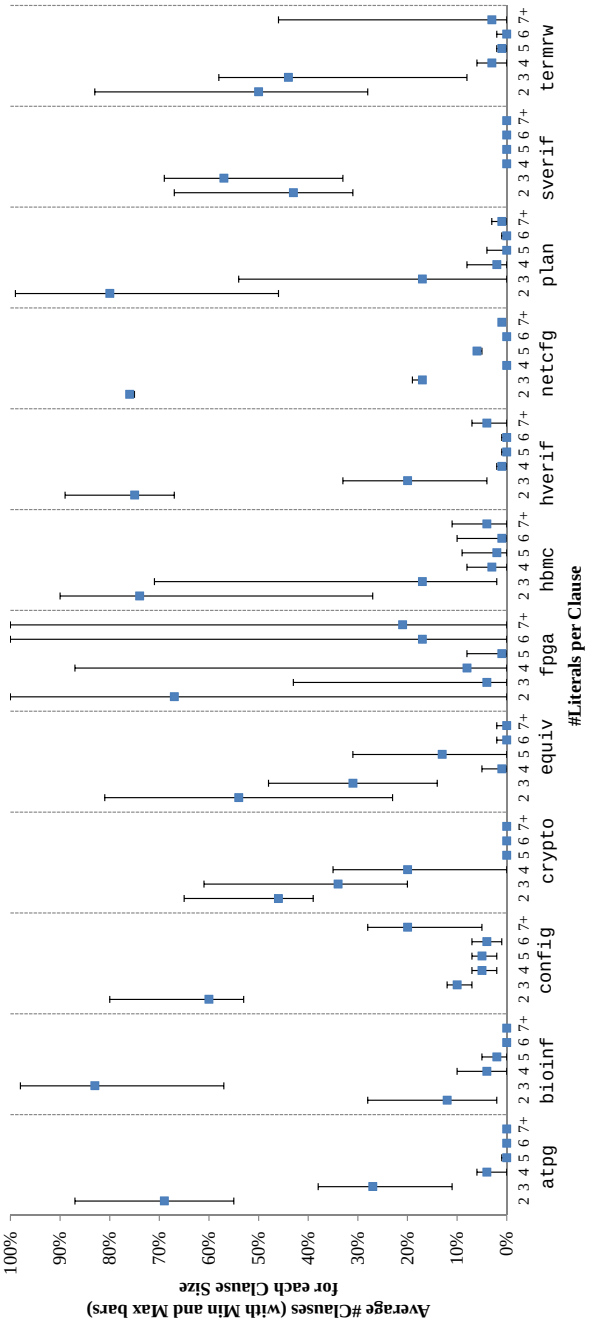


Figure 2: Clause Size Distribution in Benchmark Families

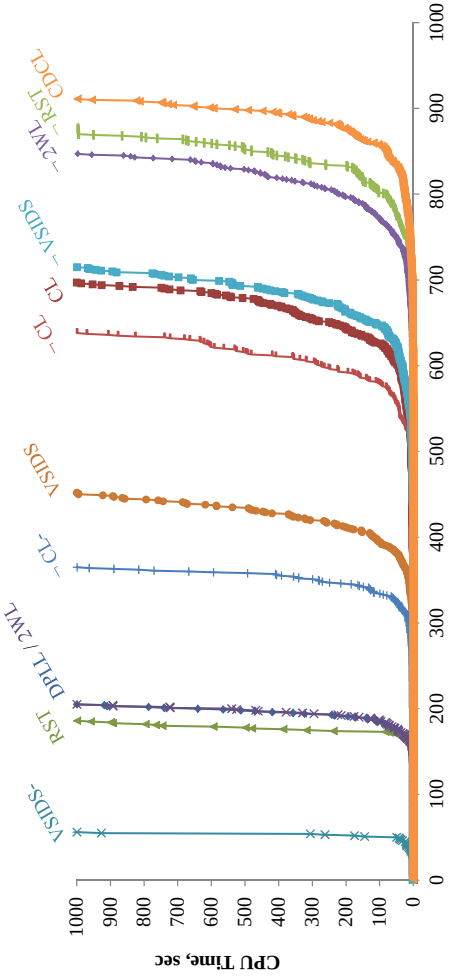
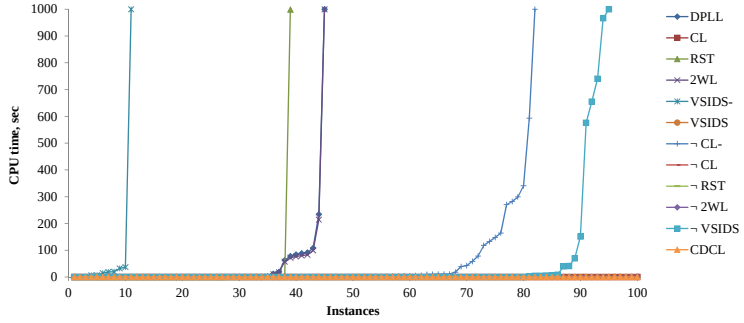
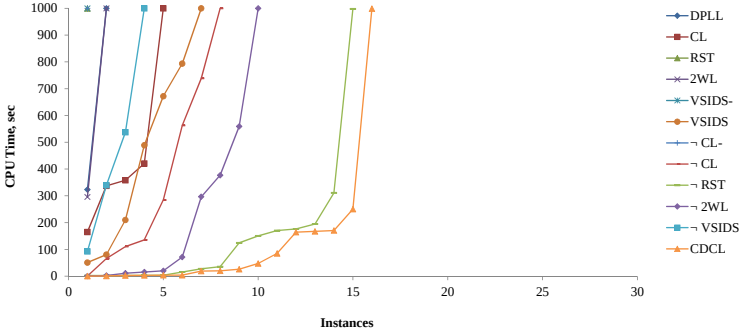


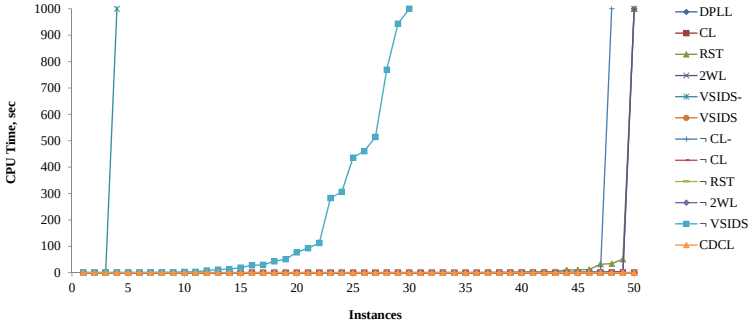
Figure 3: Run Time Distributions



(a) Run Time Distribution for the `atpg` Family

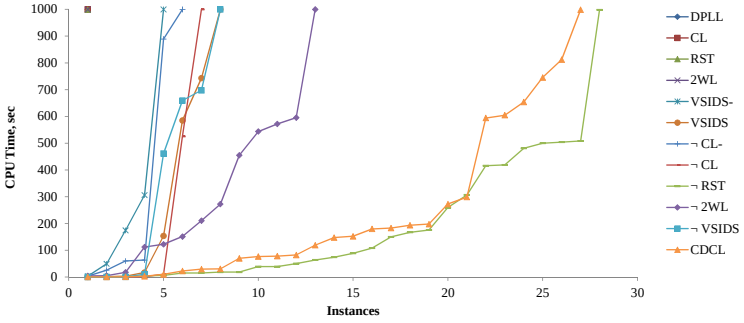


(b) Run Time Distribution for the `bioinf` Family

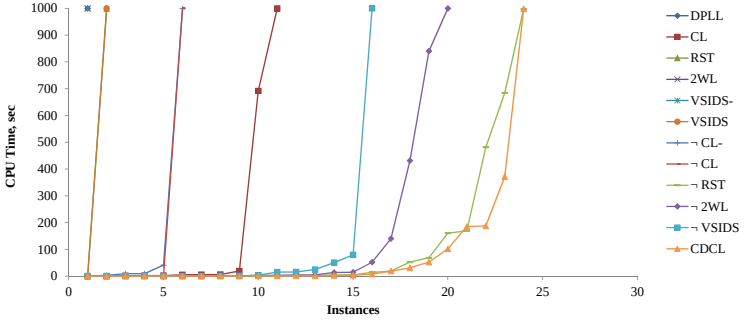


(c) Run Time Distribution for the `config` Family

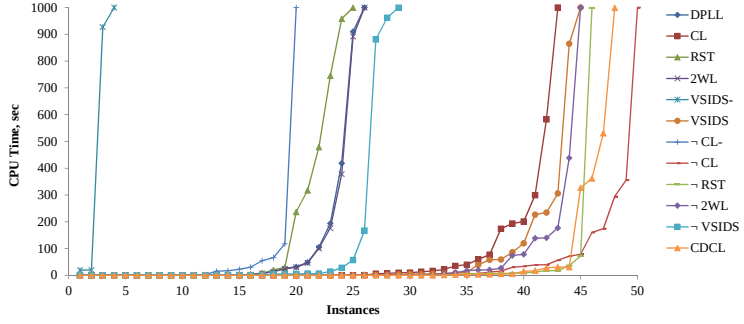
Figure 4: Detailed Run Time Distributions



(d) Run Time Distribution for the `crypto` Family

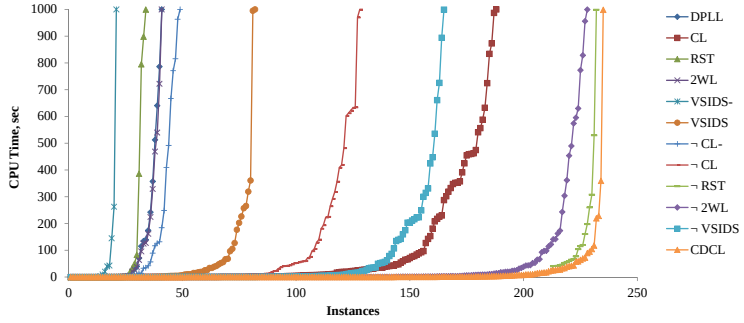


(e) Run Time Distribution for the `equiv` Family

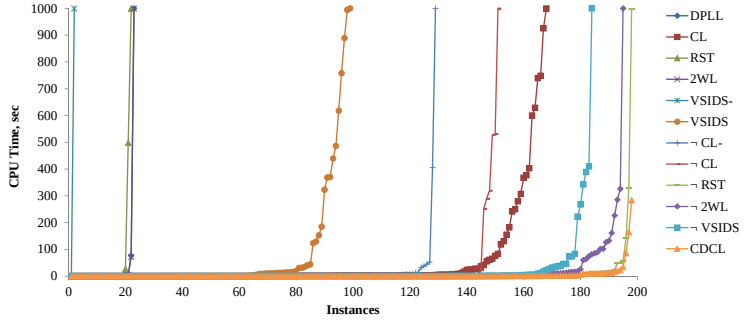


(f) Run Time Distribution for the `fpga` Family

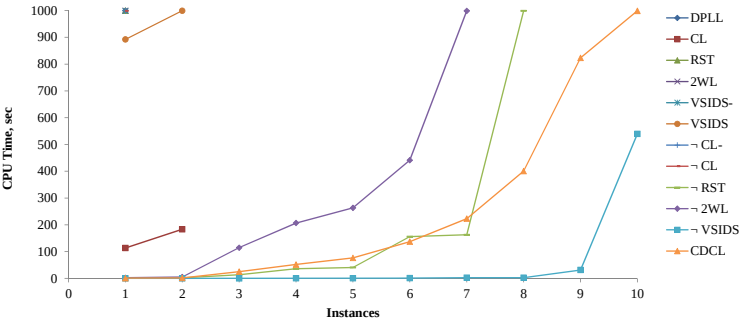
Figure 4: Detailed Run Time Distributions (cont.)



(g) Run Time Distribution for the `hbmc` Family

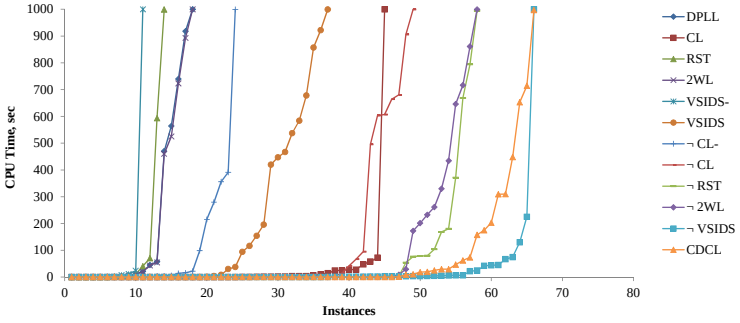


(h) Run Time Distribution for the `hverif` Family

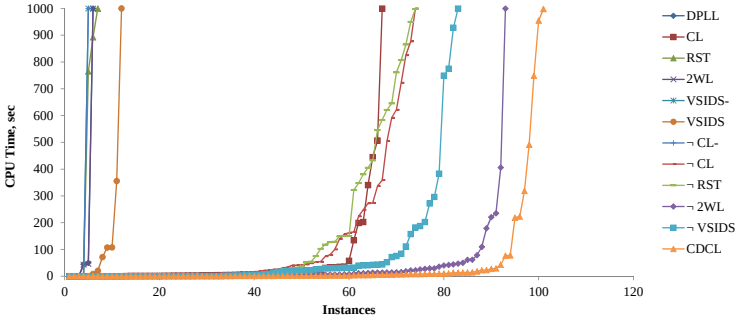


(i) Run Time Distribution for the `netcfg` Family

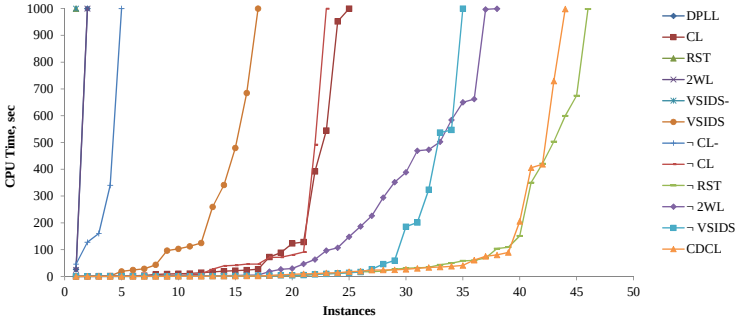
Figure 4: Detailed Run Time Distributions (cont.)



(j) Run Time Distribution for the `plan` Family



(k) Run Time Distribution for the `sverif` Family



(l) Run Time Distribution for the `termrw` Family

Figure 4: Detailed Run Time Distributions (cont.)