Advances and Insights into Parallel SAT Solving

Stephen Plaza, Ian Kountanis, Zaher Andraus, Valeria Bertacco, Trevor Mudge
EECS Department, University of Michigan, Ann Arbor, MI 48109-2121
{splaza, ikountan, zandrawi, valeria, tnm}@umich.edu

ABSTRACT

The recent improvements in SAT solving algorithms, driven by the quest to solve increasingly complex problem instances, has produced techniques whose objective is to prune large portions of the search space to converge quickly to a solution (for instance, conflict-driven learning). In particular, solutions have been suggested in this area which attack the problem by attempting to parallelize DPLL-based SAT. However, so far the results have been mixed, which, consequently, have led to a scarcity of research in this space. One of the challenges of this direction is that finding a good partitioning of the space is not straightforward when using a DPLL-based algorithm. Moreover, partitioning the problem limits the benefits that can be ripped from effective learning and good variable ordering heuristics in sequential solvers.

In this paper, we propose techniques which improve upon previous approaches, primarily by improving the quality of learning during the search. Our first technique implements a parallel version of recursive learning (RL), which we use as a preprocessor to simplify the initial instance. Unlike previous DPLL-based approaches, RL can be easily partitioned and each processor learns information that is beneficial to solving the problem at hand. Our results indicate that this initial preprocessing can be done efficiently when divided among several processors, thereby boosting the performance of solving the resulting instance. In addition to RL, we explore strategies for improving parallel DPLL-based algorithms by adopting heuristics which have proven effective in the sequential domain, such as VSIDs, and by applying them to shared learning and partitioning. We show results that indicate improvement over previous work and reveal that future enhancement could lead to better mechanisms for pruning search space through shared learning strategies in a parallel framework.

1. INTRODUCTION

Several approaches to develop efficient SAT solvers have been proposed in the past, including techniques involving resolution and heuristic local search. Most SAT solvers today, however, use an algorithm that is a variant of the DPLL algorithm introduced in [1]. Several improvements have been made to enhance the performance of the baseline DPLL algorithm: *conflict-driven learning* and *non-chronological backtracking* have been studied in [2], and have contributed to exceptional performance improvements over a wide variety of SAT instances. Boolean Constraint Propagation (BCP) is a component of SAT solvers that is used extensively at run-time: its performance has been streamlined in [3]. Techniques to find a good decision order for the variable in the SAT instance [3] and various preprocessing techniques [4] have also been considered.

Despite these advances, many important SAT problems are still beyond the capabilities of present solvers. Because of the inherent complexity of this problem, the amount of improvements that can be made seems to be limited, and, as a result, parallel approaches seem to have the potential for breaking the performance barrier.

The most common approach to parallelizing a SAT solver uses the DPLL-algorithm as a starting point and assigns different *guiding paths* [5, 6] or initial sets of assignments to different processors so that the SAT space can be partitioned effectively [7]. However, the authors offer little insight of what would be an effective guiding path to adequately partition the work. A second issue in these approaches involves the aspect of communicating learned information obtained on individual solvers through conflict-driven learning to other solvers running in parallel. The analysis of this aspect is also inconclusive, since only simple filtering heuristics are suggested [7, 5]. The results of several of these parallel ventures are generally positive but inconsistent. There is little experimental data that shows that the techniques developed are effective.

In this paper we present advances in parallel SAT solving by adopting strategies which effectively prune the search space. The difficulty of partitioning a DPLL-based algorithm, which we grasped from the literature, led us to consider the alternative of using *recursive learning* (RL). Potentially, a SAT instance can be solved entirely by recursive learning, but this strategy would be much less efficient than the original sequential DPLL algorithm. However, we found that the recursive learning algorithm can be effectively partitioned and therefore could be efficiently applied as a preprocessor for improving the initial SAT instance which our results indicate in several cases.

We also provide improvements to DPLL-based parallel algorithms by incorporating strategies effective in the sequential domain and applying them in the parallel one. In particular, we propose different strategies for partitioning a SAT instance among individual DPLL-based solver engines with the intent of maximizing the amount of useful computation. The most effective techniques take advantage of variable counting strategies originally developed for sequential solvers. Finally, we present learning techniques specific to a parallel context, that perform considerably better than strategies previously considered.

In the following sections, we begin our analysis with some of the relevant previous work and the necessary background information to present our techniques. We then discuss our three main contributions in the context of our implementation of a parallel SAT solver, presenting our strategy for 1) parallelizing recursive learning, 2) choosing an appropriate *guiding path* when splitting a SAT instance, and 3) offering a learning filter that selects clauses based on the activity of the literals they include. We present experimental results that evaluate each of these techniques individually, and conclude by outlining future research directions.

2. PREVIOUS WORK

Research on the parallelization of SAT problems has pursued two general directions–coarse-grain parallelization and fine-grain parallelization. Fine-grain parallelization strategies target the Boolean Constraint Propagation (BCP) used extensively in solving SAT problems. The assignment of a variable must be propagated to every relevant clause to check for implications and conflicts that can result from the assignment. BCP can be parallelized by splitting the number of clauses among separate processing units to achieve linear speedup in BCP. Coarse-grain parallelization strategies involve finding a partition of the search-space by starting different processing units with different variable assignments.

Fine-grain parallelization would be hard to implement on general distributed or CMP architecture because of the frequency that

BCP is used and therefore the demanding communication requirement that it entails. As a result, fine-grain parallelization has been implemented on specific parallel architectures [8] and also by mapping the SAT instance onto specific architecture such as FPGAs [9, 10]. These techniques provide noticeable parallelization of SAT but are limited by the lack of scalability and by the architectural specificity required.

Coarse-grain parallelization has been implemented on large, distributed architectures as well as smaller multiprocessor approaches. Coarse-grain parallelization is limited by two factors. In SAT, determining the *guiding paths* that partition the problem adequately among the processors is not straightforward. In addition, conflict-based learning done on sequential solvers must be communicated intelligently to other solvers to improve the performance of the parallel solver. Several designs have been suggested but most use common strategies for partitioning the problem and for communicating learned information on the sequential solver [11, 6, 7]. These designs all show improvement but tend to focus more on the architectural issues such as communication and load balancing but little on improved *guiding paths* or learning strategies.

3. BACKGROUND

The SAT problem consists of choosing an assignment for a set of variables, V, that satisfies a Boolean equation or discovering that no such assignment exists. The Boolean equation is expressed in conjunctive-normal form (CNF), $F = (a+b+c)(d+e)$...., which is a conjunction of *clauses*, like $(a+b+c)$ and $(d+e)$. A clause is a disjunction of *literals* which are the input variables of F that can have a polarity of positive or negative.

3.1 SAT Algorithm

The basic approach for solving SAT is a branch and bound algorithm called the Davis-Putnam (DPLL) algorithm [1]. Several innovations such as non-chronological backtracking, conflict-driven learning, and decision heuristics greatly improve upon this approach [2, 3, 12]. The essential components of a SAT solver are shown in the algorithm in Figure 1.

```
search {
  while(true) {
    propagate();
    if(conflict) {
      analyze_conflict();
      if(top_level_conflict) return UNSAT;
      backtrack();
    }
    else if(satisfied) return SAT;
    else decide();
  }
}
```

Figure 1: Pseudocode for a generalized DPLL-SAT Algorithm

The `search` function explores the decision tree until a satisfying assignment is found or the entire space is traversed without finding any such assignment. The `decide` function picks the next literal that will be propagated. Many methods exist for picking this variable. In the next section, we will describe the VSIDS algorithm developed in Chaff [3]. The `propagate` function performs the BCP operation. If this assignment is not a satisfying one, the algorithm checks whether the current assignment has produced a conflict. The `analyze_conflict` function does the conflict-driven learning. The information learned through this process is encoded as clauses and will be referred to as *learnts*. The `backtrack` function undoes the previous assignments that contributed to the conflict. Periodically, the `search` function is stopped and restarted. Results show that random restarts are critical in finding a solution in a reasonable amount of time, particularly for complex instances, by avoiding portions of the problem where the learning is very localized [3, 13].

3.2 VSIDS Heuristic

Various strategies for choosing a variable in `decide` can greatly effect performance. Various heuristics have been explored in the past in the attempt to select decision variables in a sequence that brings the solver to fast convergence. The VSIDS (Variable State Independent Decaying Sum) algorithm involves associating an "activity" count with each literal [3]. Whenever a learnt clause is generated from a conflict, each literal that occurs in the clause has its activity incremented by a certain value. Periodically, every literal in the problem instance is divided by some experimentally tuned number. When a `decide` is called, the highest-valued undecided literal is chosen.

3.3 Recursive Learning

In addition to conflict-driven learning, other techniques can be employed to discover learnt clauses that could prune the search-space. One such technique, recursive learning (RL), involves assigning selected variables in the CNF instance to certain values with the objective to discover *necessary variables assignments*. A necessary variable assignment is a variable that must be assigned to a certain value in any satisfiable solution of the instance. For instance, in the formula, $(a + b)(a' + c)(b' + c)$, the variable c is assigned to 1 because either a or b must be assigned to 1 to satisfy the clause, $(a + b)$. Hence, a 1-literal learnt clause (c) can be added to the database. Recursive learning algorithms have been developed to identify these situations. Moreover, as the name implies, the algorithm can be applied recursively to construct learnt clauses of size greater than one literal. A more detailed presentation of these recursive applications of the algorithm is addressed in Section 4.1. Recursive Learning has been examined in ATPG applications as well as a technique to adopt during DPLL to augmenting the performance of a SAT solver [14, 15, 16, 17, 18, 19].

3.4 Parallel SAT Approaches

Effectively partitioning the search space of a SAT problem is an important aspect in the parallelization of the DPLL-algorithm. Before the popularization of conflict-driven learning, this partitioning and the derived goal to balance the load among the processors dominated the research scene of parallel SAT solvers. One strategy used involves sending to each SAT solver an initial set of assignments called a *guiding path* [6, 5]. Several strategies can be employed to find a guiding path for a particular solver. In [5], the next decision variable chosen by the heuristic in the sequential solver is used to decide how to construct a guiding path. In general, these strategies do not guarantee that the sub-problems created by each guiding path will result in a balanced work load among the solvers.

With the development of conflict-learning, parallel solvers need the ability to communicate newly learned information to each other. Because of the large amount of communication required to send all this data along with the increased BCP costs stemming from the addition of several learnt clauses from different solvers, there is an additional requirement to deploy some type of "filtering" strategy to reduce the number of learnt clauses that are communicated among parallel solvers. The strategy most commonly used involves filtering the learnt clauses sent to other SAT solvers based on their size [7, 5]. For instance learnt clauses with more than x literals will not be sent to other solvers. The logic behind this heuristic comes from the belief that learnt clauses with fewer literals procure a more aggressive pruning of the search space.

4. NEW APPROACHES IN PARALLEL SAT

Previous implementations of parallel SAT solvers use very similar techniques for learning and for creating new guiding paths. Although these implementations often perform faster than non-parallel SAT solvers, the results are not clearly attributed to any particular technique. In other words, it is unclear whether the speedup is attributable to a good parallel algorithm or simply to the variance among various heuristics in a sequential solver. In addition, these parallel approaches fail to take advantage of techniques that have achieved success in the sequential domain. For example, a sequential solver using effective learning can achieve orders of magnitude speedup in many cases. However, learning in a parallel solver has not been implemented effectively and many important learnts in one solver may never be added to another. As a result, current sequential solvers can actually outperform their parallel counterparts.

In the following few sections, we introduce different techniques that address some of the limitations of previous parallel SAT implementations. In the next section, we implement a parallel recursive learning algorithm as a preprocessor to a sequential version of a SAT solver. Parallelizing recursive learning is straightforward and offers the sequential solver learnt clauses that prune its space considerably. We then implement a DPLL-based distributed solver that carefully picks guiding paths and exchanges learnt clauses intelligently between the processors.

4.1 Recursive Learning Preprocessor

A basic definition of recursive learning was offered in Section 3.3. Recursive learning (RL) is one way to further constrain the initial problem by identifying necessary assignments. Despite the exponential complexity of the RL algorithm, we noticed that the ability to parallelize RL is as simple as dividing the problems into N pieces for N different processors or clients. Because of this behavior, we implemented a parallel RL algorithm to act as a preprocessor for a DPLL-based sequential solver. Even though the RL learnts reduce the computation time of a sequential solver it often becomes prohibitively expensive to compute these learnts as a preprocessing step. Therefore we offer a fundamental contribution to parallel SAT by offering a parallel approach to RL that has the potential to reduce the search space and subsequently improve the performance of a sequential solver. The following few paragraphs explain how RL is performed sequentially and then explains how this algorithm is parallelized.

The complexity of the RL algorithm depends on the level of recursion that is desired. For instance, 1-RL would involve the computation of necessary 1-literal assignments involving 1 level of recursion. In others words, 1-RL produces learnt clauses of 1-literal size. In general, X-RL produces learnt clauses of X literals and requires more computation as X increases. Algorithms will now be presented for 1-RL and 2-RL and computation for any X-RL is easily inferred from this. The algorithm for performing 1-RL on a set of clauses is provided in Figure 2.

The algorithm begins by iterating through each clause in the set of clauses that define the SAT instance. We initialize the list, `necessary_assigns`, to contain all possible assignments, or $2N$ assignments for a problem with N variables. For each clause every literal in the clause is examined one at a time. One literal will be assumed at a time and any implications of this decision are propagated through all the clauses. The assignments implied by assuming this lit are intersected with the current necessary assignments that exist for the current clause being examined. The function `undoDecision` clears the assignment and implications produced previously. After every literal is examined in the clause, the literals that were propagated by each assignment can now be added as 1-literal learnt clauses to the initial problem.

```
1-RL_Algorithm(clauses) {
  for each C in clauses {
    necessary_assigns(UNIV);
    for each lit in C {
      assume(lit);
      implications = propagate();
      necessary_assigns =
                intersect(necessary_assigns,
                implications);
      undoDecision(lit);
    }
    1lit_learnts += necessary_assigns;
  }
}
```

Figure 2: Algorithm to Calculate Recursive Learning Learnts containing only one literal

Example 1. Consider the following instance: $(a+b)(a'+c)(b'+c)(c'+d)$. By performing the algorithm in Figure 2, we see that the literals c and d can be added to the initial list of clauses. This occurs because $(a=1) \rightarrow c \rightarrow d$ and $(b=1) \rightarrow c \rightarrow d$ when the first clause is examined. Since the assignment $a=1$ or $b=1$ needs to be made to satisfy the clause, $(a+b)$, and $c=1$ and $d=1$ are implied by both such assignments, both are necessary assignments and can be added as learned information.

The previous algorithm works with the knowledge that every clause must have at least one literal that is true for a satisfiable solution to be found. Other combinations of assignments can be made to produce other types of learnt clauses. Figure 3 shows the algorithm for 2-RL that produces learnt clauses with two literals.

```
2-RL_Algorithm(clauses) {
  for each C in clauses {
    for each lit1 in C {
      necessary_assigns(UNIV);
      assume(~lit1);
      for each lit2 in C {
        if(lit1 == lit2) continue;
        assume(lit2);
        implications = propagate();
        necessary_assigns =
                  intersect(necessary_assigns,
                  implications);
        undoDecision(lit2);
      }
      undoDecision(~lit1);
      generate_2lit_learnts(necessary_assigns);
    }
  }
}
```

Figure 3: Algorithm to Calculate Recursive Learning Learnts containing two literals

This algorithm is similar to the 1-RL algorithm; however, instead of assigning 1 literal at time, two literals are assumed and their implications are propagated. The first lit is assumed to be false. The innermost loop then iterates through the remaining literals in the clause and identifies implications that are common to all these sets of assignments. 2-literal learnt clauses are generated by calling `generate_2lit_literals`. The learnt clauses generated consist of the literal that was assumed false in the particular clause being examined and each literal that was implied after iterating through the remaining literals in that particular clause. To clarify this procedure and its correctness, we offer the following example:

Example 2. Consider a set of clauses $(a+b'+c)(a+b+d)(a+c'+d)$. The first step of the algorithm assigns $a=0$ and $b=0$ which implies d. Next, $a=0$ and $c=1$ which also

implies d. The algorithm identifies d as a necessary assignment when assuming $a = 0$. In other words the 2-literal learnt clause $(a + d)$ could be added to the set of clauses. This occurs because $a'b' \rightarrow d$ is logically equivalent to $(a + b + d)$ and $a'c \rightarrow d$ is logically equivalent to $(a + c' + d)$. By resolving $(a + b + d)$ with $(a + b' + c)$, $(a + c + d)$ is derived. Then $(a + c' + d)$ can be resolved with $(a + c + d)$ to produce $(a + d)$. Continuing with this algorithm literal b' would then be examined followed by c and then every other clause and literal in the formula. However, no other 2-RL clauses exist for this particular example.

Calculating learnt clauses using recursive learning can be an expensive procedure that can involve an exponential amount of work depending on how much recursion is performed. In our implementation of RL, we find all 1-RL and 2-RL learnts for a particular SAT instance as described in Figure 2 and Figure 3. We avoid other levels of recursion because of the complexity. Despite the potential usefulness of generating 1-literal and 2-literal clauses that can prune the search space of a SAT solver, the complexity of 1-RL and 2-RL might make it an unattractive preprocessor for a sequential SAT solver. However, we noticed that this algorithm can easily be parallelized while DPLL cannot.

We parallelize RL by initiating multiple client processors with the original SAT instance or set of clauses. A central server manages the work of the client processors. The server's first task is to assign to each client a section of clauses to examine. In other words, for two clients, one client would iterate through the first half of the clauses (the first loop in Figures 2 and 3 would be split into two) and the other client would examine the second half. When each client finishes its recursive learning algorithms, the server must obtain the new learnt clauses and concatenate them to the initial set of clauses. At this point the new set of clauses is used as input to a SAT solver (either sequential or parallel). The speedup achieved by parallelizing SAT is nearly linear to the number of client processors involved in recursive learning.

4.2 Parallel SAT Solver

In addition to implementing a parallel preprocessor based on RL, we implemented a parallel version of a DPLL-based SAT algorithm to identify useful partitioning and learning heuristics to gain insight into parallelizing SAT effectively. Our main goal is to develop a parallel framework that will generate many useful learnts and send those learnts to solvers that can benefit from them. Effective partitioning is needed to maximize the amount learnts that are generated.

Our design consists of multiple client processors connected via a high speed network with a single global server that coordinates the clients' efforts. In the next section, we examine a partitioning scheme that our distributed solver uses to effectively balance the problem among all the clients. We then discuss our strategy for sending learnt clauses from one client to another while ensuring that the extra cost in BCP associated with adding learnt clauses is offset by the quality of the learnt clauses added.

4.2.1 Splitting Strategies.

Several previous parallel SAT solvers explained their techniques for constructing guiding paths to partition the SAT instance among clients. In our parallel solver, we let the server pick the initial guiding paths to send to each client. For instance, if a is a variable in the SAT instance, literal a could be sent as a guiding path to one client and a' could be sent to another client. Because choosing a variable poorly will not partition the SAT search space effectively, the server picks an initial variable based on the VSIDS activity similar to how a sequential solver chooses a high VSIDS variable to assign the next variable to improve its performance. Since VSIDS activity changes during the SAT solver's execution, we experimentally decided on a insignificantly small amount of time that the server runs sequentially to produce VSIDS information that could be used to construct the initial guiding paths.

After the initial partition, the server's primary responsibility is in making sure that each client has work to do by requesting guiding paths that active clients recommend. For instance, if client A is solving the SAT instance with a guiding path a, b, c, d' (in other words, $(a = 1, b = 1, c = 1, d = 0)$) and client B is currently not doing any solving. Client A must decide on a variable x such that it will now have a guiding path (a, b, c, d', x) and client B will have a guiding path of (a, b, c, d', x') guaranteeing the completeness of the search. The client that is chosen to give the guided path is the client that has been working on its search space the longest. We pick the variable, x, based primarily on its VSIDS activity according to the client being chosen. This is a slight variation to some previous approaches that just choose the next variable picked by the `decide` function. However, when sequential solvers assign variables based on VSIDS activity, the next variable returned by `decide` is often close to the highest VSIDS activity. Either approach should be an effective way to construct new guiding paths. We decided to let the client decide x instead of the server because the client's VSIDS information is more particular to the sub-problem that is being partitioned than the initial VSIDS information obtained by the server.

4.2.2 Learning Strategies.

One of the challenges involved in parallelizing SAT is deciding what learnt clauses to send to which clients. Traditionally, the analysis of the effects of sharing learnt clauses among multiple processors solving SAT focused on using a size filter to choose learnt clauses that should be sent to each client. For instance, one strategy would be to send learnt clauses that only have fewer than 10 literals. This approach was adopted for two reasons. First, it is believed that smaller learnt clauses would more effectively prune the search space. Secondly, not limiting the amount of learnt clauses sent would negatively effect each solver by increasing the processing time required by BCP negating the utility of learning more information.

In our approach, we acknowledge the importance of limiting the amount of learnt clauses that are sent to other clients. However, we reject the notion that sending learnt clauses by size is effective for the same reason that learnt clauses obtained by minimal cut do not necessarily outperform 1-UIP cuts that involve more literals as explained in [20]. With this in mind, we filter learnt clauses based on the quality of those learnt clauses as indicated by an activity count that is associated with each learnt clause that indicates how much the learnt clause has been used by the sequential solver [21, 22] during conflict analysis. The concept is similar to VSIDS. We therefore choose the highest activity learnt clauses to send to other clients. As with previous solvers, we limit the number of learnt clauses we send based on some experimentally-guided threshold. The server, as with other approaches, prevents any learnt clause from being sent to a client that will immediately satisfy the learnt clause based on the client's guiding path.

5. EXPERIMENTAL RESULTS

We use the very extensible and efficient sequential solver, MiniSat [21], as the foundation of our parallel solver and for our baseline. We test our benchmarks on a 100 mbps network with Pentium 4 3.2GHz processors with 1GB of memory running Linux. The benchmarks that we use are described and developed in [23, 24, 25, 26]. We use three SATISFIABLE benchmarks, hanoi5, hgen,

Table 1: Runtime for the 1-RL algorithm using different number of clients

Benchmark	1Cl	2Cl	4Cl	8Cl
1dlx_c_bp_f	8.26	4.38	2.25	1.21
1dlx_c_ex_bp_u_f	21.86	11.5	6.29	3.25
4pipe	3.12	1.7	0.91	0.49
5pipe	13.3	7.12	3.89	2.04
9vliw_bp_mc	30.64	16.36	8.27	4.62
engine_4_nd	3.85	2.03	1.13	0.68
engine_5_nd_case1	45.61	24.84	13.94	7.62
hanoi5	0.15	0.08	0.04	0.02

Table 2: Runtime for the 2-RL algorithm using different number of clients

Benchmark	1Cl	2Cl	4Cl	8Cl
1dlx_c_bp_f	275	45	27	17
1dlx_c_ex_bp_u_f	2378	1574	728	521
2bitadd_10	0.07	0.05	0.05	0.04
3bitadd_31	9.01	9.11	9.01	7.86
4pipe	42	23	13	7.88
5pipe	1316	700	130	40
9vliw_bp_mc	1258	655	325	166
engine_4_nd	50	29	17	11
engine_5_nd_case1	2489	1398	746	487
hanoi5	0.47	0.31	0.19	0.14

and 3bitadd_31, while the rest are UNSATISFIABLE. Two of the benchmarks, hgen and urqh3x3, are randomly generated. We chose benchmarks from test suites that were solvable by MiniSAT and did not lead to negligible runtimes. Some large benchmark files were not included because they tended to timeout due to memory constraints in the underlying solver such as the omitted benchmarks from the engin_unsat test suite. Smaller benchmark instances from many of the represented suites such as 2pipe, 3pipe, and hanoi4 have negligible runtimes and were inadequate for demonstrating the performance of our approaches.

The following sections first examine the positive contribution of preprocessing the benchmarks with parallel RL and then show some preliminary results for implementing a parallel DPLL-based algorithm that reveal the potential for further research for developing a more comprehensive parallel learning framework.

5.1 Parallel RL Results

We implemented parallel versions of 1-RL and 2-RL as a preprocessor for a sequential solver. In Table 1, we show the computation time required to perform 1-RL on a benchmark for 1, 2, 4, and 8 clients. Two things stand out in this table. First, the computation time required for 1-RL is relatively insignificant when compared to the sequential time to solve the SAT instance. Second, the speedup achieved by adding more clients is nearly linear to the number of clients. Some benchmarks are excluded from the table because 1-RL does not generate any 1-literal learnt clauses. In these cases, the run-time for preprocessing is close to 0 seconds for 1 client.

Given the positive results of 1-RL, we explored the potential benefits of implementing parallel 2-RL. Most of the results in Table 2, do indicate the same linear trends as with 1-RL. In addition, benchmarks that do not have 2-RL learnt clauses can be computed in time close to 0 seconds. The only negative with 2-RL is that the

Table 3: Sequential solver runtime with and without the learnts found from Recursive Learning

Benchmark	Seq.	Seq w/1RL	Seq. w/2RL
1dlx_c_bp_f	669.17	211.40	173.08
1dlx_c_ex_bp_u_f	277.90	115.32	249.68
2bitadd_10	263.44	–	145.67
3bitadd_31	1597.62	–	0.31
4pipe	98.16	29.33	132.79
5pipe	50.30	43.79	21.92
9vliw_bp_mc	185.03	174.77	94.15
engine_4_nd	163.05	210.84	176.53
engine_5_nd_case1	68.13	50.99	83.66
hanoi5	36.90	238.62	952.54

Table 4: Overall runtime of the parallel SAT solver for different number of clients

Benchmark	1Cl	2Cl	4Cl	8Cl
1dlx_c_bp_f	669	379	178	137
1dlx_c_ex_bp_u_f	278	195	118	105
2bitadd_10	263	228	77	66
3bitadd_31	1598	3	3	3
4pipe	98	92	57	53
5pipe	50	70	70	56
9vliw_bp_mc	185	149	121	135
engine_4_nd	163	103	64	54
engine_5_nd_case1	68	81	57	59
fpga11_12_uns_rcr	628	781	292	253
fpga11_13_uns_rcr	1724	TIMEOUT	TIMEOUT	744
hanoi5	37	36	26	30
hgen6-4-24-n250-01	10087	118	175	88
hole10	37	47	46	28
urqh3x3	224	278	179	107

run-times of performing 2-RL often exceeds the runtime necessary to solve the instance on a sequential solver without RL preprocessor. However, this parallel solution makes 2-RL viable when more clients are added.

We have shown that calculating 1-RL and 2-RL can be done quickly. In Table 3, we illustrate the effectiveness of RL preprocessing for a sequential solver. The second column is the baseline that gives the time required to solve the instance without any preprocessing. The third and fourth columns show the time it takes when 1-RL is performed and 2-RL is performed respectively. According to the table, 1-RL shows pretty consistent improvements over the baseline. The only serious degradation is in the SATISFIABLE benchmark hanoi5 where additional learnt clauses probably steered the search heuristics in the wrong direction.

The performance of 2-RL is a little less uniform in its improvement. Most benchmarks do show improvement over the baseline. In addition, two benchmarks, 2bitadd_10 and 3bitadd_31 are very effective where 1-RL does not generate any 1-literal learnt clauses. The degradation seen is most likely attributable to not effectively filtering the 2-RL learnt clauses. We implemented a simple filter based on the number of learnt clauses to be added to the original problem. In general, these results indicate the potential of parallel preprocessing. RL preprocessing techniques coupled with effective filtering strategies could improve sequential solving dramatically.

5.2 Parallel Solver

In addition to preprocessing results, we offer preliminary results for a distributed parallel SAT solver that we developed. The runtimes that we produced for 2, 4, and 8 clients are displayed in Table 4. These results positively reflect on our implementation based on VSIDS generated guiding paths and activity based learning. Although, there is some variance in the numbers, the trends show improvement as the number of clients increase. There is one notable exception, fpga11_13_uns_rcr that reached a 10000 second timeout. In this case, 2 and 4 clients most likely perform poorly from an experimentally-tuned strategy that we implemented that occasionally deletes low activity learnt clauses. The extra-ordinary result for 3bitadd_31 is due to the increased ability of a parallel solver to find a satisfiable solution since it looks at a different portions of the search space simultaneously.

These results only show the potential benefits of parallelizing SAT, but do not provide much insight on whether our strategies are effective. The next two sections describe the effectiveness of our technique for generating guiding paths and our learning strategy.

5.2.1 *Splitting Results.*

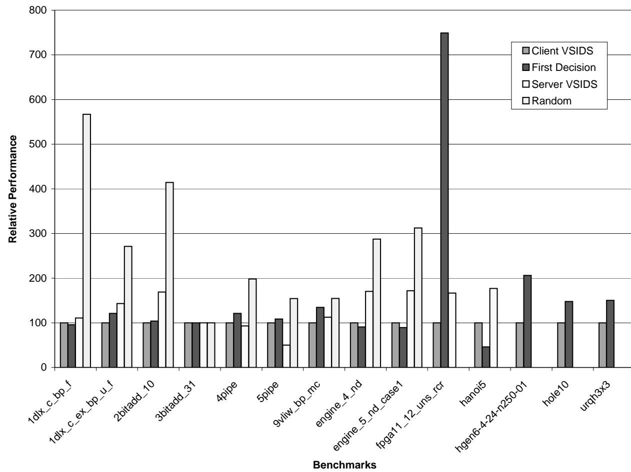

Figure 4: Relative performance of different techniques for generating guiding paths

In Figure 4, we show the relative improvement of our technique of using client generated guiding paths based on VSIDS activity with 4 clients. The first bar in the bar graph represents this technique and is set to 100. The other bars in the graph are normalized by the first one. The second bar describes a similar guiding path strategy as ours that simply uses the next variable to be assigned by the sequential solver to form the guiding path. The third bar shows the performance associated with generating guiding paths based on static VSIDS information produced by the server at the start of program execution to create the initial guiding paths. This VSIDS information on the server is never updated after this initialization. The final bar shows the performance associated with randomly generating guiding paths. Bars that are excluded indicate situations where that strategy required more than 10000 seconds and therefore timed-out. In this figure, shorter bars have better relative performance than taller bars.

Appropriately, randomly generating guiding paths performs much worse than the other strategies presented in Figure 4. More interestingly though, both client VSIDS guiding paths and first decision guiding paths show pretty consistent improvement over using the server's VSIDS information. Thus, it appears that guiding paths are better when generated on the clients where its activity information is more relevant to its current sub-problem. Finally, and even somewhat surprisingly, our technique of using highest VSIDS activity achieves relatively consistent speedup over the technique used in [5] despite the fact that the first decision variable in an assignment trail is often the highest VSIDS variable.

These results illustrate that dynamic techniques that reflect the current sub-problems improve the quality of the parallel SAT framework. Therefore, we believe that future research into maintaining global VSIDs data dynamically could produce partitions that will be the most beneficial to every solver.

5.2.2 *Learning.*

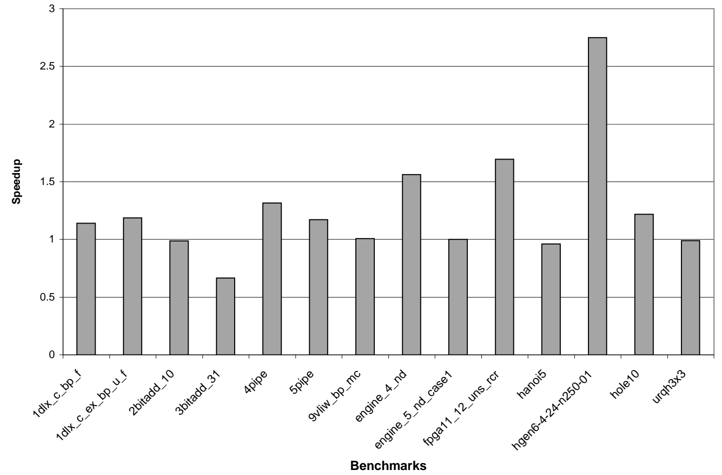

Figure 5: Speedup achieved by filtering learnt clauses by activity rather than by number of literals

In Figure 5, we explore the benefits of filtering learnt information based on the learnt clause's activity instead of its size. The figure shows the speedup associated with implementing an activity-based filter over a size-based filter (implemented with 4 clients). The results show, on average, an improvement when filtering is based on its activity rather than by size. Where speedup is not seen, the runtime for both techniques is approximately the same. These results indicate that sending small learnt clauses as a filtering technique is not very effective.

Future successful implementations of parallel SAT will require that the learnts available to each solver be the most relevant and useful for pruning its current subspace. Development into more global strategies such as maintaining activity counts across all solvers could be used to distribute learnts stored in a global pool to solvers that currently require them. In this fashion, the memory requirements of maintaining many learnts in a certain solver can be alleviated by providing a separate repository and sending them to the solver only when necessary. There is a twofold benefit in this 1) the BCP cost can be reduced by minimizing the size of the clause database and 2) each solver will have access to the most relevant learnts produced by all solvers.

6. CONCLUSIONS

In this work, we examined a range of techniques to prune the search space explored when attempting to solve a SAT problem instance. Specifically, we have developed an algorithm to generate 1-literal constraints in parallel, through the 1-RL algorithm, which constrains the initial instance by finding 1-literal assignments. We

show that 1-RL is an effective preprocessing technique, and because of the ability to easily parallelize RL, aggressive preprocessing schemes could lead to interesting new possibilities for additional speedups. Furthermore, we present an algorithm for a parallel DPLL-based SAT solver that offers insights into partitioning strategies and techniques for communicating useful learnt clauses. This is achieved by adopting effective techniques and heuristics in a sequential domain, such as VSIDS, and applying it to a parallel domain. The result is a solver that performs much faster than the sequential baseline.

Our results also indicate the potential for future work and improvement. Unlike other parallel DPLL-based approaches, we recognize the effectiveness of recent advances in SAT and strive to find models for it in the parallel setting. In particular, RL can be effectively parallelized to add high-quality learnts to the initial instance. Future work on heuristically filtering these added learnts could improve performance. Also, our work in DPLL-based parallelization indicates that improving the heuristics to take into account more global information could lead to further gains.

7. REFERENCES

[1] Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7** (1960) 201–215

[2] Marques-Silva, J., Sakallah, K.: Grasp: A search algorithm for propositional satisfiability. IEEE Transactions on Computers (1996)

[3] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: DAC, Proceedings of Design Automation Conference. (2001) 530–535

[4] Lynce, I., Marques-Silva, J.: Probing-based preprocessing techniques for propositional satisfiability (2003)

[5] Blochinger, W., Sinz, C., Kucklin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Computing **29** (2003) 969–994

[6] Zhang, H., Bonacina, M.P., Hsiang, J.: Psato: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation **11** (1996) 1–18

[7] Chrabakh, W., Wolski, R.: Gradsat: A parallel sat solver for the grid. UCSB Computer Science Technical Report (2003)

[8] Zhao, Y., Moskewicz, M., Madigan, C., Malik, S.: Accelerating boolean satisfiability through application specific processing. In: Proceedings of the 14th International Symposium on System Synthesis. (2001) 244–249

[9] Zhong, P., Martonosi, M., Ashar, P., Malik, S.: Using configurable computing to accelerate boolean satisfiability. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **18** (1999) 861–868

[10] Abramovici, M., DeSousa, J., Saab, D.: A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In: DAC, Proceedings of Design Automation Conference. (1999) 684–690

[11] Okushi, F.: Parallel cooperative propositional theorem proving. Annals of Mathematics and Artificial Intelligence **26** (1999) 59–85

[12] Zhang, H.: Sato: An efficient propositional prover. In: Proceedings of the 14th International Conference on Automated Deduction, Springer-Verlag (1997) 272–275

[13] Baptista, L., Silva, J.P.M.: Using randomization and learning to solve hard real-world instances of satisfiability. In: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, Springer-Verlag (2000) 489–494

[14] Kunz, W., Pradhan, D.K.: Recursive learning: an attractive alternative to the decision tree for test generation in digital circuits. In: In Proceedings of the International Test Conference. (1992) 816–825

[15] Aloul, F.A., Sakallah, K.A.: An experimental evaluation of conflict diagnosis and recursive learning in boolean satisfiability. (2000)

[16] Chakradhar, S.T., Agrawal, V.D.: A transitive closure based algorithm for test generation. In: Proceedings of the 28th ACM/IEEE Design Automation Conference. (1991) 353–358

[17] Yuji, M.Z.: (Speeding up sat based atpg for logic verification by recursive learning)

[18] Bacchus, F.: Enhancing davis putnam with extended binary clause reasoning (2002)

[19] Silva, J.P.M., Glass, T.: Combinational equivalence checking using satisfiability and recursive learning. In: DATE. (1999) 145–149

[20] Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: ICCAD, Proceedings of the International Conference on Computer Aided Design. (2001) 279–285

[21] Een, N., Sorensson, N.: An extensible sat-solver. In SAT (2003)

[22] Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat solver (2002)

[23] Velev, M.: Sat benchmarks. (http://www.ece.cmu.edu/ mvelev/sat_benchmarks.html)

[24] SATLIB: The satisfiability library. (http://www.satlib.org)

[25] Aloul, F.: Sat benchmarks. (http://www.eecs.umich.edu/ faloul/benchmarks.html)

[26] Crawford, J.M.: Sat benchmarks. (http://www.cirl.uoregon.edu/crawford/beijing/)