Integrating CNF and BDD Based SAT Solvers

Sivaram Gopalakrishnan, Vijay Durairaj and Priyank Kalla
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT-84112
{sgopalak, durairaj, kalla}@ece.utah.edu

ABSTRACT: *This paper presents an integrated infrastructure of CNF and BDD based tools to solve the Boolean Satisfiability problem. We use both CNF and BDDs not only as a means of representation, but also to efficiently analyze, prune and guide the search. We describe a method to successfully re-orient the decision making strategies of contemporary CNF tools in a manner that enables an efficient integration with BDDs. Keeping in mind that BDDs suffer from memory explosion problems, we describe learning-based search space pruning techniques that augment the already employed conflict analysis procedures of CNF tools. Our infrastructure is targeted towards solving those hard-to-solve instances where contemporary CNF tools invest significant search times. Experiments conducted over a wide range of benchmarks demonstrate the promise of our approach.*

I. Introduction

Boolean Satisfiability (SAT) is the problem of finding a solution (if one exists) to the equation $f = \mathbf{1}$, where f is a Boolean formula to be satisfied. The formula (f) can be represented in Conjunctive Normal Form (CNF), or with Binary Decision Diagrams (BDDs) [1]. Contemporary SAT tools [2] [3] [4] [5] [6] [7] use either representation, but *usually* not both, to solve the problem. This paper describes a framework that tightly integrates both CNF-SAT and BDD-based methods to solve the SAT problem. Both CNF and BDDs are used not only as a means to represent the constraints, but also to efficiently analyze, prune and guide the search.

Classical approaches to CNF-SAT are based on variations of the well known Davis-Putnam (DP) [8] and Davis-Logemann-Loveland (DLL) [9] procedures. Recent approaches [2] [3] [4] [5] etc., additionally employ sophisticated methods such as constraint propagation and simplification [10], conflict analysis [2], learning and non-chronological backtracks to efficiently analyze and prune the search space.

Binary Decision Diagrams (BDDs) [1] [11] have also been used to solve the SAT problem. Early BDD-based SAT approaches [12] [13] modeled the overall constraints (all clauses of the CNF formula) using a monolithic BDD that represents all possible solutions. However, storing the product of all constraints within a monolithic BDD is memory-wise infeasible. To overcome this problem, [6] and [7] proposed to store the constraints in partitions; a BDD is used to represent the conjunction of all the clauses within a partition. Memory explosion problem, however, still restricts the scalability of these techniques.

Zero-suppressed BDDs (ZBDDs) [14] have also been used for SAT in [15] [16] [17]. However, ZBDDs also suffer from size explosion problems, particularly when constraints are *not* sparsely populated. Moreover, ZBDDs are not as manipulable as ROBDDs.

There have been a few attempts to integrate CNF and BDD-based methods. However, the objective has not always been a dedicated SAT search. The techniques of [18] [19] [20] integrate CNF-SAT tools and BDDs for equivalence checking, while that of [21] uses both CNF and BDDs for reachability analysis computations. The work of [22] uses BDDs only to represent the clauses; the search is performed using the DP procedure.

Contemporary CNF-SAT tools have had little difficulty in solving large problem instances that have plenty of solutions distributed throughout the search space. Even for instances that have no solutions, these tools are quick to evaluate infeasibility, as they employ highly accomplished conflict analysis and search space pruning methods. However, for difficult-to-solve instances that have few feasible solutions, these tools spend considerable amount of time searching for a solution.

BDDs, on the other hand, provide powerful implication analysis due to their implicit representation and efficient manipulation capabilities. This enables them to tackle those problems that have a "hard-core". Unfortunately, their use is rather limited due to the size explosion problem. It has been observed that so long as the BDD size can be contained, solutions can be found quickly, often with predictable run-times [6] [7] [23]. If, for relatively large hard-to-solve instances, CNF tools invest a large amount of time, and BDDs run into memory explosion, how can their respective capabilities and limitations be leveraged to efficiently search for a SAT solution?

With the above issues in mind, we present a tightly integrated infrastructure of CNF and BDD-based SAT engines. Our framework not only retains the conflict-based learning power of CNF-tools, but also augments it with a new

learning-based search-space (and BDD-size) pruning technique. Modifications to conventional CNF-search procedures are devised that resolve a large number of variables as well as simplify the subsequent problem for BDDs. Combining CNF and BDD-based methods allows to leverage their respective powers of search, implication and conflict resolution, time versus memory constraints, etc. We have implemented the BDD-based SAT technique (called SSP-BSAT) using the CUDD [24] library and integrated it with the CNF-based tool CHAFF [4] to run our experiments.

II. The Integrated Framework: A First Look

The general framework describing our integration is depicted in Fig. 1. The input to our tool is the CNF formula to be satisfied, represented as a clause list. The search begins by using the CNF-SAT tool to perform decisions and subsequent implications. Once the search results in a pre-computed number of variable assignments (a pre-defined threshold), the remaining *unresolved clauses* are then given to BDDs as constraints to be satisfied further. BDDs can then be used to store the conjunction of all the clauses If the BDDs provide the solution, then the assignments made by CNF solver, together with those of BDDs, comprise the satisfying solution. If the BDDs return a "no-solution" for the unresolved clauses, the process backtracks to the CNF-SAT solver at the corresponding decision level with a conflict. The process thus goes back-and-forth between CNF and BDDs, until a SAT solution is found (either by CNF or by BDDs), or UNSAT is proved.

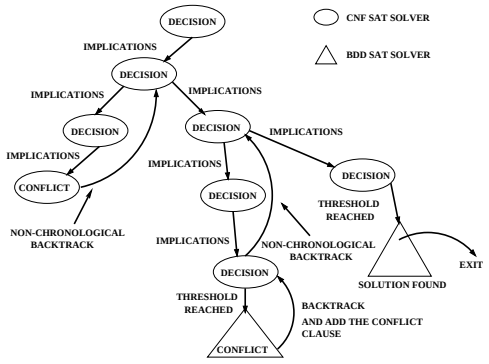

Fig. 1. CNF+BDD Integration: The Overall Framework.

In order to achieve the above described integration efficiently, we address the following issues:

1. How should the search be oriented in the CNF solver such that while not only do a large number of variables get resolved, but the subsequent problem for BDDs also gets simplified?
2. At what point should the search be transferred from CNF to BDDs? In other words, what should be the "threshold" that decides constraint transfer from CNF to BDDs?
3. Once BDDs are invoked to satisfy the unresolved clauses, how do we perform the search so as to contain the growth of BDD size? What kind of learning and search space pruning techniques should be employed for the same?
4. How should the information learned through conflict analysis be transferred and utilized across the CNF and BDD domains?

Subsequent sections of the paper answer the above questions.

III. Modifications for CNF-SAT

The search process in CNF solvers generally begins by selecting a variable and deciding upon a corresponding assignment (0 or 1) to that variable. The decision strategy greatly impacts upon the performance of SAT tools. In [25], it was argued that the Dynamic Largest Individual Sum (DLIS) heuristic proved to be a good all-round strategy. DLIS selects a literal (x or $\overline{x}$) that appears most frequently in unresolved clauses. In the development of CHAFF [4], the decision strategy implemented was the Variable State Independent Decaying Sum (VSIDS) heuristic.

In order to derive a good decision strategy for the CNF-part of our integration, we analyzed the distribution of the number of occurrences (activity) of variables in the clauses for various benchmarks from the DIMACS suite [26]. Two particular distributions corresponding to benchmarks *ii16b2* and *hanoi4* are shown in Fig. 2. It can be observed from the figures that some variables occur in very few clauses. We refer to such variables as being *locally distributed*. Further, some variables appear in a large number of clauses. Such variables are referred to as being *globally distributed*.

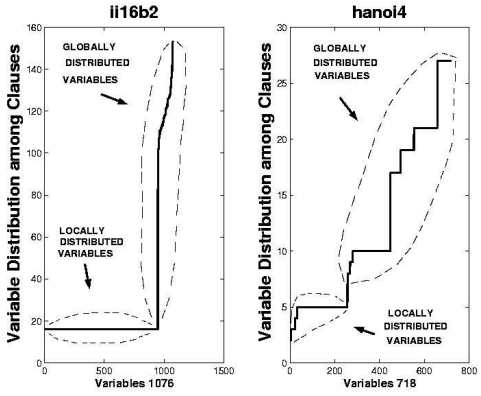

Fig. 2. Distribution of Variables among Clauses

First, let us consider the *ii16b2* example that has a large number of locally distributed variables and a smaller number of globally distributed variables. Using a DLIS-type strategy, if we resolve the globally distributed variables first, then the resulting partially unresolved clauses would mostly contain locally distributed variables. Subsequently, using BDDs to construct the "product" (conjunction) of these unresolved clauses causes memory-explosion. This is because if functions $f_1, f_2 \ldots, f_n$ have non-intersecting support variables, then the BDD-size for the iterative conjunction of all f_i con-

tinues to increase. However, if they have common support, then the size of the product is often observed to be smaller [11]. Hence, resolving these locally distributed variables first (reverse DLIS) enables better performance of BDDs.

Now let us consider the *hanoi4* example which has significantly more globally distributed variables than locally distributed ones. Resolving the few locally distributed variables first using CNF-SAT may not be beneficial in this case. Intuitively, resolution of just a few variables would imply that the CNF tool has not done enough work to solve its part of the problem. Subsequently, the resulting subproblem for BDDs may potentially be almost as difficult as the original one. In such a case, we would rather that the CNF tool invest some effort to resolve those variables that have high activity.

Observations from Preliminary Experiments: To analyze the effect of branching strategies w.r.t. global versus local variables, we performed preliminary experiments with a rudimentary integration of CNF-SAT and BDDs. In one experiment, first the locally distributed variables were resolved using CNF-SAT, and the remaining problem was given to BDDs. In another experiment, CNF-SAT was used to resolve global variables first and BDDs for the rest. Some results are shown in Table I. Benchmarks *ii16b2* and *ii16a1* have a larger number of locally distributed variables. In this case, our strategy of using CNF-SAT to resolve local variables first pays rich dividends as it simplifies the problem for BDDs. However, the opposite is true for benchmarks *hanoi4, NQueens15*; this is because these benchmarks have very few locally distributed variables.

TABLE I
IMPACT OF DECISION STRATEGY

Benchmark	Vars/Clauses	CNF - local vars BDD - global vars	CNF - global vars BDD - local vars
ii16b2	1076/16121	0.12 sec	4.7 sec
ii16a1	1650/19368	0.41 sec	0.6 sec
hanoi4	718/4934	91.1 sec	1.31 sec
Nqueen15	225/5195	49.32 sec	17.53 sec

Quantitative Assessment of Global vs Local Activity: So far, we have treated the variable activity (locally versus globally distributed) only on *qualitative* terms. Now we describe a *quantitative* measure to classify a variable as being locally or globally distributed. First, we identify the variable(s) that have the highest activity. For example, in benchmark *hanoi4* there are 718 variables. Let these variables be $x_0, x_1, \ldots, x_{717}$. Let variable x_{717} have the highest activity. From Fig. 2, activity of x_{717} is 27. If the activity of any variable is less than 20% of the highest activity, then it is classified as being locally distributed; globally distributed otherwise. For example, variable x_{100} in *hanoi4* has activity = 5. Since $5 < 20\%$ of 27, x_{100} is classified as being locally distributed.

Deriving the CNF-SAT Decision Strategy: The experiments in Table I justify our intuition regarding branching strategies and provide guidelines to orient our search according to the variable activity. If more than 50% of the variables are locally distributed, then we employ CNF-SAT to first resolve the local variables and transfer the remaining problem to BDDs. Otherwise, we employ CNF-SAT to first resolve the globally distributed variables and then use BDDs.

It has been observed in [4] [5] that difficult problems generate a significant number of conflicts. Hence, importance has to be given to the knowledge gained from the appended conflict-clauses. This is another issue (akin to CHAFF's VSIDS heuristic) that we want to model in the search process of our solver. On encountering conflicts, we allow the CNF-part of our solver to add conflict-induced clauses and proceed with its book-keeping and backtracking procedures.

Constraint Transfer from CNF to BDDs: Constraint transfer from CNF to BDDs occurs once a pre-defined "threshold" is reached. We have experimentally observed that if we use a monolithic BDD to construct and represent the conjunction of all the clauses, then a threshold value of 100 *unresolved variables* appears to be most suitable for an optimal performance. Otherwise, we have observed with most benchmarks that the BDD-product generally tends to explode for a larger number of variables. Unfortunately, a threshold value of 100 unresolved variables for BDDs is too less to have any significant performance impact even for moderately sized problems. To overcome this limitation, the next section presents a new BDD-based method to efficiently store a larger set of constraints and perform the search while intelligently pruning the BDD-size.

IV. BDD-Based SAT with Search Space Pruning

Once the constraints are transferred to BDDs as unresolved clauses, we represent them as implicitly conjoined BDDs [27] ($f_1 \cdot f_2 \cdot \ldots \cdot f_n$), and then proceed to search for a solution. To account for BDD-size explosion, we need to generate sub-functions with smaller BDD-size. This can be achieved by performing orthonormal (Shannon's) expansion of these BDDs w.r.t. a variable.

Let f be a Boolean function and x be a variable in its support. According to Shannon's expansion: $f = x \cdot f_x + \overline{x} \cdot f_{\overline{x}}$, where $f_x = f(x = 1)$ and $f_{\overline{x}} = f(x = 0)$ are positive and negative co-factors of f, respectively. Thus, if $f \cdot g$ is the SAT problem, then the solution can be found in their respective cofactors according to the following computation: $f \cdot g = x \cdot f_x \cdot g_x + \overline{x} \cdot f_{\overline{x}} \cdot g_{\overline{x}}$. The significance of this expansion is that BDD-size of the cofactors is smaller than that of the original functions. Thus, in the context of SAT search, successive application of Shannon's expansion w.r.t. new variables would lead to sub-functions with smaller BDDs, whose product can possibly be built.

Another important aspect of Shannon's expansion, cofactor computations and our BDD-based search needs mention. For a Boolean function f, if $f_x \supseteq f_{\overline{x}}$ ($f_x \subseteq f_{\overline{x}}$), then f is said to be positive (negative) *unate* in variable x. Otherwise f is *binate* in x. For an efficient search using Shannon's expansion, the choice of splitting variable is important. Brayton *et al.* in [28] proposed a heuristic that consists of selecting *binate* variables because successive application of Shannon's expansion w.r.t. binate variables most likely results in unate sub-problems. SAT search on unate sub-problems is much easier [28][6].

We follow a similar approach. For each input variable x, we evaluate how many functions in the constraint set are binate in x. We select that variable over which most functions are binate. We define this variable as the **most active binate** variable. After expanding all the functions $f^1, \ldots, f^n$ over x, we search for the solution in the corresponding cofactors. The expansion is performed until there are no more binate variables or the (user-defined) co-factoring depth has been reached. At this point, the BDD product can be built to find the solution. If the solution is found, our job is completed. If the solution is not found, then we learn that the current region of the search space being explored corresponds only to non-solution points. Subsequently, we prune the search space by eliminating these non-solution points from the original constraint set.

Search Space Pruning: Let $Z = f^1 \cdot f^2 \cdot f^3$ correspond to the constraint set to be satisfied. Let a be the chosen decomposing variable. Cofactoring Z w.r.t. a corresponds to $Z_a = f^1_a \cdot f^2_a \cdot f^3_a$. Let b be the next variable over which orthonormal expansion is performed. This results in $Z_{ab} = f^1_{ab} \cdot f^2_{ab} \cdot f^3_{ab}$. Assuming that BDD size reduces significantly, the product of $f^1_{ab} \cdot f^2_{ab} \cdot f^3_{ab}$ is computed. If this corresponds to a no-solution, then it implies that assignments $a = 1$ and $b = 1$ lead to non-solution points. These points can now be eliminated (pruned) from the constraint set, individually, by virtue of the following theorem.

Theorem 1: Let f and g be two functions with variables x and y in their support. It is required to find a satisfying solution for the product $f \cdot g$. Assume that there exists a solution for $f \cdot g$. If the solution is not found with the assignments x = 1 and y = 1 then it implies that the solution is contained in $(f \cdot g \cdot \overline{xy})$.

Proof: Shannon's expansion on $f \cdot g$ w.r.t. x, y results in:

$$f \cdot g = xy f_{xy} g_{xy} + \bar{x}y f_{\bar{x}y} g_{\bar{x}y} + \bar{y}x f_{\bar{y}x} g_{\bar{y}x} + \bar{y}\bar{x} f_{\bar{y}\bar{x}} g_{\bar{y}\bar{x}} \quad (1)$$

From the above equation, it can be concluded that if the solution does not exist with the assignments $x = 1$ and $y = 1$, then all the points pertaining to the assignment $x = y = 1$ can be removed from the search space composed of $f \cdot g$. Subsequently, the constraints can be pruned using set difference: $\tilde{f} \cdot \tilde{g} = f \cdot g - x \cdot y = (f \cdot g) \cap (\overline{x \cdot y}) = (f \cdot (\overline{x \cdot y})) \cdot (g \cdot (\overline{x \cdot y}))$ Substituting Eqn. (1) in (2) results in: (2)

$$(f \cdot g) \cap (\overline{xy}) = \bar{x}y f_{\bar{x}y} g_{\bar{x}y} + \bar{y}x f_{\bar{y}x} g_{\bar{y}x} + \bar{y}\bar{x} f_{\bar{y}\bar{x}} g_{\bar{y}\bar{x}} \quad (3)$$

The above equation does not have points pertaining to $x = y = 1$. The updated constraints $\tilde{f} \cdot \tilde{g}$ now contain a smaller search space. The solution can now be found in the updated region. ■

The elimination of the non-solution points corresponding to assignments $a = 1, b = 1$ can be performed by repeated intersection of $(\overline{a \cdot b})$ with the individual constraints (f^i). Note that this intersection may potentially increase the size of a BDD. In such a case, we may not update that particular BDD. Once the constraint set has been updated, the search could be *restarted* with a new constraint set composed of smaller BDDs. The search process would require an additional check to ensure that previously visited assignments are not revisited. This check can be performed by storing all the visited paths also in an ROBDD. Before proceeding into the search, a check can be performed to see if the current path has already been visited. If it has been visited before, then we may backtrack by inverting the polarity of the variable at the current decision level. The above technique has been programmed in an algorithm SSP-BSAT described in Algorithm 1.

```
SSP-BSAT(f_list = f^1, ..., f^n)
while !solution AND all constraints non-zero AND visited_path != 1 do
  /* x = next most binate variable */
  while binate variables OR cofactoring depth not reached do
    if no more binate variables then
      build the product of (f_list) and return
    end if
    if path not visited then
      path = path AND x; cof_f_list = f^1_x, ..., f^n_x;
    else
      path = path AND x̄; cof_f_list = f^1_x̄, ..., f^n_x̄;
    end if
  end while
  /* Either unateness or cofactoring depth reached */
  result = product of constraints in cof_f_list;
  if solution found then
    return (result AND path);
  end if
  /* Non-solution, prune the search space */
  pruned_f_list = f_list AND (path)'
  update visited path;
  restart search with f_list = pruned_f_list;
end while
```

Algorithm 1: SSP-BSAT: BDD-SAT & search space pruning.

Theorem 2: The algorithm SSP-BSAT converges to a solution, if one exists. If a solution does not exists, SSP-BSAT proves unsatisfiability.

Proof: The correctness and convergence of SSP-BSAT is trivially proved. Successive removal of non-solution points

would: (i) render one or more functions in the constraint set empty, implying an infeasible instance; or (ii) result in smaller BDDs whereby their product can be easily built. Moreover, if all the paths have been visited and the solution has not been found, it implies infeasibility. Furthermore, if a solution exists, the search will always find it. ■

Table II presents some results that demonstrate the power of SSP-BSAT *viz-a-viz* CHAFF. Since the effectiveness of BDDs is limited by the number of variables, we examine some of the smaller benchmarks. For the *pigeonhole* benchmarks, iterative pruning results in one or more functions (BDDs) in the constraint set becoming empty. For this reason, SSP-BSAT proves unsatisfiability much faster than CHAFF. For multipliers and barrel shifters, SSP-BSAT was able to find a solution within the first few iterations.

TABLE II
CPU TIME COMPARISON: SSP-BSAT VS CHAFF

Benchmark	Vars / Clauses	Chaff	SSP-BSAT
hole-8	72 / 297	0.67s	0.03s
hole-9	90 / 415	3.45s	0.04s
hole-10	110 / 561	19.06s	0.24s
aim100-2-no2	100 / 200	0.01s	0.01s
bshift32	1512 / 3529	0.03s	0.01s
bshift64	3492 / 8218	0.1s	0.01s
11×11 mult	7556 / 17208	1.05s	0.01s

V. Efficient Implementation of CNF+SSP-BSAT

For efficient and tight integration of CNF + SSP-BSAT, we have implemented further enhancements to the CNF and SSP-BSAT engines that are described below.

BDD-product Computations: Once the constraints are transferred from CNF to BDDs as unresolved clauses, we do not directly invoke SSP-BSAT. First, we attempt to construct the conjunction of all the constraints within a monolithic BDD. If, during intermediate conjunction operations, the BDD-size exceeds $500K$ nodes, we abort the computation and then invoke SSP-BSAT. Furthermore, when storing the constraints (clauses) as partitioned BDDs, we ensure that the size of each BDD does not increase beyond $5K$ nodes. We have experimentally observed that these BDD-size values provide optimal performance.

Search Process in SSP-BSAT: The search process in SSP-BSAT relies on iterative computations of most active binate variables. As the number of variables in the unresolved constraint set increases, this computation becomes very time-consuming. Alternatively, CNF-tools also compute the order of variables (corresponding to variable activity) over which they perform decisions. We have replaced the computation for the most active binate variables in SSP-BSAT with this pre-computed order to perform the search. Thus, along with the transfer of unresolved constraints from CNF to BDDs, we transfer this order of variables too. As a result, we have observed an order of magnitude speedup in SSP-BSAT alone.

BDD-size Reduction through Constraining It is desirable to construct the product of constraint BDDs as soon as possible to avoid large recursion depths. To achieve this goal, the size of the constraint BDDs needs to be reduced in the early stages of the search. This can be achieved as follows: Let $f \cdot g$ be the SAT problem. According to Shannon's expansion: $f = g \cdot f_g + g' \cdot f_{g'}$. Thus, $f \cdot g = g \cdot (g \cdot f_g + g' \cdot f_{g'}) = g \cdot f_g$. This implies that the SAT problem $f \cdot g$ now reduces to $g \cdot f_g$. Hence, after each pruning step, we select the smallest BDD (f^k) to perform *safe BDD minimization* [29] of all f^i w.r.t. $f^k, i \neq k$. After a few iterations of pruning and BDD-minimization, the size of the BDDs reduces to a point which obviates any further need for iterative co-factor computations. Now the BDD-product can be constructed much earlier in the search.

Conflict Analysis across CNF-BDD Boundary: If the BDD-part of the framework returns a no-solution for the partially unresolved clauses, it implies that the decisions made by the CNF tool caused the conflict. Thus, we can add a corresponding conflict clause to the original constraint database, similar to the partial clause addition techniques of [30].

We have incorporated the above enhancements to the CNF and BDD engines of our integrated solver. For the CNF part, we have implemented the described modifications to the CHAFF tool [4]. For the BDD-part, SSP-BSAT algorithm has been correspondingly finessed. Using the C++ interface of the CUDD package, we have integrated the modified engines towards an efficient SAT tool. Using the integrated SAT solver, we have performed experiments over a wide range of benchmarks. The results are presented in Table III.

Analyzing the Results: CHAFF (using its default VSIDS scheme) and our integrated solver (CHAFF+SSP-BSAT) were used to solve SAT on the examples and the run-time statistics were recorded. For our integrated solver, the constraints were transferred from CNF to BDDs when the number of unresolved variables (threshold) reached 50% of the total variables, or 300, whichever was smaller. As an example, *fpga10_8* has 120 variables. When the number of unresolved variables reached 60 ($50\% of 120 = 60 < 300$), constraints were transferred to BDDs. The recursion depth in SSP-BSAT was fixed at 10. BDD variable re-ordering was *not* used in any experiment.

It can be observed for most of the medium-sized difficult SAT instances, such as *hole9, hole10, ii16b, hanoi5, fpga,* the performance of CHAFF+SSP-BSAT is an order of magnitude better than that of CHAFF alone. Marginally poorer performance of CHAFF+SSP-BSAT for some benchmarks can be attributed to the overhead in transferring constraints from CNF to BDDs and vice-versa. The benchmarks for which CHAFF+BSAT really demonstrates poor performance

are *fpga10_9, fpga12_11, pret150 and 9vliw*. Out of these 9vliw has too many variables for BDDs to be used efficiently. Benchmark *pret150* is an UNSAT instance. Our motivation behind the integration of CNF and BDD-based SAT methods was to solve hard problems that have few solutions. We did not expect our integrated technique to solve very large and unsatisfiable problems. The results seem to vindicate our search strategies.

TABLE III
CPU TIME COMPARISON OF CHAFF VS INTEGRATED SOLVER. CPU TIME IS EXPRESSED IN SECONDS

Bench-mark	Vars/Clauses	Chaff	CNF + SSP-BSAT
hole-8	72 / 297	**0.67**	1.02
hole-9	90 / 415	3.45	**1.7**
hole-10	110 / 561	19.06	**3.02**
ii16a1	1650 / 19368	0.56	**0.32**
ii16a2	1602 / 23281	0.05	0.21
ii16b1	1728 / 24792	96.46	**0.69**
ii16b2	1076 / 16121	3.55	**0.12**
ii32c4	759 / 20862	0.4	**0.30**
par8-1	350 / 1149	**0.01**	0.02
par16-5	1015 / 3358	**0.69**	1.47
pret150_25	150 /400	**0.01**	51.3
hanoi4	718 / 4934	**0.88**	1.31
hanoi5	1931 / 14468	2148	**145.75**
fpga10_8	120 / 448	2.4	**1.49**
fpga10_9	135 / 549	**2.16**	621
fpga12_8	144 / 560	985.57	**106.6**
fpga12_9	162 / 684	1591.44	**291.45**
fpga12_10	180 / 820	1549.14	**133.94**
fpga13_9	176 / 759	>2000	**452.07**
fpga12_11	198 /968	**476.9**	4259.64
9vliw_bug1	19617 / 257010	**62**	717.45

VI. **Conclusions**

This paper has described a framework that tightly integrates both CNF-SAT and BDD-based methods to solve the SAT problem. Both CNF and BDDs are used not only as a means to represent the constraints, but also to efficiently analyze, prune and guide the search towards a solution. The proposed infrastructure has been developed to target difficult-to-solve SAT problems. Modifications to conventional CNF-SAT tools was described, and a new BDD-based SAT method (called SSP-BSAT) was introduced that employs efficient search space pruning. The results demonstrate the applicability of our integrated solver towards solving relatively large problems that have a hard-core.

REFERENCES

[1] R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, vol. C-35, pp. 677–691, August 1986.

[2] J. Marques-Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability", *in ICCAD'96*, pp. 220–227, Nov. 1996.

[3] H. Zhang, "SATO: An Efficient Propositional Prover", *in In Proc. Conference on Automated Deduction*, pp. 272–275, 1997.

[4] M. Moskewicz, C. Madigan, L. Zhao, and S. Malik, "CHAFF: Engineering and Efficient SAT Solver", *in In Proc. Design Automation Conference*, pp. 530–535, June 2001.

[5] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver", *in DATE, pp 142-149*, 2002.

[6] P. Kalla, Z. Zeng, M. J. Ciesielski, and C. Huang, "A BDD-based Satisfiability Infrastructure using the Unate Recursive Paradigm", *in Proc. Design Automation and Test in Europe, DATE'00*, 2000.

[7] R. Damiano and J. Kukula, "Checking Satisfiability of a Conjunction of BDDs", *in DAC, pp 818-823*, 2003.

[8] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory", *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

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

[10] R. Zabih and D. A. McAllester, "A Rearrangement Search Strategy for Determining Propositional Satisfiability", *in Proc. Natl. Conf. on AI*, 1988.

[11] K. S. Brace, R. Rudell, and R. E. Bryant, "Efficient Implementation of the BDD Package", *in DAC*, pp. 40–45, 1990.

[12] B. Lin and F. Somenzi, "Minimization of Symbolic Relations", *in ICCAD*, 90.

[13] S. Jeong and F. Somenzi, "A New Algorithm for the Binate Covering Problem and its Application to the Minimization of Boolean Relations", *in ICCAD*, 92.

[14] S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems", *in DAC*, pp. 272–277, 93.

[15] P. Chatalic and L. Simon, "Multi-resolution on compressed sets of clauses", *in Proc.of 12th International Conference on Tools with Artificial Intelligence (ICTAI)*, Nov 2000.

[16] P. Chatalic and L. Simon, "Zres: the old dp meets zbdds", *in Proc.of 17th Conf. of Autom. Deduction (CADE)*, 2000.

[17] F. Aloul, M. Mneimneh, and K. Sakallah, "Backtrack search using zbdds", *in International Workshop on Logic and Synthesis*, pp. 293–297, June 2001.

[18] J. R. Burch and V. Singhal, "Tight Integration of Combinational Verification Methods", *in ICCAD*, pp. 570–576, 98.

[19] S. Reda and A. Salem, "Combinational equivalence checking using boolean satisfiability and bdds", *in DATE*, 2001.

[20] A. Gupta and P. Ashar, "Integrating a boolean satisfiability checker and bdds for combinational equivalence checking", *in In Proc. Intl. Conf. on VLSI Design, pages 222-225*, 1997.

[21] A. Gupta and *et al.*, "Sat-based image computation with application in reachability analysis", *in Dagstuhl Seminar on Design and Test*, 2001.

[22] T. Uribe and M. Stickel, "Ordered binary decision diagrams and the davis-putnam procedure", *in In J.P. Jouannaud, editor, Ist Intl. Conf. on Constraints in Comp. Logics, volume 845 of LNCS, pp34-4*, September 1994.

[23] T. Villa and *et al.*, "Explicit and Implicit Algorithms for Binate Covering Problems", *IEEE Trans. CAD*, vol. Vol. 16, pp. 677–691, July 1997.

[24] F. Somenzi, "Colorado Decision Diagram Package", Computer Programme, 1997.

[25] J. P. M. Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms", *in Portuguese Conf. on Artificial Intelligence*, 1999.

[26] DIMACS, "Benchmarks for boolean sat", ftp://dimacs.rutgers.edu/pub/challange/sat/benchmarks.

[27] A. J. Hu, G. York, and D. L. Dill, "New Techniques for Efficient Verification with Implicitly Conjoined BDDs", *in Proc. DAC*, pp. 276–282, 1994.

[28] R.K. Brayton, C. McMullen, G.D. Hachtel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[29] Jerry R. Burch Y. Hong, Peter A. Beerel and Kenneth L. McMillan, "Safe bdd minimization using don't cares", *in DAC, pp208-213*, 1997.

[30] S. Pilarski and G. Hu, "SAT with Partial Clauses and Backleaps", *in In Proc. DAC*, pp. 743–746, 2002.