Extreme SAT-based Constraint Solving with R-Solve

James Ezick, Jonathan Springer, Tom Henretty
Reservoir Labs, Inc.
New York, NY USA
{ezick, springer, henretty}@reservoir.com

Chanseok Oh
Department of Computer Science
New York University
New York, NY USA
chanseok@cs.nyu.edu

***Abstract*—We present the architecture of *R-Solve*, a high-performance constraint solver that extends the gains made in SAT performance over the past fifteen years on static decision problems to problems that require on-the-fly adaptation, solution space exploration and optimization. R-Solve brings together a modern, open SAT solver, HPC ideas in ensemble and collaborative parallel solving and a novel set of extensions supporting, for the first time, an efficient system for unrestricted incremental solving. These extensions, which we call *Smart Repair*, enable R-Solve to address problems in dynamic planning and constrained optimization involving complex logical and arithmetic constraints. We provide insight into the rationale and challenges of extending an aggressively optimized competitive solver to support a new class of problem, provide a performance milestone illustrating the potential of R-Solve and Smart Repair and discuss new frontiers for further HPC research in SAT-based constraint solving and constrained optimization.**

Keywords—constraint solving; parallel SAT solving; unrestricted incremental solving; loosely cooperative parallelism; dynamic planning; constrained optimization; multi-objective optimization

I. INTRODUCTION

Since 2000, the algorithmic performance of single-node sequential solvers for the Boolean Satisfiability (SAT) problem has improved by more than 1000x [1, 2]. While the SAT problem has worst-case NP-complete complexity, most structured problems do not exhibit anywhere near worst case complexity and modern solvers have evolved into highly successful tools for solving a range of practical problems including resource allocation, planning, and hardware and software verification [1]. As a practical tool, SAT solvers fill a niche in that they can efficiently reason within a single framework about complex logical constraints that capture concepts such as coordination, deconfliction and mutual exclusion and fixed-point arithmetic constraints that can include non-linear expressions.

As a practical tool, open off-the-shelf SAT solvers suffer from two key limitations.

1. Each SAT problem instance input is inherently limited to a static SAT-or-UNSAT decision problem.

A generic modern SAT solver, given a logical formula over a set of Boolean variables (typically a conjunction of constraints), will either return a result of SAT along with a single example assignment to the variables or else a result of UNSAT. If a second example SAT assignment is required or if the logical expression is modified, the solver must be restarted from the beginning with no retained information. While specialized solvers exist that can count the number of satisfying assignments [3] or that provide a limited (addition of refinement constraints only) incremental solving capability [4], these solvers do not fully address this limitation in a practical sense and are not typically at the leading edge of performance.

2. Solvers at the leading edge of performance are aggressively optimized and notoriously difficult to either parallelize or extend.

Modern solvers accept logical expressions in Conjunctive Normal Form (CNF) and use custom data structures to access and manage clauses (disjunctions of literals) affected by a variable assignment [5]. In addition, one of the key optimizations in modern SAT solving is to use a system for generating new learned clauses that capture ground truths about the problem instance and concisely prune the search tree of possibly valid assignments [6]. Although the propagation of logical implications, called Boolean Constraint Propagation (BCP), from test assignments can be parallelized in theory, the data structures and learning required to achieve top performance result in extreme non-locality of memory references and difficult synchronization requirements that have frustrated most attempts at effective fine-grained parallelism [7, 8]. Further, aggressive optimization tends to make the implementation fragile to modification or the addition of state to support extended features.

In this paper we introduce an architecture for a fully unrestricted incremental SAT solver, *R-Solve*, that addresses both of the key limitations described in this section. R-Solve is an "extreme" SAT solver in that it combines the new technology of *Smart Repair*, which enables unrestricted incremental solving (unlimited addition or subtraction of constraints during or after the execution of the solver), with techniques in ensemble solving and clause sharing for MPI-connected clusters while retaining (modulo a manageable amount of communication overhead) the performance characteristics of a leading-edge open solver. Smart Repair is implemented as an API that can be applied to any of several open solvers and R-Solve provides a platform for new applications of SAT, including optimization and dynamic replanning problems, and opens new frontiers for improving performance for practical problems on HPC platforms.

The remainder of this paper is organized as follows. Section II provides an overview of the concept behind Smart

This work was funded in part by the Defense Advanced Research Projects Agency (DARPA) Architectures for Cognitive Information Processing (ACIP) program under Army SBIR contract W91CRB-11-C-0087.

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

Repair as a general mechanism for implementing unrestricted incremental solving and explains how it extends SAT to new application domains. Section III briefly describes R-Solve's approach to parallelism. The implementation challenges of bringing together Smart Repair and R-Solve's system of loosely cooperative parallelism are addressed in Section IV. Section V presents initial performance results and Section VI concludes with directions for future work.

II. SMART REPAIR

Smart Repair is an extension to the dominantly used Conflict-Driven Clause Learning (CDCL) method of SAT solving [6] enabling efficient unrestricted incremental solving through the discrimination and retention of unaffected learned information across a modification of the formula.

A. Foundations of Modern CDCL SAT Solving

The baseline CDCL algorithm assumes a problem formula presented in CNF as a set of original clauses and executes according to the following high-level loop:

1. An unassigned variable is chosen for assignment (decision assignment: *true* or *false*).
2. The logical implications of that assignment are discovered through BCP. These implications become additional assignments which are then also propagated serially using a work queue.
3. The process terminates when either (1) all variables are assigned (problem is solved), (2) there are no more assignments to propagate (process repeats with a new decision assignment), or (3) a conflict is discovered (a variable is assigned both *true* and *false*).

Assignments in this process occur serially and are referred to as the *assignment stack*. The decision assignments in the stack partition the stack into *decision levels*. Assignments that are discovered to be absolute are said to be *decision-level zero* (DL 0). The first decision marks the beginning of DL 1. Each decision level then includes the Step 1 decision assignment plus all of the BCP-implied assignments discovered from that assignment. If BCP neither solves the problem nor ends in a conflict, a return to Step 1 marks a new decision level.

In the case of a conflict, a sub-process referred to as *conflict analysis* is invoked that has the following effect:

1. The assignment stack is rolled back to some earlier decision level (one or more levels) undoing assignments sufficient to resolve and avoid reoccurrence of the root conflict.
2. A learned conflict clause is generated. These clauses are logical implications of the original and previously learned clauses and are intended to prune the search tree by capturing high-value information. All clauses, both original and learned, are stored in a data structure referred to as the *clause database*. Learned unit (single-literal) clauses are also added to DL 0. The conflict clause generation process may also add an assignment to the new top-of-stack decision level as a logical implication of the clause being generated. This assignment is then propagated by the normal BCP process. This round of BCP is treated as an extension of Step 2 of the high-level loop. Any new conflict would be resolved as described by Step 3.

A conflict caused by DL 0 assignments results in a return of UNSAT. The decision heuristic used to select a variable in Step 1 and the workings of the BCP and conflict analysis processes are not relevant to the concept of Smart Repair.

B. Enabling Efficient Unrestricted Incremental SAT Solving

The key observation from the basic description of CDCL is that each learned clause is derived from one or more existing clauses. In this way, the clause database can be considered a Directed Acyclic Graph (DAG) rooted in the original clauses. When one or more original clauses is eliminated, any clause reachable in the DAG representation of the clause database loses its logical support and must be pruned. When the clause database is modified either through the addition or subtraction of clauses, the solver is rolled back to DL 0 and the previous assignment stack is replayed as a sequence of decisions until it is exhausted or a conflict is found. This quickly rebuilds the internal data structures of the solver in a directed and consistent way. Note that rolling back the assignment stack has no impact on the clause database – learned clauses are not tied to a decision level and are strictly the logical consequence of their antecedent (predecessor) clauses in the DAG.

While the necessity to manage learned clauses as a prerequisite for implementing unrestricted incremental solving has been understood for some time [9], R-Solve is the first solver to overcome the performance barriers necessary to implement this capability in a state-of-the-art solver [10]. To do this, R-Solve leverages an MPI-connected multi-node architecture in which the DAG management is offloaded to a separate process. This master process then serves as a bridge with an outward-facing API to the user and an inward-facing API to the solver. Integrating the management of the Smart Repair DAG with modern optimizations to the core CDCL algorithm and the needs of long-running applications required additional design features which are covered in Section IV.

C. Applications of Smart Repair

Smart Repair is a fully general mechanism that extends the normal capability of a SAT-solver so that it may address several additional real-world application domains. Examples of problem domains that benefit from the unrestricted incremental solving capability provided by Smart Repair include:

- **Planning in Dynamic Environments**

Smart Repair permits the direct capture and incorporation of dynamic changes to the environment model of a constrained planning problem. Modification scenarios that are completely supported with Smart Repair that were at most only partially supported with prior restricted incremental solving include:

- Changes in the availability of a resource or asset
- Emergence of new, or elimination of prior, obstacles or rules for employing assets
- Changes in goal conditions or value of the goals

Without Smart Repair, adaptation to any of these conditions would require abandoning all prior work and restarting with a new solver invocation.

- **Planning with a Human in the Loop**

In many practical applications, the solution to a planning problem rendered by a SAT solver is subject to review by a human before a commitment to execute is made. Similar to planning in a dynamic environment, Smart Repair allows a human operator to either introduce new, or relax existing, constraints for the purpose of driving to a new, more suitable solution that replaces an existing solution in part or in whole.

- **Iterating through a Solution Space**

In native operation, a SAT solver returns a single solution to a constraint system. In many applications, such as model checking, each satisfying assignment decodes into a unique example of interest. Smart Repair (as with prior restricted incremental solving) enables solution space iteration by permitting re-solving following the addition of constraints that specifically preclude the solution(s) already found.

- **Optimizing over Complex Constraints**

SAT is an NP-complete problem. Solved repetitively for different fitness values, k, Smart Repair can produce provably optimal solutions for NP-hard k-optimization problems using strategic (non-sequential) search over the fitness space. In this method, the constraints encoding the fitness function are isolated and modified for each attempted k and the problem is re-solved producing tighter solutions until the fitness value is found at which the problem transitions from SAT to UNSAT. The last fitness value returning SAT is provably optimal.

The addition of this capability means that SAT techniques can be applied to optimization problems with complex coordination, deconfliction and mutual-exclusion constraints (all of which have natural representations in Boolean logic) and fixed-point arithmetic constraints (including non-linear constraints) which encode as sequential circuits and thus translate to SAT. All complex optimization problems are essentially search problems where the solution algorithm relies on some underlying assumptions or substructure – locality for hill-climbing methods, unrestricted matching for auction algorithms, the utility of combining parts of prior solutions for evolutionary algorithms, etc. SAT-based optimization is driven by a completely different set of substructure assumptions, namely a logical substructure expressed in the problem model. The addition of Smart Repair represents a new tool for addressing optimization problems that are not a good fit to the substructure assumptions exploited by other methods – problems with pervasive discontinuities or non-linearity in the fitness landscape, problems with deep logical constraints and problems with large infeasible solution regions.

III. Loosely Cooperative Parallelism

The CDCL algorithm is notoriously difficult to effectively parallelize. The algorithm is dominated by the BCP routine (80-90% of runtime for most implementations), which exhibits extreme non-locality of memory references into the clause database. Further, while implied assignments are stored on a work queue, which can, in theory, be parallelized, in practice fine-grained parallelism has generally proven incompatible with the data structure used to access clauses for implication testing. This data structure is not read-only and requires modifications, in the form of pointer updates and literal swaps within each clause, which create difficult synchronization barriers. Despite attempts to smartly cluster the clauses, distributing the database has generally proven ineffective.

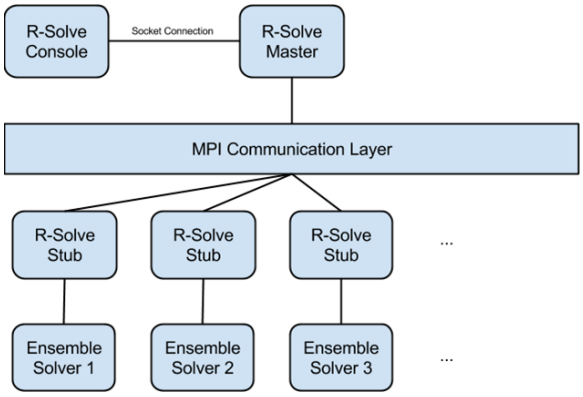

Figure 1. Implementation of Smart Repair as a master process communicating with solvers sharing clauses.

An alternate approach, parallelism by partial assignment [11], where each process picks up from a different fixed assignment to a subset of the variables, creates difficult load-balance issues. The lifetime of each assignment stack prefix is difficult to predict. Further, information from one process cannot generally be used by any other process since each is running from a different set of initial assignment assumptions.

R-Solve builds on a system of loosely cooperative parallelism that has become common in the field [12]. The CDCL algorithm contains a number of tunable tactics and random seeds that modify how the decision heuristic and conflict analysis processes operate. These choices form a space over which a solver can be configured. Differently configured solvers naturally drive to different parts of the search space. In experiments, it is not uncommon over 100 runs for the best solver to perform 5-25x faster than the median solver. Speed-up is achieved by having the solvers independently race to a solution. An advantage of this approach is that clauses learned by one solver can be shared among the other solvers to amplify the speedup. This is what we mean by loosely cooperative parallelism – the independent solvers broadcast especially high-value clauses (in particular, unit and binary clauses) and the receiving solvers check and incorporate the new clauses on a periodic basis. Shared clauses that conflict with the local assignment stack trigger a rollback and replay operation that is a special case of the routine used when clauses are incrementally updated in Smart Repair.

In R-Solve, the clause sharing infrastructure leverages the Smart Repair infrastructure – the master process managing the global Smart Repair clause implication DAG serves multiple solvers and provides a switchboard for detecting duplicate clauses and making rational choices about which clauses to share. This design is illustrated in Figure 1.

IV. R-SOLVE IMPLEMENTATION

R-Solve is built on Glucose 2.1, an open, publically-available CDCL SAT solver [13]. Glucose is a competitive solver that has ranked at or near the top of both SAT and UNSAT application categories in competitions from 2009, 2011, 2012 and 2013. In addition to the standard CDCL algorithm, Glucose derives its performance from specific optimizations in constraint preprocessing, variable elimination, its decision heuristic, BCP work queue ordering, clause learning, clause reduction and learned clause attrition. As made available for download, Glucose is a strictly single-node, sequential solver developed for execution on desktop hardware. As with all competitive solvers, Glucose is a command-line tool that accepts Boolean formulas in the standard DIMACS CNF format. Glucose is written in C++ and comprises approximately 5000 lines of source code.

The remainder of this section highlights several of the design challenges that were addressed in transforming Glucose into the core of R-Solve. Emphasis is given to describing how the high-level concepts of Smart Repair and loosely cooperative parallelism described in Sections II and III, respectively, had to be extended to support the complete set of highly-specialized optimizations that has made Glucose one of world's highest-performing open SAT solvers.

A. R-Solve Console

The user interface for R-Solve is an interactive console that can run on a host PC. The console connects to the master process through a UNIX socket and communicates using Google Protocol Buffer [14] messages. The console provides a robust set of commands to load, solve, incrementally update and test the solution to a problem instance and to manage the configuration and shutdown of the solver ensemble. The console supports both blocking and non-blocking modes of solving, the non-blocking mode being necessary so that constraint updates can be passed to the solver while the solver is running. For testing and benchmarking, the console accepts problems in the standard CNF format, but understands stylized comments that delimit blocks of clauses that can be independently activated or deactivated. Each block is labeled and intuitively captures one or more high-level constraints.

B. R-Solve Ensemble API

The R-Solve ensemble is an MPI-connected collection of solver processes controlled by a master process. The master process maintains the Smart Repair DAG, manages the loosely cooperative parallelism and communicates commands to a solver-side stub. The stub then presents to the solver the R-Solve ensemble API that provides a set of operations and data formats that the solver must utilize to participate in the ensemble. This API includes 20 routines for accepting a SAT instance, communicating learned clauses, accepting shared clauses and incremental updates, responding to requests for configuration changes and status queries, restarting and shutting down. The expectation is that any CDCL solver should be adaptable for participation in an R-Solve ensemble provided that the necessary stub API calls and responses are integrated into the solver. In adapting Glucose, ~1000 lines of code were touched. The goal of the stub and API architecture is to compartmentalize the modifications that must be made to the core solver while eliminating the need to re-implement low-level details such as message buffering and synchronization. For testing and benchmarking, the ensemble is configured using a single file that maps an expression with a free variable for the rank of the solver process to each solver option. Independent evaluation of the expressions yields unique initial configuration settings for each solver in the ensemble.

C. Support for Variable Elimination

Variable elimination [15] is a significant preprocessing optimization performed by Glucose and other leading SAT solvers to remove infrequently used variables before CDCL solving begins. Given the set of clauses where a variable **X** occurs in the positive phase **(X l_i)** and the set of clauses where **X** occurs in the negative phase **(~X m_j)**, it can be shown that the Cartesian product set of clauses **(l_i m_j)** is equisatisfiable to the original set. The assignments to variables occurring in the **l_i** and **m_j** expressions dictate the assignment to **X** in a post-processing step. Because of the Cartesian product operation, variable elimination cannot be used on variables that occur frequently in both phases. In R-Solve, variable elimination must be a persistent operation in order to support incoming updated and shared clauses. Persistence requires reapplication of variable elimination (in the order in which the variables were originally eliminated) and, when a frequency threshold is crossed, the undoing of the variable elimination for that variable. R-Solve supports this persistent variable elimination capability through the solver stub mechanism.

D. Support for Clause Modification

Aside from variable elimination, modern solvers perform other periodic clause simplification optimizations that rewrite existing clauses. For example, once a variable is assigned at DL 0, that variable may be removed from clauses that contain that variable in the opposite phase (an empty clause indicates UNSAT). In these cases, the rewritten clause is the logical consequence of existing clauses. However, if an antecedent of the rewritten clause is ever eliminated, it is necessary for correctness to restore the prior form of the clause. To do this, Smart Repair supports undo edges that point from the rewritten clause to the prior representation. If a Smart Repair operation removes the rewritten clause (but not the prior), then the undo edge triggers a restore operation in which the prior clause is added to the set of clauses each solver must reincorporate.

E. Optimizations for Clause Sharing

It is common for cooperating solvers to learn identical clauses. The master process uses a hashing scheme based on the number and identity of the literals in each learned clause to identify duplicates. Duplicate clauses are then not repetitively shared. However, the Smart Repair DAG retains duplicates with distinct antecedent sets. This increases the likelihood that at least one copy will be unaffected by an update.

F. Support for Graph Reduction

While the individual solvers perform periodic attrition of seldom used or low-value clauses to manage the size of their

local clause database, the master process cannot prune arbitrary nodes from the clause DAG while retaining the reachability criteria without the potential for cross-product edge explosion. While the solvers do communicate removed learned clauses to the master process and while those clause nodes are removed when they have out-degree of 0 or 1, there is still a need to enforce a hard upper limit on the size of the graph. To enforce a limit, R-Solve uses the same technique used when original clauses are removed in Smart Repair. A subset of the original clauses is selected and "removed" and then immediately reinserted. The effect is that all learned clauses reachable from the selected original clauses are pruned from the graph. As an optimization, communication of the removed clauses to the solvers is delayed until the next actual Smart Repair operation. In the interim, new learned clauses that have removed clauses as antecedents are added to the set of clauses to be removed at the next operation rather than added to the DAG. Through this lazy mechanism, the solvers can continue to benefit from the pruned (but not invalid) learned clauses until the next Smart Repair operation. This reduction can be repeated iteratively until the desired reduction in total graph size is achieved.

V. PERFORMANCE

R-Solve uses the *Salt* [16] language and translation tool for generating CNF problem instances from high-level constraints. The Salt language provides ~120 operators supporting logical, bit set and fixed-point arithmetic expressions. The Salt compiler tunes the output CNF for search-efficiency and has optimizations specifically targeted at accelerating the processing of arithmetic expressions. The current version of Salt is a static, single-pass tool, but supports the generation of labeled constraint blocks that can be independently activated or deactivated by an R-Solve console script.

Modern SAT solvers are adept at solving problems spanning logical and arithmetic constraints. As reference points, Glucose will solve a hard (for humans) Sudoku puzzle in ~5-10ms of CPU time. Rendered from Salt, Glucose can typically recover the integer roots of a randomly generated 6^{th}-degree polynomial with 32-bit integer coefficients in 500-2000ms of CPU time. Reducing to cubic polynomials with 16-bit coefficients, the CPU time drops to 10-30ms. Most 64-bit problems are intractable. SAT is not, by any means, the best method for finding roots and in framing most problems it is advantageous to prefer logical to arithmetic constraints. However, these reference points have value in that they ground expectations and show that SAT can solve both logical and non-linear arithmetic constraints in a common framework.

For testing, we have been using a generic resource allocation multi-objective optimization problem framework rendered through Salt in which one or more among a heterogeneous set of *assets* must be applied to address each of a set of distinct *tasks*. Asset application is according to rules – some specific to the nature of the asset, some to the nature of the task and some universal. The rules simulate complex coordination, deconfliction and mutual-exclusion constraints. Further, a fitness function can be defined that includes components for tasks achieved or not achieved, assets used or not used, specific costs for applying a particular asset to a particular task (e.g., simulating proximity or suitability of the asset to the task) and a global component derived from the complete solution. The global component allows modeling of such considerations as the quality of the distribution of unused assets or unaccomplished tasks – it is a function that may not be piecewise computable, can involve non-linear terms such as sum-of-squares, and has meaning only in the context of the complete solution. The current testing framework does not include an explicit temporal component (planning), but that would be a straightforward extension. The framework allows optimization of the fitness function and also rule modification (e.g., elimination of an asset, addition or a new task, new rules for accomplishing a task, etc.).

Because of the static nature of Salt, our current testing requires the pre-generation of all potentially necessary labeled constraint blocks followed by scripting that simulates what would otherwise be the automatic process of activating or deactivating the clause blocks to drive to a solution. As an illustrative example, we simulated an optimization problem involving 12 tasks and 42 interacting assets split evenly into two kinds, with two classes in each kind, where one of each kind was necessary to complete each task. All tasks had to be completed and each asset could only be assigned to at most one task. Each asset used included a fixed cost plus a compatibility cost based on proximity to the task (different for each task). A global function assigned higher fitness to solutions that maintained a better balance between unused classes of asset within each kind. The fitness function was computed using 16-bit arithmetic (Salt supports any choice of bit precision, so the precision can be fit to the problem). The formulation took advantage of opportunities to reduce arithmetic constraints to easier logical constraints (e.g., a maximum Euclidean distance constraint could be reduced to a logical constraint prohibiting asset-task matches that would be out of range). The goal was to find a provably optimal solution.

The problem required 16 iterations of Smart Repair to converge on the optimal fitness value. The problem consisted of 22,430 variables with ~97,000 clauses active at any one time (~90,000 common to all instances). Running Glucose serially on the 16 SAT instances required 19.7s of total single-node CPU time. With Smart Repair using a single-node ensemble, the total CPU time to converge on the optimal solution was reduced to 3.1s (an 84% reduction). Subtracting out the time to the first solution (0.96s) common to both methods, Smart Repair completed the remaining 15 iterations 89% faster than the Glucose method (restart at each fitness level).

For the single-node ensemble, the Smart Repair graph reached 220,053 nodes. For a 4-node ensemble the graph increased to 652,093 nodes. The current implementation supports up to 63 solvers. In testing, clause generation rates are highly variable, but the master process has been able to service up to 16 solvers consistently for problems that can be solved in a few minutes, and more solvers in many cases. We have, however, been able to construct pathological examples that overwhelm the master process with a combination of high-frequency conflicts and learned clauses with huge numbers (> 5000) of antecedents. We have also observed performance degradation on long runs as the graph gets large. We are using these examples to investigate ways to further improve the management of the Smart Repair graph.

VI. FUTURE WORK

The remarkable progress in effective SAT solving over the past two decades has been driven, in large part, by annual competitions that measure speed at solving a range of both structured and unstructured (random) problems. Quantifiable progress on a range of problems has been an enormous driver of innovation in the academic SAT community. However, if there is a criticism against competitive solving, it is that its focus is too narrow – performance is only measured on problems that have been pre-rendered to CNF. To the extent that solvers try to preprocess problem instances for better performance, the effort is generally spent applying pattern-driven syntactic transforms that, at best, rediscover (often at considerable cost) the lost underlying substructure of the root problem being solved. It is our belief that the largest unexplored territory of future progress in practical SAT solving is in expanding consideration of the problem being solved beyond the narrow limits of the CNF representation. R-Solve provides a platform for that exploration.

We are now developing a second-generation version of Salt that can drive Smart Repair and fully leverage the power of unrestricted incremental solving. To date, our benchmarking has been limited to problems where we have pre-rendered all of the possible necessary constraint blocks to CNF and then used scripts to activate and deactivate them – simulating what should be an automatic, on-the-fly process. The new version of Salt will include a persistent intermediate representation that can accept constraint updates in the Salt language and then reflect those changes to clause block updates. Salt will also be able to interact directly with the outward-facing API to automatically generate the constraint updates necessary to drive optimization problems based on a fitness function.

In addition to functionality improvements and maintaining existing routines for optimizing the CNF based on the substructure of the constraint type being encoded, Salt will carry forward ideas that we have begun to experiment with that leverage problem meta-data to accelerate solving. For instance, an integral part of the process for transforming a Boolean expression into CNF is the introduction of auxiliary variables that capture subexpressions through enforced logical equivalence [17]. Salt already makes strategic use of the placement of these variables, for example, binding them to the carry bits in addition operations. However, the distinction between base and auxiliary variables is not provided to the solver. Experiments suggest that the SAT algorithm can be accelerated by biasing the variable selection in the decision heuristic toward base variables, leading to statistically faster conflict discovery. Future versions of R-Solve will support a meta-data format that ensures that this sort of information is carried through the solution process.

On the solver side, the centralized nature of Smart Repair provides a platform to take more aggressive advantage of information being produced by the component solvers. This includes more advanced routines for deciding which clauses to share and a system of intelligent and adaptive ensemble management that we have begun to experiment with called *Smart Configuration*. Recent work has demonstrated that machine learning techniques can be applied to meta-data collected from problems rendered in CNF and used to select a best-suited solver from a portfolio of candidate solvers [18]. For R-Solve, we are working toward a more advanced version of this idea that profiles Salt-level constraints and accepts user-supplied information such as an expectation of whether the problem is likely to be SAT vs. UNSAT to configure an entire portfolio of solvers. Then, as the solvers run, centralized monitoring in the master process measures the relative progress of the component solvers and tunes the ensemble by either modifying or replacing underperforming members.

Finally, we believe that the basic ideas in this paper should ultimately be applicable to Satisfiability Modulo Theories (SMT) solvers [19]. SMT solvers have a similar top-level structure to SAT solvers, but with underlying theory solvers for specific constraint groups. Applying the ideas of optimized compilation, parallelism for HPC, Smart Repair and Smart Configuration to solvers in the SMT domain should provide an even more extreme class of solver.

REFERENCES

[1] S. Malik and L. Zhang, "Boolean Satisfiability From Theoretical Hardness to Practical Success," CACM, Vol. 52, No. 8, pp. 76-82, August 2009.

[2] G. Audemard and L. Simon, "Predicting Learnt Clauses Quality in Modern SAT Solvers," IJCAI'09, July 2009.

[3] J. Davies and F. Bacchus, "Using More Reasoning to Improve #SAT Solving," AAAI-07, July 2007.

[4] N. Eén and N. Sörensson, "Temproral Induction By Incremental SAT Solving," Electr. Notes Theor. Comput. Sci. 89(4): 543-560 (2003).

[5] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," DAC 2001, June 2001.

[6] L. Zhang, C. F. Madigan, M. Moskewicz and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," ICCAD 2001: 279-285.

[7] D. Singer, "Parallel Resolution of the Satisfiability Problem: A Survey," Publications du LITA, N° 2007-101 Service de reprographie de l'U.F.R. M.I.M. Universit´e Paul Verlaine – Metz, 2007.

[8] Y. Hamadi and C. M. Wintersteiger, "Seven Challenges in Parallel SAT Solving," AI Magazine 34(2): 99-106 (2013).

[9] J. Whittenmore, J. Jim and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," DAC 2001, June 2001.

[10] S. Wieringa and K. Heljanko, "Asynchronous multi-core incremental SAT solving," TACAS'13, pp. 139-153, March 2013.

[11] H. Zang, M.P. Bonacina and J. Hsiang, "PSATO: A Distributed Propositional Prover and its Applications to Quasigroup Problems," Journal of Symbolic Computation, Vol.21, pp.543-560, 1996.

[12] W. Blochinger, C. Sinz and W. Küchlin, "PaSAT-Parallel SAT-Checking with Lemma Exchange: Implementation and Applications," SAT'01, June 2001.

[13] http://www.labri.fr/perso/lsimon/glucose/

[14] https://code.google.com/p/protobuf/

[15] N. Eén and A. Biere, "Effective Preprocessing in SAT through Variable and Clause Elimination," SAT'05, pp. 61-75, June 2005.

[16] J. Ezick, "Salt: An Application Logic and Translation Tool for SAT," Reservoir Labs Technical Report, March 2007.

[17] G. Tseitin, "On the Complexity of Derivation in Propositional Calculus," In Studies in Constructive Mathematics and Mathematical Logic, Part 2, 1968.

[18] L. Xu, F. Hunter, H. H. Hoos and K. Leyton-Brown, "SATzilla: Portfolio-based Algorithm Selection for SAT, " Journal of Artificial Intelligence Research 32, pp. 565-606, 2008.

[19] R. Sebastiani, "Lazy Satisfiability Modulo Theories," Journal on Satisfiability, Boolean Modeling and Computation 1, 2006.