Efficient Symbolic Simulation via Dynamic Scheduling, Don't Caring, and Case Splitting

Viresh Paruthi[1], Christian Jacobi[2], and Kai Weber[2]

[1] IBM Systems Group, Austin, TX
[2] IBM Deutschland Entwicklung GmbH, Boeblingen, Germany

Abstract. Most computer-aided design frameworks rely upon building BDD representations from netlist descriptions. In this paper, we present efficient algorithms for building BDDs from netlists. First, we introduce a dynamic scheduling algorithm for building BDDs for gates of the netlist, using an efficient hybrid of depth- and breadth-first traversal, and constant propagation. Second, we introduce a dynamic algorithm for optimally leveraging constraints and invariants as *don't-cares* during the building of BDDs for intermediate gates. Third, we present an automated and complete *case splitting* approach which is triggered by resource bounds. Unlike prior work in case splitting which focused upon variable cofactoring, our approach leverages the full power of our don't-caring solution and intelligently selects arbitrary functions to apply as constraints to maximally reduce peak BDD size while minimizing the number of cases to be explored. While these techniques may be applied to enhance the building of BDDs for arbitrary applications, we focus on their application within cycle-based symbolic simulation. Experiments confirm the effectiveness of these synergistic approaches in enabling optimal BDD building with minimal resources.

1 Introduction

Many applications in computer-aided design rely to some degree upon building BDD representations from netlist descriptions, such as combinational and sequential equivalence checking, bounded, unbounded, and inductive property checking, and design optimization and abstraction algorithms. Even modern satisfiability solvers, increasingly finding applications in domains for which BDD-based techniques were long considered the only alternative (such as unbounded verification), are likely to use a hybrid-algorithm scheme integrating BDDs for optimality [16, 15].

In this paper, we present an efficient set of synergistic algorithms for building BDDs from a netlist. First, we present a new scheduling algorithm for optimal BDD building. Our proposed resource-constrained interleaved depth-first and (modified) breadth-first schedule heuristically converges upon an optimal schedule for building BDDs. The scheme dynamically alternates between a depth-first and breadth-first schedule with progressively increasing resources until all BDDs for the netlist nodes have been built. Such a scheme combines the advantages of building BDDs with either of the two schedules, and augments it further by doing so in a resource-constrained manner resulting in a robust summation of their strengths. Furthermore the resource-constrained scheme handles constant propagation very efficiently which is particularly effective in property checking and equivalence checking frameworks.

Second, we present a novel method to take advantage of constraints and invariants to optimize intermediate BDDs. Constraints arise as user-specified restrictions of the environment, and also as a means to perform manual case splitting for computational efficiency. Essentially, constraints and invariants are applied as don't-cares when building BDDs in an attempt to optimize their size by heuristically factoring in the constraints early. This is closely intertwined with the scheduling algorithm described above such as to realize its benefits at each step of the BDD building process. Additionally, this is controlled by BDD size thresholds resulting in a tight and robust algorithm that dynamically trades-off resources invested with the desired reduction in BDD sizes.

Third, we describe an automatic and complete case splitting strategy that decomposes the problem into smaller problems, thus enabling building BDDs of the netlist without exceeding resources. In addition to case splitting on inputs, we present techniques for case splitting on internal signals by *constraining* them to constant values, and propagating these constraints to other BDDs. This is equivalent to restricting the inputs in the support of the chosen internal signal to values that cause it to assume the selected constant value. Note that this nicely interacts with the resource-constrained BDD building and its efficient constant propagation, and with the don't-care optimization of intermediate node BDDs. We additionally present new heuristics to choose signals to case split upon. Completeness is ensured by applying all possible values to the case split inputs and signals. The method gracefully degrades into underapproximate analysis once global resources are exceeded by not exploring all case split branches.

In this paper we present the described algorithms in the context of a cycle-based symbolic simulation (CBSS) [5] engine. A CBSS performs a cycle-by-cycle symbolic simulation of the design under test, and thus extends the cycle simulation methodology to symbolic values. The simulator essentially performs forward bounded symbolic simulation starting from the initial states. Symbolic values (represented as BDD variables) are assigned to the inputs in every cycle and propagated through the circuit to the outputs and state variables. This technique enables simulating large input spaces in parallel due to the inputs assuming symbolic values at each time-step. The bottleneck of the approach lies in the possible explosion of the BDD representations of the netlist nodes and state variables; this is alleviated by our proposed BDD-building scheme, don't-care optimization, and case splitting strategy.

We briefly describe synergies of this engine with other transformation and verification algorithms in a *Transformation-Based Verification* (TBV) [18] framework. By utilizing a sequence of transformation engines we may achieve significant reductions in the size of the design in a manner that benefits simulating it symbolically using the described algorithm. Additionally, the simulator may be leveraged as a falsification and proof algorithm in a number of settings.

Related Work Other researchers [20, 21, 23] have studied various scheduling techniques (e.g., DFS, BFS, hybrid) for BDD operations inside a BDD package, eg. in which order to traverse the BDD sub-graphs when ANDing two BDDs. The order of processing in BDD operations themselves is a different (and independent) question from the order in which BDDs for gates in a netlist are built. The latter question pertains to our work, and has also been studied by Aloul et al. [2]. They propose the use of partitioning and placement information to obtain a min-cut based scheduling for the gates, i.e. gates

which are close together in the circuit are scheduled close together as well. A drawback of their method is that they spend a considerable amount of time obtaining a schedule. Our method is more robust and dynamically adapts itself to different circuit topologies.

Rather then looking at a schedule for the whole netlist, Murgai et al. [19] delve into finding an optimal schedule for combining BDDs at the inputs of a multi-input AND gate when attempting to build the BDD for the gate output. They select which two BDDs to combine next based on a size- and support-based analysis of the candidate BDDs. Their approach is complementary to our approach and may easily be integrated into our overall netlist schedule. DFS and BFS are commonly used schedules for building BDDs for netlists. We extend these by proposing a hybrid DFS-BFS schedule for this task and further optimize by building BDDs in a resource-constrained manner and propagating constants efficiently.

Algorithms for optimizing BDDs with respect to don't-care sets has been studied in [10, 11]. We utilize and extend these algorithms by dynamically choosing the BDD-minimization algorithm based on size thresholds. We additionally propose the novel application of constraints as don't-cares during intermediate BDD building which often substantially reduces peak BDD size.

Wilson et al. [22] use ternary symbolic simulation (X-values) to abstract internal nodes to deal with the computational complexity. They also briefly mention case splitting on input variables, but do not detail their algorithms for selecting case split nodes, or the management of case splits. Our method extends their work by also being able to split upon internal nodes and using different heuristics to select nodes to case split upon. Completeness in our approach is ensured by symbolically simulating all possible values of the case split inputs and signals, and is handled automatically unlike the manual case splitting technique presented in [1]. In contrast to the approximating approach presented by Bertacco et al. [5, 6] our approach is complete in that it checks the design exhaustively.

A recent body of work addresses the generally exponential relation between the number of variables in the support of a BDD and its size by *reparameterizing* the representation onto a smaller set of variables, e.g. [4]. This technique has been extended to cycle-based symbolic simulation by reparameterizing unfolded input variables [5, 6, 8]. Such approaches are complementary to the techniques presented in this paper. Our techniques may be used to more efficiently compute the desired BDDs for the functions to be reparameterized. After reparameterization, our approach may again be used to continue the computations, seeded by the results of the reparameterization.

Organization The rest of the paper is organized as follows. The next section (Section 2) introduces some notation used throughout the paper to aid in describing our approach. Section 3 gives a high-level overview of the CBSS algorithm. In Section 4 we present an optimal scheduling technique for building BDDs for gates in a netlist representation of a design. Next we describe a method to optimally utilize constraints and invariants in Section 5. Section 6 describes efficient techniques to perform case splitting to deal with the complexity of symbolic analysis, and Section 7 delves into synergies of symbolic simulation with other algorithms in a Transformation-Based Verification framework. Lastly we present experimental results in Section 8, followed by concluding the paper.

2 Netlists: Syntax and Semantics

A *netlist* is a tuple $N = \langle\langle V, E\rangle, G, Z\rangle$ comprising a directed graph with nodes V and edges $E \subseteq V \times V$. Function $G : V \mapsto types$ represents a semantic mapping from nodes to gate $types$, including constants, primary inputs (i.e., nondeterministic bits), registers (denoted as the set $R \subset V$), and combinational gates with various functions. Function $Z : R \mapsto V$ is the initial value mapping $Z(v)$ of each register v, where $Z(v)$ may not contain registers in its transitive fanin cone. The *semantics of a netlist* N are defined in terms of traces: $\{0, 1\}$ valuations to netlist nodes over time which are consistent with G. We denote the value of gate v at time i in trace p by $p(v, i)$.

Our verification problem is represented entirely as a netlist, and consists of a set of *targets* $T \subseteq V$ correlating to a set of properties $AG(\neg t), \forall t \in T$. We thus assume that the netlist is a composition of the *design under test*, its *environment* (encoding *input assumptions*), and its *property automata*. The goal of the verification process is to find a way to drive a '1' to a target node, or to prove that no such assertion of the target is possible. If the former, a counterexample trace showing the sequence of assignments to the inputs in every cycle leading up to the fail is generated.

A set of *constraints* $C \subseteq V$ may be used to filter the stimulus that can be applied to the design. In the presence of constraints, a target $t \in T$ is defined to be hit in trace $p \in P$ at cycle i if $p(t, i) = 1$ and $p(c, i') = 1$ for all $c \in C, i' \leq i$. A target is *unreachable* if it cannot be hit along any path. Algorithmically, when searching for a way to drive a '1' to a target, the verification process must prune its search along paths which violate constraints.

A set of *invariants* $I \subseteq V$ specify properties inherent in the design itself. I.e. invariants will always evaluate to '1' in every time-step along every trace, at least until a constraint is violated. Invariants encode "truths" about a design that may be utilized as constraints to tighten overapproximate techniques (such as induction) to enhance proof capability. Invariants may be generated using a variety of mechanisms, e.g. the negation of targets previously proven unreachable.

We map all designs into a netlist representation containing only primary inputs, one "constant zero" node, 2-input AND gates, inverters, and registers, using straightforward logic synthesis techniques to eliminate more complex gate types [16]. Inverters are represented implicitly as edge attributes in the representation.

3 Background

A Cycle-based Symbolic Simulator (CBSS) [5] performs a cycle-by-cycle symbolic simulation of the design under test, typically using BDDs. It applies symbolic values at the inputs in every cycle, and propagates those to the state-variables and targets. Hence, state-variables and targets are always expressed in terms of symbolic input values, i.e., as Boolean functions of the symbolic inputs applied in the current and all prior cycles. If a target is hit, counterexample traces are generated by simply assigning concrete values to the symbolic input values in the cycles leading up to the fail.

Figure 1 gives an outline of the algorithm. The algorithm applies symbolic inputs in the form of new BDD variables at the inputs in every cycle, in function **create_variables**. At the outset, BDDs for the initial-states of the state-variables are computed and stored at the respective state-variables via function **update_state_variables**. Next, BDDs for

```
Algorithm cycle_sym(num_cycles) {
  for (cycle_num = 0; cycle_num ≤ num_cycles; cycle_num++) {
    create_variables(inputs); // Create new BDD variables for inputs in the current cycle
    if (cycle_num == 0) {
      build_node_bdds(initial_state_fns); // Build BDDs for the initial states
      update_state_variables(initial_state_fns); // Initialize the design
    }
    build_node_bdds(constraints); // Build BDDs for the constraints
    build_node_bdds(targets); // Build BDDs for the targets
    constrain_node_bdds(targets, constraints); // Constrain target BDDs
    check_targets(targets); // Check targets for being hit
    if (all_targets_solved(targets)) return;
    build_node_bdds(next_state_fns); // Build BDDs for the next-functions
    update_state_variables(next_state_fns); // Update state-vars
  }
}
```

Fig. 1. Generic cycle-based symbolic simulation algorithm

the constraints and targets are obtained by evaluating the combinational logic of the netlist starting with the new BDD variables at the inputs and the current BDDs at the state-variables. The computation of the constraints ANDs the constraint valuations (BDDs) from the previous cycles to the BDDs obtained for the constraint nodes in the current cycle. These "accumulated" constraint BDDs are then ANDed with the target BDDs (function **constrain_node_bdds**) before the targets are checked for being hit in function **check_targets** to ensure that the target valuations are consistent with the *care set* defined by the constraints. Thereafter, the combinational next-state logic of the state-variables is evaluated (again starting at the current BDD variables of the primary inputs and current state-variable BDDs), followed by updating the state-variables with the valuations obtained at the respective next-state functions. The process is iterated until all targets are *solved* (i.e. hit) or the design has been simulated symbolically for the specified maximum number of cycles.

4 Dynamic BDD Scheduling Algorithm

The bulk of the time during symbolic simulation is spent building BDDs for nodes in a netlist graph (function **build_node_bdds** in Fig. 1). A set of nodes whose BDDs are required to be built at each step, called "sinks," are identified. Sinks correspond to targets, constraints, invariants, initial-state and next-state functions of state variables. BDDs of some netlist nodes are available at the beginning of each cycle, namely those of the current content of state-variables, and new BDD variables created for the primary inputs (function **create_variables** in Fig. 1). The BDD building task is to compute BDDs for the sink nodes, starting at nodes for which BDDs exist, according to the semantics of the gates in the underlying combinational network.

It is known that different schedules for building BDDs for nodes of a netlist lead to significantly different peak numbers of BDD nodes [19, 2]. It is of utmost importance that this peak number be kept as low as possible. A large number of BDD nodes results in bad memory and cache performance, and severely degrades performance of expensive optimization algorithms such as Dynamic Variable Reordering (DVO) and

Garbage Collection. Optimal DVO has an impractically high computational complexity in the worst case (the problem is known to be NP-Hard [7]). Practical DVO approaches look for local minima based on time or memory limitations. They are likely to find better variable orderings when they are called on smaller number of active BDD nodes.

The BDDs for the sink nodes are built topologically starting at the inputs and state-variables, nodes for which BDDs exist at the beginning of a cycle. Two standard and commonly used schedules for building BDDs are depth-first (DFS) traversal of the netlist starting at the sink nodes, and breadth-first (BFS) traversal starting at the inputs and state-variables. Each of the two schedules have certain advantages and drawbacks depending on the structure of the netlist. Intuitively, when a netlist has many "independent components" which do not fan out to other parts of the netlist, DFS is often more efficient. This is because it builds BDDs for the components successively, hence only has the intermediate BDDs of a single component "alive" at any time. The algorithm is able to free BDDs for nodes in the component as soon as BDDs of their fanouts have been built. In contrast, the levelized nature of BFS builds BDDs of all components simultaneously causing many intermediate BDDs to be "alive" at the same time. But if a netlist node n has many fanouts, each processed by DFS along separate branches, the levelized BFS schedule is likely to perform better. The BDD for n can be freed as soon as all fanout gates of n are built, which often happens sooner with BFS particularly when the fanouts of n are level-wise close to n. This reduces the average "lifetime" of BDDs thus reducing the peak number of alive nodes. The experimental results in Section 8 demonstrate that each method outperforms the other method on some examples.

We extend the standard DFS- and BFS-based BDD building algorithms by applying them in a resource-constrained manner, using the algorithm of Figure 2. The algorithm builds BDDs for netlist nodes per the chosen schedule, but builds BDDs for gates in the netlist only up to a certain BDD size limit, i.e. it gives up building the BDD for a node if it exceeds an imposed size limit. After all node BDDs within this limit have been built, the limit is increased and the algorithm is applied again. We extend this further by alternating between DFS- and BFS-based schedules. Once all node BDDs within the current size limit have been built, the algorithm increases the size limit and switches to the other BDD building schedule. Such an interleaved hybrid DFS-BFS scheme brings together both a DFS and a BFS scheme in a tight and robust integration combining the advantages of both, and alleviating some of their drawbacks. The new scheme works in a "push-pull" manner by going back and forth between the two schedules. The DFS or the "pull" scheme uncovers any paths building BDDs along which may suffice to build the BDD for a sink node. The "push" or the levelized BFS traversal causes BDDs to be propagated quickly from the inputs toward the outputs with a tight control on the consumed resources.The resource limits further ensure that the overall algorithm does not get stuck in any one computation that does not contribute to the final result.

Building BDDs iteratively in a resource-constrained manner has several advantages over conventional approaches. First, since we restrict the BDD sizes at each iteration, DVO algorithms are able to converge on a good variable order when BDDs are small, causing larger BDDs that are computed later to be more compact and smaller. Second, the resource-constrained scheme ensures that nodes that have small BDDs can be computed and gotten out of the way early (and subsequently freed), to "make way" for

```
Algorithm build_node_bdds(sink_nodes) {
  // Compute DFS and BFS schedules for nodes in the cone-of-influence
  dfs_schedule = compute_dfs_schedule(sink_nodes);
  bfs_schedule = compute_bfs_schedule(sink_nodes);
  bdd_size_limit = INITIAL_BDD_SIZE_LIMIT;
  while (1) {
    // Attempt building BDDs within the current bdd-size-limit using a DFS schedule
    build_node_bdds_aux(dfs_schedule, bdd_size_limit);
    if (all_sink_node_bdds_built(sink_nodes))
      return SUCCESS;
    if (bdd_size_limit ≥ MAX_BDD_SIZE_LIMIT))
      return INCOMPLETE;
    bdd_size_limit = bdd_size_limit + DELTA_BDD_SIZE_LIMIT;
    // Attempt building BDDs within the current bdd-size-limit using a BFS schedule
    build_node_bdds_aux(bfs_schedule, bdd_size_limit);
    if (all_sink_node_bdds_built(sink_nodes)
      return SUCCESS;
    if (bdd_size_limit ≥ MAX_BDD_SIZE_LIMIT)
      return INCOMPLETE;
    bdd_size_limit = bdd_size_limit + DELTA_BDD_SIZE_LIMIT;
  }
}
```

Fig. 2. Interleaved DFS-BFS resource-constrained BDD building algorithm

larger BDDs later. Third, the resource-constrained algorithm can uncover and propagate constants very effectively. Note that if an input to an AND gate evaluates to a constant '0' there is no need to evaluate its other input function. Traditional BDD building approaches may spend a large amount of time and memory computing the BDD of that other fanin node function. Our resource-bounded scheme will effectively iterate between evaluating the function of both the fanin nodes under increasing size limits. If BDD size limits along either branch are exceeded, the scheme gives up building the BDD for this branch and moves on to the next node in the schedule, heuristically discovering the constant without the need to evaluate the more complex branch. This situation arises frequently in real designs, e.g. at multiplexers where some multiplexer data input functions may have significantly higher BDD complexity than the others and the selector signal is a constant, thus enabling the multiplexer to be evaluated by sampling the simpler data input. Once we discover a constant at a node we recursively propagate it along all the fanouts of this node.

We generalize this further to efficiently derive constants at partially evaluated multi-input AND (and OR) gates. It is frequently the case that the BDD for such a gate cannot be computed due to exceeding BDD size limits on some of the inputs, but the available BDDs together already imply a constant for the gate output, for example due to complimentary inputs along two branches. We recognize and exploit this situation by building a "partial" BDD for the multi-input AND structure by successively combining BDDs of the available fanin nodes within the current BDD size limit. If this evaluates to a constant at any point, we don't need to build the BDDs for the remaining fanin nodes, and instead we trigger constant propagation as described above.

5 Dynamic Don't Caring under Constraints and Invariants

Constraints are often used in verification to prune the possible input stimulus of the design. Semantically, the verification tool must discard any states for which a constraint evaluates to a '0'. In that sense, constraints impose "hard restrictions" on the evaluations performed by the verification tool, splitting the input space into two parts - the "valid" or the "care" set, and the "invalid" or the "don't-care" set. In the CBSS algorithm this is achieved by ANDing the accumulated constraints of the current and past cycles to the targets before they are checked for being hit. Recall that, during overapproximate search, our framework treats invariants as constraints.

We have found that constraints may be efficiently exploited as don't-cares to optimize intermediate BDDs during the course of the overall computation. This is achieved by modifying the intermediate BDDs in a manner such that they evaluate to the same Boolean values within the care set, but they are free to assume values in the don't-care set towards the goal of minimizing the size of the BDD [10, 11]. In some applications of constraints, like manual case splitting [13], or automatic case splitting (cf. Sect. 6), this minimization is key to the successful completion of the symbolic simulation without memory explosion.

We present a technique to exploit constraints and invariants optimally in a symbolic simulation setting. At each time-step of the symbolic simulation process BDDs for the constraints and invariants are computed and subsequently applied as don't-cares when building BDDs for the netlist nodes. This is done in a manner that ensures BDD sizes do not increase as a result of the don't-caring. The don't-caring is done by using one of the BDD *constrain* [10], *restrict* [10] and *compact* [11] operations. These algorithms ensure that the BDD valuations within the care set are unchanged, but for all values outside the care set they freely choose a '0' or a '1' value to minimize the BDD size.

Intuitively, don't-caring heuristically factors in the constraints early, and doing so helps to reduce the BDD representation of the intermediate nodes. The behaviors added by the intermediate application of the don't-cares will ultimately be eliminated before targets are checked for being hit (function **constrain_node_bdds** in Fig. 1). In a sense, our scheme rules out and/or adds behaviors precluded by the constraints early on by application of the constraints when building intermediate BDDs, as opposed to doing this only at the end once all the exact BDDs have been built.

Exact minimization is known to be NP-hard [11]; the *constrain*, *restrict*, and *compact* operators are therefore heuristic minimization algorithms. In the listed order, they are increasingly powerful in minimizing BDD sizes, at the cost of (often dramatically) increased runtime. Therefore, in our symbolic simulation algorithm we apply the cheapest, or the least computationally expensive, operation *constrain* first, and depending on size thresholds automatically switch to more expensive algorithms. This ensures a dynamic compromise between time and memory requirements. Also, this threshold-based scheme applies the cheap and fast minimization operation to the many small BDDs, and applies the more expensive operations to the only (hopefully) few large BDDs.

Note that some verification problems use constraints only for restricting the input stimulus, and have only minimal BDD reduction potential. Applying the expensive minimization algorithms to such designs will only marginally decrease the BDD size, but

```
Algorithm dont_care_node_bdd(node_bdd, constraint_bdds) {
  if (bdd_size(node_bdd) < BDD_SIZE_THRESHOLD_CONSTRAIN)
    return node_bdd; // return if too small
  foreach (constraint_bdd in constraint_bdds) {
    if (supports_intersect(node_bdd, constraint_bdd)) {
      res_bdd = bdd_constrain_threshold(node_bdd, constraint_bdd); // constrain
      if (bdd_size(res_bdd) ≥ BDD_SIZE_THRESHOLD_RESTRICT)
        res_bdd = bdd_restrict_threshold(node_bdd, constraint_bdd); // restrict
      if (bdd_size(res_bdd) ≥ BDD_SIZE_THRESHOLD_COMPACT)
        res_bdd = bdd_compact(node_bdd, constraint_bdd); // compact
      node_bdd = res_bdd;
    }
  }
  return node_bdd;
}
```

Fig. 3. Algorithm for optimizing node BDDs using don't-cares

may have a severe impact on runtime. For such problems it is best to set the thresholds of the expensive operations very high. The *constrain* operator is so fast that it usually is worthwhile even on such examples.

Figure 3 gives an outline of the algorithm. Whenever a BDD for a netlist node has been built, the BDD is optimized by applying all the constraint BDDs as don't-cares, in function **dont_care_node_bdd**.[1] If the BDD size is below the threshold for the application of the *constrain* operator, the function immediately returns. Otherwise, any don't-caring first checks for the intersection of the cone-of-influence of the BDDs of the constraints with that of the node (function **supports_intersect**), and applies only those constraints that have some overlap. Functions **bdd_constrain_threshold** and **bdd_restrict_threshold** apply the *constrain* and *restrict* operators respectively, but additionally ensure that the size of the resultant BDD is no greater than the argument BDD by returning the argument BDD if the application of these operators causes the BDD size to increase (which is possible [10]).

It may be noted that the BDD for a constraint in any time-step is a conjunction of the BDD obtained for it in the current and all previous time-steps. If at any point the BDD for the constraint becomes a zero BDD, it implies that the design does not have a legal state-space beyond this time-step and any unsolved targets are trivially unreachable.

6 Automated Case Splitting

In this section we describe automated case splitting strategies to ameliorate the BDD explosion which may occur during symbolic simulation. The described method ensures that the total number of BDD nodes does not exceed a specified limit, ultimately enabling symbolic simulation to complete computations which otherwise would be prone to memory-out conditions. In our proposed method we address the memory blow-up when computing intermediate BDDs as follows:

[1] Note that we cannot use don't-care minimization when building BDDs for the constraint nodes themselves; if we did, we could alter their care set.

- If the total number of BDD nodes exceed a certain threshold, we select a netlist node to case split on, and a constant value to be applied to the selected node.
- Upon case splitting the BDD sizes drop significantly and we continue with the symbolic analysis. Note that we may case split on any number of netlist nodes at different steps and stages of the symbolic simulation.
- Once the symbolic analysis completes, i.e. the design has been symbolically simulated for the required number of time-steps, we "backtrack" to the last case split (and the time-step in which it was applied) and set the selected netlist node to the other constant, and complete the symbolic analysis on this branch. This is continued until all case splits are covered, ensuring completeness.

All case splits are entered onto a stack that snapshots BDDs for the non-chosen value of the case split node (and discards the current BDDs at the node) to enable backtracking to this case split. The case splitting decomposes the problem into significantly smaller subproblems each of which is then individually discharged. Expensive BDD operations such as DVO benefit greatly from such a decomposition due to the subproblems being much smaller, and the fact that they can be solved independently of the others; in particular, DVO can apply different variable orderings along different branches of the case splits. A parallel may be drawn between case splitting and satisfiability (SAT) approaches with the case split nodes representing decision variables - the BDDs encode all possible solutions of the netlist nodes for the particular value of the case split node as opposed to SAT systematically exploring the solutions one-by-one.

We propose two techniques to select the netlist node or nodes to case split upon. We have found these to be very effective in managing space complexity of BDD operations. We describe these in the context of selecting a single node to case split upon, but they can be easily extended to case split on multiple nodes in one step:

- *Case split on the "fattest" variable(s)*. The fattest variable, at a given point in time, is defined as a variable that has the largest number of BDD nodes in all the live BDDs. Hence, setting this variable to a constant causes the largest reduction in the number of BDD nodes.
- *Case split on an internal node via constraining*. Here we select a netlist node other than inputs to case split upon based on algorithmic analysis. The analysis may include the reduction potential by examining the netlist graph and BDDs available for internal nodes. Next, the BDD for the selected case split node or its inverse is treated as a constraint, which is then added to the list of constraints as a *derived* constraint. The new constraint is subsequently used for minimizing all other BDDs by means of don't-caring as described in the previous section. The derived constraint is later removed from the list of constraints when the algorithm backtracks. For the other branch of the split, the inverse of the case split BDD is applied as a constraint. As an example, we may try don't-caring all live BDDs with the BDD for each node, and select the one that gives maximal reduction. Note that a constraint is effectively a restriction on the variables in its support and divides the input space according to the constraint BDD. Essentially, case splitting on an internal netlist node in a certain cycle of the symbolic simulation is equivalent to removing the logic in the cone-of-influence of this node up to the current time-step in an unfolded ver-

sion of the netlist - and then using the Boolean consequences of this reduction for minimizing other BDDs.

If the global resources are exhausted this case splitting gracefully degrades into underapproximate analysis by not exploring all branches. In underapproximate analysis, at every case split the algorithm heuristically chooses the branch to explore next, which enables semi-formal analysis. For example, the case split algorithm can be configured to always select the simpler branch first (i.e. the smaller one after case splitting) in order to reach very deep into the state space. Using underapproximate symbolic simulation thus balances the benefits of standard binary simulation (reaching very deep) with the power of symbolic simulation, effectively simulating a large number of input combinations in parallel, hence visiting a large number of states.

7 Transformation Synergies

Here we briefly sketch scenarios and interactions of this engine with other algorithms that we have found to be useful. We have deployed the symbolic simulator as an engine in the IBM internal TBV [18] system *SixthSense*. Such a system is capable of maximally exploiting the synergy of the transformation and verification algorithms encapsulated in the system as engines against the verification problem under consideration.

Approaches that build BDDs from netlist representations tend to benefit dramatically from prior simplifying transformations applied to the netlist. For example, redundancy removal and logic rewriting algorithms [16] that reduce the number of gates in the netlist reduce the number of distinct BDDs that need to be built, and may even reduce the cutwidth of the netlist implying a need for fewer live BDD nodes. In a CBSS approach, reductions to the sequential netlist are particularly useful, as they reduce the complexity of every subsequent time-frame of the symbolic evaluation. In particular, we have found the input reductions enabled by structurally abstracting the netlist through reparameterization [4] to be very beneficial to symbolic simulation, often times improving performance by orders of magnitude. Note that this is complementary to traditional approaches that reparameterize state sets during symbolic simulation [1, 5, 6, 8]. In fact, both these can be combined into a powerful two-step process that reparameterizes the structural sequential netlist, followed by reparameterizing the next-state BDDs at every time-step of the symbolic simulation.

In a semi-formal setting when performing a directed search of the state-space of the design, symbolic simulation performs a broad simulation of the design within specified resources. The engine thus uncovers large portions of the state space, and allows for probabilistically uncommon scenarios to be exposed that cause the fail events to be hit. When performing an exhaustive k-step bounded model check of the design, the symbolic simulator often outperforms SAT-based bounded model checking [16], particularly when the number of inputs is not too large or for small values of k. Additionally, we have found this engine to be very useful when attempting proofs via k-step BDD-based induction. Furthermore, the engine may be used to obtain proofs in conjunction with an engine that computes a *diameter* estimate of the design [3].

Localization [17] augmented with counterexample-guided abstraction refinement [9] has been shown to be an effective technique for obtaining proofs. Such paradigms rely upon exhaustive bounded search to provide counterexamples from which to refine the

S.No.	Design	Design Size					#Cycles
		#Inputs	#Registers	#Gates	#Targets	#Constraints	
1	FPU_ADD	440	5025	79105	84	5	26
2	FPU_FMA	452	5785	72272	82	4	18
3	IBM_03	25	119	2460	1	2	33
4	IBM_06	37	140	3157	1	2	32
5	SLB	57	692	3754	1	0	8
6	CHI	112	92	732	1	0	9
7	SCU	71	187	810	1	0	23

Table 1. Details of examples used in the experiments

S.No.	DFS		Res. DFS		BFS		Res. BFS		DFS-BFS	
	T(s)	N(10^6)	T(s)	N(10^6)	T(s)	N(10^6)	T(s)	N(10^6)	T(s)	N(10^6)
1	inf	inf	40.06	0.17	inf	inf	45.46	0.24	40.22	0.17
2	inf	inf	14580	101.14	inf	inf	inf	inf	10399	96.78
3	57.2	3.12	54.7	3.14	59.1	3.75	58.1	3.68	52.1	3.14
4	7392	96.56	7916	111.39	8991	129.79	8150	118.23	7897	106.60
5	1901	329.44	2094	291.84	1982	304.73	1976	291.84	1832	291.83
6	1000	91.26	907	83.85	1019	89.27	1021	88.57	910	85.94
7	112	6.58	91	6.18	120	74.31	113	74.08	84	6.28

Table 2. BDD node count and runtimes for the different schedules without DVO

abstracted design. A symbolic simulation engine is apt for performing such bounded analysis of the localized design. Additionally, the exhaustive representation using BDDs may be inherently exploited to derive minimally sized refinements.

8 Experimental Results and Conclusions

In order to gauge the effectiveness of various aspects of our symbolic simulation algorithm we chose a diverse set of industrial designs to conduct our experiments on (see Table 1). All experiments were run on an IBM pSeries computer with POWER4 processors running at 1.4GHz using the IBM internal verification tool *SixthSense*. All designs were put through reductions using a BDD-based combinational redundancy removal engine [16] before the symbolic simulator was applied. FPU_ADD and FPU_FMA are the verification problems of the dataflow for a floating-point "add" and "fused-multiply-add" instruction respectively [13]. IBM_03 and IBM_06 are examples from the IBM Formal Verification Benchmarks [12]. These were randomly chosen from among those with constraints. SLB is a Segment Lookaside Buffer, CHI is a Channel Interface and SCU is a Storage Control Unit. SLB, CHI, and SCU are optimized control intensive circuits that have been put through a number of design transformations.

We ran experiments to measure the resources (time/memory) required to symbolically simulate the above designs with all three scheduling schemes, namely BFS, DFS and DFS-BFS, in different settings. In order to show the benefits of resource constraining, we ran the DFS and BFS schemes with and without resource constraints. The results are given in Table 2. A value of "inf" indicates that the particular run exploded (> 500

S.No.	DFS		Res. DFS		BFS		Res. BFS		DFS-BFS	
	T(s)	N(10^6)	T(s)	N(10^6)	T(s)	N(10^6)	T(s)	N(10^6)	T(s)	N(10^6)
3	236	2.54	239	2.65	168	2.98	172	2.97	215	2.64
4	24507	94.13	25135	100.19	26979	100.91	26103	101.47	24369	100.01
5	648	7.7	577	6.80	1122	14.78	1127	7.68	631	6.81
6	2643	61.39	1346	53.43	3042	16.86	2270	25.66	2000	30.69
7	382	59.66	398	55.57	582	69.35	524	67.48	311	56.40

Table 3. BDD node count and runtimes for the different schedules with DVO

S.No.	Without don't-caring		With don't-caring	
	Time(s)	Nodes(10^6)	Time(s)	Nodes(10^6)
1	inf	inf	40.22	0.17
2	inf	inf	10399	96.78
3	48.44	3.50	52.11	3.14
4	7990	109.40	7897	106.60

Table 4. BDD node count and runtimes with and without don't-caring using constraints

million) in the number of BDD nodes. The "#Nodes" in the tables is the peak number of BDD nodes reported by the BDD package [14]. In the first set of experiments we turned Dynamic Variable Reordering (DVO) off to get a true comparison since DVO can skew results due to its heuristic nature. The table also compares and contrasts the three scheduling techniques. The benefits of resource constraining are amply clear from the results. It is indispensable for the FPU designs where we see the runs explode without any resource constraining, and go through easily with resource constraining. This is likely due to the propagation of a large number of constants which resource constraining specializes in taking advantage of, in particular when many such constants are created due to constraints [13]. The effects are somewhat less pronounced in some other examples due to the fact that they have symbolic initial values or are highly optimized, causing less constants to propagate. Note that resource constraining is inherent in the hybrid DFS-BFS interleaved scheme as it enables switching between the two underlying schemes. It is clear that each of DFS and BFS outperforms the other on different examples. The hybrid DFS-BFS scheme clearly stands out as the most robust, and nicely combines the individual benefits of DFS and BFS schedules. By and large it has a peak number of BDD nodes that is close to or less than the lower of the peaks of the two underlying schemes, and runtime that is close to or better than the faster one.

We repeated the above experiment this time with DVO enabled (Table 3). We observed a somewhat similar pattern, though things varied a bit more. We attribute this to the heuristic nature of DVO, and the fact that we used low effort DVO. The heuristic nature of DVO is clearly demonstrated by the FPU examples that explode now in both non-resource-constrained as well as in the resource-constrained case. The hybrid DFS-BFS scheme again comes across as the best overall and shows consistent performance in different scenarios re-enforcing our claim. It provides the benefits of both resource constraining as well as a summation of the strengths of DFS and BFS schedules resulting in a powerful and robust approach that works for all cases.

Design	Target Status	No case splitting		Case split on fattest variables			Evaluation
		T(s)	N(10^6)	T(s)	N(10^6)	#Cases	
2	Reachable	57.11	3.48	40.07	1.06	4(0)	Underapprox
4	Reachable	7897	106.6	16097	11.70	5(4)	Underapprox
5	Unreachable	631	6.81	570	6.24	3	Complete
6	Unreachable	910	85.94	806	36.44	4	Complete

Table 5. BDD node count and runtimes with and without case splitting using constraints

Next we measured the impact of handling constraints as don't-cares during intermediate BDD building. The hybrid DFS-BFS scheme was used for the purposes of this experiment. The results are summarized in Table 4. Only designs containing constraints were used for this experiment. The intermediate don't-caring using constraints (cf. Section 5) is absolutely essential to get the FPU examples through - the runs simply explode otherwise. The impact of factoring in constraints early can have a significant impact on reducing intermediate BDD sizes, or it may not depending on the nature of the constraints and the design. If a constraint prunes a fair amount of the input stimulus it may be very effective in reducing BDD sizes, but on the other hand if the BDDs of the internal nodes are already optimized then it may not do much. Hence, in our scheme we use threshold based don't-caring that is cheap for the most part, and apply more aggressive but computationally complex don't-caring only for very large BDDs. Such an approach was essential to automatic verification of FPUs as described in [13].

Lastly, we ran those designs with a large number of nodes with case splitting enabled (Table 5). The results are a mix of underapproximate evaluation (with some backtracks) for designs in which the targets were hittable, and others for which complete case splitting was done to prove that the targets are not hittable boundedly. The benefits of case splitting in both cases is clear. It helps to hit the reachable targets much sooner while visiting a large number of states of the design and within resources bounds, and enables completing exhaustive bounded checks on designs with unreachable targets without exploding in memory. Essentially, it trades-off complexity in memory with time, but is a compromise that is worth it to complete analysis on a design which otherwise may not - though for example #6 the overall performance is much better possibly due to a reduced number of BDD nodes to deal with. The numbers in parenthesis in the "#Cases" column indicates the number of case splits for which the other branch was evaluated as well. Hence, in the case of example #4, 4 of the 5 case splits were fully evaluated. Case splitting on internal nodes was necessary to verify FPU designs using formal methods in a fully-automated manner, as detailed in [13].

Conclusion We presented a robust set of algorithms for building BDDs efficiently for netlists. We presented a scheduling scheme that dynamically converges upon a heuristically optimal schedule for computing BDDs using an efficient hybrid of depth- and breadth-first search called out in an interleaved manner under resource constraints. We introduced a dynamic algorithm, tightly integrated with the scheduling scheme, to optimally leverage constraints and invariants as don't-cares when building BDDs for intermediate gates in the netlist. Additionally, we described an automatic and complete case splitting approach that is triggered and controlled by resource bounds to decompose the overall problem into simpler parts which are then solved individually. The presented

approach takes advantage of the full power of our don't-caring solution and smartly selects arbitrary functions to apply as constraints to maximally reduce peak BDD size while minimizing the number of cases to be explored.

References

1. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC*, 1999.
2. F. A. Aloul, I. L. Markov, and K. A. Sakallah. Improving the efficiency of Circuit-to-BDD conversion by gate and input ordering. In *ICCD*, 2002.
3. J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In *Computer-Aided Verification*, July 2002.
4. J. Baumgartner and H. Mony. Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies. In *CHARME*, 2005.
5. V. Bertacco, M. Damiano, and S. Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *Design Automation Conference*, June 1999.
6. V. Bertacco and K. Olukotun. Efficient state representation for symbolic simulation. In *Design Automation Conference*, June 2002.
7. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3), Sept. 1992.
8. P. Chauhan, E. Clarke, and D. Kroening. A SAT-based algorithm for reparameterization in symbolic simulation. In *Design Automation Conference*, June 2004.
9. E. M. Clarke, O. Grumburg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, July 2000.
10. O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, 1989.
11. Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan. Safe BDD minimization using don't cares. In *Design Automation Conference*, June 1997.
12. IBM Formal Verification Benchmark Library. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/fvbenchmarks.html.
13. C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner. Automatic formal verification of fused-multiply-add floating point units. In *DATE*, 2005.
14. G. Janssen. Design of a pointerless bdd package. In *IWLS*, 2001.
15. H.-S. Jin, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In *CAV*, 2004.
16. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. CAD*, 21(12), 2002.
17. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
18. H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. In *FMCAD*, 2004.
19. R. Murgai, J. Jain, and M. Fujita. Efficient scheduling techniques for ROBDD construction. In *VLSI Design*, pages 394–401, 1999.
20. H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *International Conference on Computer-Aided Design*, 1993.
21. J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hiercharchy. In *DAC*, pages 635–640, 1996.
22. C. Wilson, D. L. Dill, and R. E. Bryant. Symbolic simulation with approximate values. In *Formal Methods in Computer-Aided Design*, pages 335–353, Nov. 2000.
23. B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron. Space- and time-efficient BDD construction via working set control. In *ASP-DAC*, pages 423–432, 1998.