Learning from BDDs in SAT-based Bounded Model Checking

Aarti Gupta, Malay Ganai
NEC Labs America
Princeton, NJ
U.S.A.
{agupta,malay}@nec-labs.com

Chao Wang
Dept. of Electrical Engineering
University of Colorado
Boulder, CO, U.S.A.
wangc@colorado.edu

Zijiang Yang, Pranav Ashar
NEC Labs America
Princeton, NJ
U.S.A.
{jyang,ashar}@nec-labs.com

ABSTRACT
Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT) procedures has recently gained popularity as an alternative to BDD-based model checking techniques for finding bugs in large designs. In this paper, we explore the use of learning from BDDs, where learned clauses generated by BDD-based analysis are added to the SAT solver, to supplement its other learning mechanisms. We propose several heuristics for guiding this process, aimed at increasing the usefulness of the learned clauses, while reducing the overheads. We demonstrate the effectiveness of our approach on several industrial designs, where BMC performance is improved and the design can be searched up to a greater depth by use of BDD-based learning.

Categories and Subject Descriptors
B.6.3 [**Design Aids**]: Verification.

General Terms
Algorithms, Performance, Experimentation.

Keywords
Boolean Satisfiability, SAT, SAT solvers, BDDs, learning, BDD learning, bounded model checking, property checking.

1. INTRODUCTION
As hardware design complexity continues to rise, there is a greater need for effective verification in order to avoid costly errors. Formal verification techniques like symbolic model checking [1, 2], based on the use of Binary Decision Diagrams (BDDs) [3], offer the potential of exhaustive coverage and the ability to detect subtle bugs. However, these techniques do not scale well in practice due to the state explosion problem.

In contrast, Bounded Model Checking (BMC) [4] focuses on finding bugs of bounded length, and successively increases the bound to search for longer traces. Given a design and a correctness property, it generates a Boolean formula, which is satisfiable if and only if there exists a witness/counterexample of length *k*. The satisfiability check is typically performed by a backend SAT solver. Due to the many recent advances in SAT solvers [5-8], SAT-based BMC can handle much larger designs in practice. As demonstrated by these SAT solvers, learned clauses play a crucial role in determining their performance, both by pruning the search space, and by dynamically affecting the choice of decision variables. At the same time, there is an overhead associated with the addition of each learned clause. Therefore, learning techniques must ensure a good tradeoff between the usefulness and the overheads of adding learned clauses.

DAC 2005, June 2-6, 2003, Anaheim, CA, USA.

1.1 BDD Learning
Our approach focuses on the use of learned clauses generated by a BDD-based analysis of the SAT problem – we call this *BDD Learning*. Essentially, a BDD is used to capture the relationship between Boolean variables of (a part of) the SAT problem, in the form of a characteristic function. In such a BDD, each path to a "0" (false) node denotes a conflict. A learned clause corresponding to this conflict is easily obtained by negating the literals that define the path. Since a BDD captures all paths to 0, i.e. all possible conflicts among its variables, the potential advantage is that *multiple* learned clauses can be generated and added to the SAT solver at the same time. In contrast, conflict-driven learning [5] typically analyzes a single conflict at a time. An example with multiple learned clauses generated from a BDD is shown in Figure 1. (The figure shows multiple terminal nodes, and no inverted edges for exposition only; standard ROBDDs can be used otherwise.)

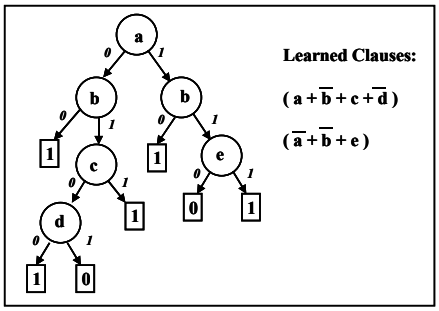

Figure 1: Example for BDD Learning

In BMC, or any circuit-based SAT application involving time frame expansion, the bulk of the Boolean constraints arise from the *k*-times unrolled transition relation of the design. Therefore, the circuit structure graph of the transition relation is a natural candidate for creating useful BDDs. The main goal for our BDD Learning technique is to be *effective* but *lightweight*, i.e. it should improve the performance of the SAT solver, but without overwhelming the SAT solver heuristics. This rules out the possibility of learning on a global scale, i.e. creating a BDD and learned clauses for every node in the circuit graph. In our experiments also, we found this to be too expensive. Therefore we

perform learning selectively, i.e. by selecting "seed" nodes in the circuit graph around which to perform learning. We distinguish between the following kinds of learning:

- *Static learning*: seed nodes are selected using static information, and learned clauses are added statically, before the SAT solver starts the search.
- *Dynamic learning*: seed nodes are selected using dynamic information, and learned clauses are added on-the-fly, during the SAT search.

Note that in static learning, the seed selections reflect only the static circuit structure, while for dynamic learning they also reflect the dynamic state of the SAT solver.

The distinguishing criteria between static and dynamic learning are – how the seeds are selected, and when the clauses are added. Note that we do *not* distinguish based on when the learned clauses are *generated*. For example, it is possible to generate BDDs and learned clauses either in advance for all seed nodes, or on-the-fly for selected seed nodes. Furthermore, BDDs can be created for seed nodes in the unrolled transition relation (starting from the initial state), or a single copy of the transition relation (without the initial state) called the template. In the former case, learned clauses are generated directly in terms of the different SAT variables in the expanded time frames. In the latter case, learned clauses are generated in terms of template variables. With each successive unrolling of the transition relation, a learned clause can be generated for the new time frame by substituting corresponding SAT variables for the template variables. (This is similar to clause replication [9], also discussed later.) In this paper, we describe details for selecting seed nodes on the unrolled transition relation, with on-the-fly generation of BDDs and learned clauses. These ideas can be extended easily to other variations described above.

We use a BDD Learning engine to encapsulate the essential tasks of seed selection, creation of BDDs, and generation of learned clauses. This engine is integrated with a standard SAT solver. In particular, BDD Learning is performed in conjunction with other learning mechanisms in the SAT solver, e.g. conflict-driven learning [5]. Furthermore, a learned clause generated by the BDD Learning engine is treated similar to other learned clauses by the SAT solver. For example, scores of variables related to these learned clause are incremented [7, 8], which can significantly impact the future decisions made by the SAT solver.

1.2 Our Contributions

Though the idea of generating learned clauses from a BDD is relatively straight forward, and has also been mentioned by other researchers [10], we believe our work to be the first to explore the associated tradeoff issues in integrating such learning within a SAT solver. We describe heuristics and parameters in our BDD Learning engine which are targeted at increasing the usefulness of the learned clauses, while reducing the overheads. In particular, our heuristics for dynamic seed selection can be potentially combined with other kinds of external learning, in order to achieve a balance in this tradeoff.

To the best of our knowledge, our work is also the first to demonstrate the practical effectiveness of BDD Learning in the context of an application such as BMC. We report experimental results on industrial designs for our prototype implementation of BMC, enhanced with a BDD Learning Engine. Our results show runtime reduction of up to 73% for searching the same number of time frames as basic BMC, and deeper searches (with additional time frames) within the allotted time.

Finally, though this paper focuses on the BMC application, our BDD Learning technique can be used to potentially improve performance in other circuit-based SAT applications as well, such as equivalence checking [10], automatic test pattern generation (ATPG) [11] etc.

1.3 Related Work

The idea of combining BDDs and SAT for verification is not new. Given that both techniques perform an implicit search on the underlying Boolean space, it is no surprise that many different ways of combining them have been explored over the years, frequently suited to the target application. Their relative benefits have been combined in many verification applications such as equivalence checking [10, 12-16], BMC [10], image computation [17], and model checking [18, 19].

The distinguishing feature of our BDD Learning method is that it directly adds learned clauses to the SAT solver. Its use is orthogonal to other BDD-based simplifications for SAT problems, such as for simplifying the goals [10], or for simplifying the problem using BDD sweeping [20]. It is also possible to combine BDD Learning techniques with clause replication techniques [9]. Clause replication enables reuse of learning in SAT applications involving time frame expansion, such as BMC. Essentially, a learned clause, consisting of literals from specific time frames, is replicated by substituting corresponding literals from other allowable time frames. This was originally applied to clauses learned by conflict analysis [9], but it applies also to clauses learned by other techniques, including BDD Learning. Furthermore, our ideas of dynamic seed selection can be used for *selective* replication, to increase its effectiveness in practice.

2. BDD Learning Engine

```
Bdd_Learning_engine() {
   update_engine_info();
   if (ready_for_learning) {
      node = select_a_seed();
      bdd = create_a_bdd(node);
      cl_list= generate_learned_clauses(bdd);}
   return(cl_list);
}
SAT_Solve() {
  while(1) {
     cl_list = bdd_learning_engine();  // new
     if (add_clauses(cl_list)== UNSAT) // new
       return UNSAT;                   // new
    if (decide_next_branch()) {
       while(deduce() == conflict) {
         blevel = analyze_conflict();
         if (blevel == 0) return UNSAT;
         else backtrack(blevel); }
     } else return SAT;
  }
}
```

Figure 2: Overall Flow of BDD Learning Technique

The overall flow of our technique is shown in Figure 2. The BDD Learning engine performs the essential tasks of seed selection, creation of a BDD, and generation of learned clauses from the BDD. Its integration with a typical DPLL-style SAT solver [21] is shown also, where the additional steps are marked "*new*" in the figure. The remainder of this section describes details of the BDD

Learning engine, and the next section focuses on its integration with the SAT solver.

2.1 Seed Selection

We explored many different heuristics for selecting seeds around which BDD Learning is performed. Since the goal is to improve the SAT solver performance, the seed selection heuristics are based on the decision ordering heuristics of the SAT solver itself. We assign a rank to each candidate seed node (not a primary input) based on criteria described below. We also keep track of which variables have already been chosen as seeds, to avoid adding duplicate learned clauses. The seed selection heuristics, listed *SSH1-SSH5*, are:

- *SSH1: Next decision rank*: The idea is to preempt the learning that would be performed by the SAT solver for a future decision. Rather than learn a single conflict clause, we learn all related conflict clauses simultaneously.
- *SSH2: Past decisions ranked back from the current one*: In this case, the idea is to learn some more clauses about variables that have been important in the past.
- *SSH3: Most frequent decisions*: The idea is that a variable chosen frequently as a decision variable, along different paths in the SAT search, is a good candidate for preempting future learning.
- *SSH4: Decisions at back-leap levels*: The back-leap technique identifies a good decision level to backtrack to, in the presence of many conflicts localized within a range of decision levels [22]. Like restarts, it allows jumping out of locally bad regions, but without having to backtrack all the way up to the starting decision level. The intuition here is that additional learning about decision variables at the back-leap levels is likely to be useful. The seeds are ranked from the backleap level down to the current decision level.
- *SSH5: Decisions at levels most often backtracked to*: Since difficult SAT problems are characterized by more number of backtracks, we keep track of decision variables at those levels to which maximum number of backtracks have taken place. The intuition is that these variables are likely to be causing more conflicts, and additional learning might help.

We experimented with choosing a single seed at a time, versus choosing multiple seeds. In most cases, the latter incurred additional overhead, without helping improve the performance. Therefore, for all experiments described in this paper, we chose a single seed whenever BDD learning was invoked.

2.2 Creation of BDDs

Once seed selection is done, we need to create BDDs that capture relationships among variables in the circuit region around the seed. We explored two different region heuristics – the fanin cone of the seed, and the circuit region around the seed including its fanins and fanouts. Since the latter results in BDDs that may not include the seed variable at all, our experimental results were uniformly better with the fanin cone heuristic.

In both cases, we created BDDs across very few logic levels, typically 5-10, in order to avoid BDD size blowup. Keeping the number of logic levels any lower would likely result in local learning, which is relatively easy to infer from the circuit constraints. The potential benefit of BDD Learning is in its ability to perform non-local learning around the seed. Apart from avoiding memory blowup, keeping the BDD sizes small has the added benefit of creating shorter paths in the BDD, thereby resulting in shorter clause lengths. In general, shorter learned clauses are likely to be more beneficial than longer learned clauses, since they require less number of assignments before resulting in an implication.

2.3 Generation of Learned Clauses

After a BDD has been created, we need to generate learned clauses. In order to directly use standard BDD packages [23], we can complement the given BDD, and simply enumerate its cubes, i.e. paths to the "1" (true) node. In order to favor shorter clauses, only those cubes that are shorter than a given maximum clause length, typically 5 - 10, are used for generating learned clauses. To avoid exploring all paths (potentially exponential) in the BDD, we enumerate only a fixed number of cubes.

An alternate method, which actually performed better in our experiments, is to implement our own fixed-depth traversal for the complemented BDD. All paths leading to "1" that are shorter than the maximum clause length are enumerated. At the same time, all paths that are greater than the maximum clause length are changed to lead to "0", thereby resulting in an under-approximated BDD. Since the maximum clause length varies from 5 to 10, this traversal is very fast. An additional strategy is to perform a universal quantification on heuristically chosen variables in the under-approximated BDD. This corresponds to performing a resolution on the learned clauses, and results in less number of learned clauses. However, in our experimental results, this typically performed worse than the fixed-depth traversal without quantification.

3. Integrating BDD Learning in SAT Solver

This section describes details of integrating a BDD Learning engine with a typical DPLL-style SAT solver [21], and highlights the issues for static and dynamic learning.

3.1 Invoking the BDD Learning Engine

We invoke the BDD Learning engine at every decision level, including the starting level 0, just before the next decision variable is chosen by the SAT solver. For static learning, we perform learning at decision level 0 only, and disable learning at all other levels (by using the condition `ready_for_learning` shown in Figure 2). For dynamic learning, invoking the BDD Learning engine at every decision level allows information regarding seed selection heuristics to be updated easily (shown in Figure 2 as `update_engine_info`). However, we do not necessarily perform BDD learning at each decision level. Since difficult SAT problems are characterized by an increased number of backtracks, we perform BDD learning after every interval during which a certain number of backtracks has taken place. We also experimented with increasing the backtrack interval parameter dynamically (as a kind of backoff), so as to not over-burden the SAT solver. For more difficult problems, this worked better than a fixed parameter.

3.2 Adding Learned Clauses to SAT Solver

For static learning, the learned clauses generated by the BDD Learning engine are added to the SAT solver before any decisions are made. This is relatively straight forward, since all implications due to the learned clauses occur at the starting level. However, the

situation is somewhat more complicated for dynamic learning, i.e. when learned clauses are added dynamically to the SAT solver.

Note that a conflict clause, i.e. a clause learned from conflict analysis by the SAT solver, is also added dynamically. However, a conflict clause is guaranteed to be either conflicting (or unit, depending upon the implementation) when it is added. This results in an immediate backtrack (or an implication at the current decision level). When multiple clauses are added after BDD Learning, or any other kind of learning performed externally with respect to the SAT solver, extra work may be required to maintain its decision level invariants.

Consider the effect of a newly added clause on the existing state of the SAT solver. The effect depends on the status of the clause, as computed with respect to the current variable assignment stack in the SAT solver, as follows:

- If a learned clause is *conflicting*, i.e. all its literals are false, then the clause can be added immediately. As soon as it is added, conflict analysis will take place, resulting in appropriate action in the SAT solver.
- If a learned clause is *unsatisfied*, but has at least two unassigned literals, then it can be added immediately without changing the decision level of the SAT solver.
- If a learned clause is *unit*, i.e. all but one of its literals are false, and the remaining one is unassigned, then adding it would cause an implication. This might require backtracking up to the level where the implication should be made. Therefore, a choice exists between adding the clause immediately, followed by potential backtracking, or delaying its addition until the SAT solver goes back to the level where the implication should be made. Note that this case also applies to non-conflicting 1-literal clauses, for which backtracking up to the starting decision level (a restart) would be required.
- Finally, if a learned clause is *satisfied*, i.e. at least one of its literals is true, we can add it immediately to the SAT solver without changing the decision level in most cases. The exception is the case when all literals but one are assigned false, and the true literal is the only literal assigned at the highest decision level, i.e. the learned clause would have caused an implication on the true literal at a lower decision level. We call this a *pseudo-satisfied* learned clause. It is similar to the case of a unit clause, and is handled in the same way.

To summarize, learned clauses that are conflicting, or are unsatisfied with at least two unassigned literals, can be added immediately to the SAT solver. Satisfied clauses, but not pseudo-satisfied clauses, can also be added immediately. Finally, unit clauses and pseudo-satisfied clauses require implications to be made. For these, we have a choice between adding them immediately followed potentially by backtracking, or waiting to add them later at the correct implication level.

3.3 Heuristics for Adding Learned Clauses

We use some additional filters to determine whether or not to add a clause generated by the BDD Learning engine to the SAT solver. First, we prefer those learned clauses that capture non-local learning, in order to avoid duplication of circuit constraints. A heuristic that we use is to check if assignments to its literals took place at different decision levels. Though it does not capture non-locality precisely, it is a good indicator. We use the term *non-local* to describe such clauses.

We also use a relevance number to determine the potential usefulness of a learned clause, defined as the sum of its true and unassigned literals. If this number is large, the learned clause is unlikely to be useful, since it will not cause an implication. Therefore, we prefer clauses with a relevance number less than a certain threshold – we call these the *relevant* clauses. (Typically, SAT solvers use a similar figure of merit to delete their own learned clauses when needed.) We found a relevance threshold of 5 to give good performance.

Finally, for unit and pseudo-satisfied learned clauses, we heuristically choose when to add them to the SAT solver. We add them immediately if the difference between the current decision level and the implication level is less than a threshold parameter, but delay adding them otherwise. We typically used a level difference of 5 as the threshold in our experiments.

Based on these additional filters and the status of a learned clause, we have organized the following levels of learning:

- Level 1 learning: adds only the conflict clauses and 1-literal unit clauses
- Level 2 learning: adds all level 1 clauses, and all unit and pseudo-satisfied clauses
- Level 3 learning: adds all level 2 clauses, and all non-local, relevant clauses (satisfied, as well as unsatisfied).

Note that these levels are organized intuitively, according to the projected usefulness of a learned clause. Furthermore, since each level includes clauses added by previous levels, we can easily investigate the effect of adding more clauses.

4. Experimental Results

We have implemented a prototype BMC framework within our in-house verification platform called DiVer. It uses an efficient hybrid SAT solver [24] at the backend, which performs better than Chaff [7] on many problems. We have also implemented a prototype BDD Learning engine, based on CUDD [23] and VIS [25], which has been integrated with our backend SAT solver.

For our experiments, we used six industrial designs, ranging in size up to 416k gates and 12.7k flip-flops in the cone of influence of the correctness property. We used BMC to check safety properties, i.e. the search was for simple counterexamples without loops. So far, we have found a counterexample for only one of them (design D6). Experiments for all designs except D1, were performed on a 2.2 GHz Dual Xeon processor machine, with 4 GB memory, running Linux 7.2. Experiments for D1 were performed on a 900 MHz Dual Sun 220R machine, with 4 GB memory, running Solaris 5.8.

4.1 Static BDD Learning

We first experimented with static BDD Learning. For our experiments, we used a maximum clause length of 6, and the seeds were the top 20 variables ranked by the SAT scoring mechanism for ordering decision variables, before any decisions are made. The results are shown in Table 1. The last design in the table, D6, has a counterexample at time frame 56. For D5, we are actually able to go much deeper than 12 time frames, but it is useful to stop earlier in the experiments because time frame 10

Design	#FF/#G	VIS BMC		DiVer BMC		DiVer BMC with Static BDD Learning												Status
						Level 1 Learning				Level 2 Learning				Level 3 Learning				
		k	Time (s)	k	Time (s)	k	Time (s)	L (s)	#LCl	k	Time (s)	L (s)	#LCl	k	Time (s)	L (s)	#LCl	
D1	12.7k/416.1k	8	1906	96	10230	95	9965	33	4	95	9898	32	5	90	9207	21	1096	worse
D2	4.2k/37.8k	30	802	64	7519	101	10684	15	159	87	10344	11	136	42	6867	15	1720	better
D3	5.2k/46.4k	29	8092	32	8667	32	7720	5	44	32	6168	5	68	32	8704	4	988	better
D4	910/18k	57	10462	89	9760	86	10204	24	105	86	10192	23	105	93	10252	28	1716	better
D5	377/19.4k	12	5868	12	109	12	109	1	0	12	109	1	0	12	155	1	44	same
D6	952/18.1k	56	9134	56	29	56	120	4	34	56	120	3	34	56	237	3	496	worse

Table 1: Experiments for Static BDD Learning show Mixed Results

presents the most difficult SAT problem for this design. For the remaining four designs, we used a time limit of 3 hours for all experiments. Also, in all our result tables, the time reported is for checking *all* depths up to the specified one, i.e. it is the *cumulative* time up to that depth.

In Table 1, Column 2 lists the number of flip-flops (#FF) and gates (#G) in the cone of influence of the property. The next two columns report results for BMC implemented in VIS [25], which uses Chaff [7] as the backend SAT solver – Column 3 reports the maximum depth searched (k), and Column 4 the total time taken (Time, in seconds). Columns 5 and 6 report the same for basic BMC in our platform DiVer, i.e. without the use of any BDD Learning. The next three sets of columns report the results for DiVer BMC with static BDD learning. For each of the three different levels of learning (Section 3.3), we report the maximum depth searched (k), the total time taken (Time, in seconds), the time spent in learming (L, in seconds), and the number of learned clauses added (#LCl), respectively. The last column indicates whether DiVer BMC with static BDD Learning performed better than, worse than, or the same as basic DiVer BMC.

Note first that the basic BMC performance in DiVer is better than the BMC performance in VIS for all designs, by orders of magnitude for many of them. This indicates that basic DiVer BMC is a good baseline for comparison. Next, note that for four of six designs, static BDD Learning is better or the same as basic BMC, i.e. it improves the runtime, and in some cases allows a deeper search. The learning time itself is quite low in all cases. Comparing the different levels of learning, for five of six designs, the impact of adding more clauses is negative. This can be seen clearly for design D2, where the maximum depth varied from 101 for Level 1, 87 for Level 2, down to 42 for Level 3. This shows the importance of learning "useful" clauses, while keeping their overheads low. On the other hand, for design D4, it was indeed more useful to add more clauses.

4.2 Dynamic BDD Learning

Next, we experimented with dynamic BDD Learning. In general, our results were worse for the combination of static and dynamic learning. Therefore, we report results for dynamic learning alone. For our experiments, we performed learning after every 100 backtracks, and used a maximum clause length of 6.

We first experimented with different levels of learning. Unlike static learning, we found that results for Level 3 learning were uniformly better for all designs. This indicates that when the seeds have been selected well, hopefully due to dynamic seed selection, it is better to learn more than to learn less. For seed selection, we tried heuristics SSH1—SSH5 (Section 2.1).

Table 2 reports the results for Level 3 learning, with the best seed selection heuristic for each design. Columns 1 – 4 are as before; the remaining columns report the results for DiVer BMC with dynamic BDD Learning. Columns 5 and 6 report the depth and time taken for the same depth as basic DiVer BMC, and Column 7 reports the percentage reduction in time. Columns 8 and 9 show the maximum depth, and the total time taken. Columns 10—13 report the learning statistics – the time spent in learning (L, in seconds), the number of learned clauses added (#LCl), the number of seeds used (#Seeds), and the seed selection heuristic that gave the best result (SSH), respectively. The last column indicates whether DiVer BMC with dynamic BDD Learning performed better than, worse than, or the same as basic DiVer BMC.

Note that dynamic BDD Learning is quite effective in all designs. In comparison to basic BMC, it reduced the runtime by up to 73%, and allowed deeper searches to be completed within the allotted time. The learning time is again quite low. More interestingly, note that even a small number of added clauses can impact the overall performance significantly. For example, in design D1, the addition of just 15 learned clauses for 3 seeds, achieved a 25% reduction in run time for completing depth 96, and allowed an additional 13 time frames to be searched. This demonstrates the effectiveness of our heuristics for choosing seeds and learned clauses to be added. Our extended results indicate that the past decision heuristic (SSH2) gives good performance in general, though it may not consistently give the best performance.

In comparison to static BDD Learning, we obtained significantly better results (either less time, or increased depth, or both) with dynamic BDD Learning for four of six designs, and it was not much worse for the remaining two. While the time for learning itself is insignificant, the difference is due to the quality of the clauses added, and their impact on future decisions. In particular, for the same level of learning (Level 3), less number of clauses are added by dynamic learning than by static learning. In a sense, this indicates that dynamic seed selection offers a better control than static seed selection over the tradeoff in adding learned clauses.

Design	#FF/#G	DiVer BMC		DiVer BMC With Dynamic BDD Learning									Status
		k	Time (s)	k	Time (s)	Reduction	Max k	Time (s)	L (s)	#LCl	#Seeds	Best SSH	
D1	12.7k/416.1k	96	10230	96	7646	25%	109	10644	1	15	3	SSH2	better
D2	4.2k/37.8k	64	7519	64	2031	73%	103	10459	11	2361	655	SSH4	better
D3	5.2k/46.4k	32	8667	32	7195	17%	32	7195	3	1494	276	SSH2	better
D4	910/18k	89	9760	89	7379	24%	92	10791	20	1464	784	SSH2	better
D5	377/19.4k	12	109	12	39	64%			1	89	32	SSH5	better
D6	952/18.1k	56	29	56	29	0%			0	0	0	SSH4	same

Table 2: Experimental Results for Dynamic BDD Learning show Significant Improvements

We also explored the effect of varying the maximum clause length. These results are shown in Table 3. The performance variation is shown in the maximum depth searched, or in the total time (where the maximum depth did not vary). A maximum clause length of 6 or 7 gave the best empirical results on all designs. This may be related to the BDD creation parameters we have used, and remains to be investigated further.

Design-feature	Maximum Clause Length				
	5	6	7	8	9
D1-depth	94	109	98	92	95
D2-depth	49	103	41	72	41
D4-depth	88	92	91	89	89
D3-time	8970	7195	6941	7384	7328
D5-time	197	39	36	36	84

Table 3: Performance Variation across Maximum Clause Length

5. Conclusions

SAT-based BMC is an effective technique for finding bugs in large designs. Its performance is critically determined by the practical efficiency of the backend SAT solver. In this paper, we have described details of a lightweight and effective BDD Learning technique, which adds learned clauses generated from BDDs to supplement other learning mechanisms in a SAT solver. We explored both static and dynamic learning using BDDs. The various heuristics and parameters in our BDD Learning engine are targeted at increasing the usefulness of learned clauses, while reducing the inherent overheads. We have demonstrated the effectiveness of our techniques on several industrial designs, where we have obtained up to 73% reduction in runtime, allowing us to perform deeper searches within the allotted time. We believe that the BMC verification framework provides many opportunities for combining the relative benefits of SAT and BDDs, and our work is a step in that direction.

References

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*: MIT Press, 1999.

[2] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*: Kluwer Academic Publishers, 1993.

[3] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35(8), pp. 677-691, 1986.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proceedings of Workshop on Tools and Algorithms for Analysis and Construction of Systems (TACAS)*, vol. 1579, *LNCS*, 1999.

[5] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506-521, 1999.

[6] H. Zhang, "SATO: An efficient propositional prover," in *Proceedings of International Conference on Automated Deduction*, vol. 1249, *LNAI*, 1997, pp. 272-275.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of Design Automation Conference*, 2001.

[8] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT-Solver," in *Proceedings of Conference on Design Automation and Test Europe (DATE)*, 2002, pp. 142-149.

[9] O. Strichman, "Pruning Techniques for the SAT-based bounded model checking," in *Proceedings of Workshop on Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2001.

[10] G. Anderson, P. Bjesse, B. Cook, and Z. Hanna, "A Proof Engine Approach to Solving Combinational Design Automation Problems," in *Proceedings of the DAC*, 2002.

[11] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*: Computer Science Press, 1990.

[12] J. Jain, R. Mukherjee, and M. Fujita, "Advanced Verification Techniques based on Learning," in *Proceedings of the Design Automation Conference,* 1995.

[13] W. Kunz, D. Pradhan, and S. M. Reddy, "A Novel Framework for Logic Verification in a Synthesis Environment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 20-32, 1996.

[14] A. Gupta and P. Ashar, "Integrating a Boolean Satisfiability Checker and BDDs for Combinational Verification," in *Proceedings of the VLSI Design Conference*, 1998.

[15] J. R. Burch and V. Singhal, "Tight integration of combinational verification methods," in *Proceedings of International Conference on Computer-Aided Design*, 1998.

[16] A. Kuehlmann, M. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *Proceedings of Design Automation Conference*, 2001.

[17] A. Gupta, Z. Yang, P. Ashar, and A. Gupta, "SAT-based Image Computation with Application in Reachability Analysis," in *Proceedings of Conference on Formal Methods in Computer-Aided Design*, 2000, pp. 354-371.

[18] P. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking," in *Proceedings of International Conference on Computer-Aided Verification*, vol. 1855, *LNCS*, 2000, pp. 124-138.

[19] P. A. Abdulla, P. Bjesse, and N. Een, "Symbolic Reachability Analysis based on SAT-Solvers," in *Proceedings of Workshop on Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2000.

[20] A. Kuehlmann and F. Krohm, "Equivalence Checking using Cuts and Heaps," in *Proceedings of DAC*, 1997.

[21] M. Davis, G. Longeman, and D. Loveland, "A Machine Program for Theorem Proving," *Communications of the ACM*, vol. 5, pp. 394-397, 1962.

[22] J. Pilarski and G. Hu, "SAT with Partial Clauses and Back-Leaps," in *Proceedings of DAC*, 2002, pp. 743-746.

[23] F. Somenzi, "CUDD: University of Colorado Decision Diagram Package, http://vlsi.colorado.edu/~fabio/CUDD/," 1998.

[24] M. Ganai, L. Zhang, P. Ashar, and A. Gupta, "Combining Strengths of Circuit-based and CNF-based Algorithms for a High Performance SAT Solver," in *Proceedings of the Design Automation Conference*, 2002.

[25] R. Brayton, F. Somenzi, and others, "VIS: Verification Interacting with Synthesis, http://vlsi.colorado.edu/~vis," 2002.