Enhanced Reachability Analysis via Automated Dynamic Netlist-Based Hint Generation

Jiazhao Xu Mark Williams Hari Mony Jason Baumgartner
IBM Systems & Technology Group

Abstract— **While SAT-based algorithms have largely displaced BDD-based verification techniques due to their typically higher scalability, there are classes of problems for which BDD-based reachability analysis is the only existing method for an automated solution. Nonetheless, reachability engines require a high degree of tuning to perform well on challenging benchmarks. In addition to clever partitioning and scheduling techniques, the use of *hints* has been proposed to decompose an otherwise breadth-first fixedpoint computation into a series of underapproximate computations, requiring a larger number of (pre-)image iterations though often significantly reducing peak BDD size and thus resource requirements. In this paper, we introduce a novel approach to boost the scalability of reachability computation: automated netlist-based hint generation. Experiments confirm that this approach can yield significant resource reductions; often over an order of magnitude on complex problems compared to reachability analysis without hints, and even compared to SAT-based proof techniques.**

I. Introduction

Since the advent of symbolic model checking more than two decades ago, automated verification tools have evolved dramatically in capacity. This evolution is due to a variety of innovations, including (in extreme brevity) advanced BDD-based techniques [1], [2], SAT-based proof [3], [4] and falsification engines [5], [6], [7], a variety of simplification and abstraction techniques to reduce problem complexity [8], [9], [10], and a modular transformation-based tool architecture to allow all of the above to synergistically decompose a complex verification problem [11] under guidance of advanced orchestration techniques [12]. Clever software engineering techniques, parallel processing, and more powerful computers upon which to run these tools have also played an important role. This boost in *scalability* has yielded a boost in *usability*, proliferating model checking from a craft requiring dedicated verification expertise to pervasive use even by non-experts, e.g., for lighter-weight assertion-based verification or sequential equivalence checking. Even state-of-the-art academic solvers such as ABC [13] and PdTrav [14] have become quite powerful through the above techniques.

The advent of unbounded SAT-based proof techniques such as interpolation [3] and IC3 [4] has played a particularly pronounced role in the scalability of contemporary model checkers. Whereas BDD-based reachability analysis tends to become impractical if the design under verification cannot be reduced or abstracted below several hundred state variables, SAT-based techniques on occasion can scale beyond tens of thousands of state variables. Nonetheless, BDDs may dramatically outperform SAT-based techniques for classes of problems, thus a well-tuned BDD-based reachability engine is an essential component of a state-of-the-art verification tool.

Numerous techniques have been developed to boost the scalability of BDD-based reachability engines. Examples include using a partitioned transition relation (TR) instead of a monolithic representation [1], advanced quantification and conjunction scheduling based upon metrics such as variable dependency [2], as well as heuristics to balance splitting and conjoining strategies [7]. The application of BDD-reduction operators such as *bdd_constrain*, *bdd_restrict* and *bdd_compact* [15] on the transition relation and state set representations have also yielded substantial scalability improvements.

The concept of *hints* was presented in [16] as a method to mitigate the BDD size explosion that often happens during intermediate steps of breadth-first reachability analysis, despite the BDDs being much more compact at early and even late stages. The intuition behind this phenomenon is that breadth-first analysis explores many disjoint design behaviors in parallel, causing asymmetries and thus bloat in the intermediate BDD representations – whereas the final reached state representation may have many asymmetries "filled in" hence be more compact. Hints are used to iteratively constrain the transition relation and thereby *direct* the symbolic search, computing states reachable along the constrained transition relation from those reached using prior hints. Completeness is ensured by finally restoring the original transition relation once the hints have been exhausted. Despite requiring a greater number of (pre-)image computations, this compaction of intermediate BDDs results in fewer and less-expensive dynamic variable ordering computations. These benefits collectively often reduce resources for complex problems, in cases enabling a solution for problems which would otherwise exhaust time or memory limitations. As noted in [16], as concurs with our practical experience: even *arbitrary* hints often reduce complexity for difficult problems. This preliminary work proposed the use of manually-generated hints based upon design insight.

This work was extended toward automation in [17], [18]. In [17] the authors propose to analyze the control-data flow graph of a behavioral Verilog design, using *branch conditions* as hints that effectively decompose the design similar to program slicing techniques. [18] extends this approach by using these conditions as a *known-complete* disjunctive partitioning, without requiring the final fixedpoint computation where the unconstrained transition relation is used to ensure completeness. While demonstrated as effective on a set of designs, these approaches are of limited practical applicability since they

978-0-9835678-2-0/12/$31.00 ©2012 FMCAD Inc.

require a high-level behavioral design format which may not be available. This becomes prohibitive in application domains such as sequential equivalence checking which may require analysis of post-synthesis netlists, and within a transformation-based verification toolset which may have applied numerous reduction and abstraction techniques to aggressively shrink the original netlist to something feasible for BDD-based analysis. These approaches also may not be suitable for classes of designs which are harder to program slice such as those with highly-pipelined or multi-threaded behavior.

Other techniques have also been proposed to reduce peak BDD size through departing from breadth-first search, such as high-density reachability analysis [19]. This technique resorts to intermediate under-approximate reachability analysis, partitioning images when BDD sizes exceed a threshold. Our practical experience with such approaches is that they suffer convergence problems (e.g., requiring a virtually-unbounded number of image computations) rendering them of limited practical utility. In contrast, a benefit of hints is that their impact on the number of image computations may provably be linearly bounded given proper controls.

In this paper, we introduce a novel automated dynamic hint generation approach to boost the scalability of reachability computation. In contrast to [17], [18], our work is focused upon generating high-quality hints from *arbitrary* netlist representations, and is triggered on-demand only when reachability computation exceeds a resource threshold. We have used this technique successfully both for property checking and sequential equivalence checking. Our specific contributions, as detailed in Section III, include a method to dynamically introduce hints to the reachability process based upon resource thresholds; dynamic algorithms to compute effective hint sequences from a transition relation; and a method to truncate reachability analysis under a given hint if it is deemed to risk increasing the number of overall image computations by too large a factor. While these techniques are all heuristic in their attempt to reduce the complexity of a reachability computation, our experiments in Section IV confirm that they often significantly boost performance for complex problems, and in many cases outperform SAT-based techniques.

II. Preliminaries

A model checking problem may be expressed as a *netlist*: a directed graph whose nodes (termed *gates*) comprise primary *inputs*, *state elements*, and a variety of combinational logic operators. State elements have associated *initial values* and *next-state functions*. A *state* is a Boolean valuation to the state elements. An *initial state* is a state consistent with the conjunction of the initial values.

The *transition relation* $TR(x, i, y)$ associated with a netlist comprises *current state variables* $\{x_1, \ldots, x_m\}$; *next state variables* $\{y_1, \ldots, y_m\}$; and *input variables* $\{i_1, \ldots, i_n\}$. It is defined in a straight-forward way from the next-state functions of the state elements of the netlist.

An *image computation* is used to compute the successors of a set of states s, defined by $\exists i.\exists x.TR(x, i, y) \wedge s$. A *reachability computation* may be performed by first setting the partial set of reached states to the initial states, then growing that set by iteratively computing its image to add to the partial set via union.

Algorithm 1 Reachability using Hints

```
 1: function FORWARDREACH(TR, hints, init_states)
 2:     reached = init_states
 3:     // true will be the last-used hint in hints
 4:     while (hint = pop(hints)) do
 5:         hint_TR = apply_hint(TR, hint) // constrain TR with hint
 6:         frontier = reached // first image with hint_TR uses reached
 7:         while (true) do
 8:             image = compute_image(hint_TR, frontier)
 9:             frontier = compute_frontier(image, reached)
10:             if (frontier is empty) then break
11:             end if
12:             reached = bdd_or(reached, frontier)
13:         end while
14:     end while
15: end function
```

III. Enhanced Reachability Algorithms

In this section we present our automated hint generation algorithms. Algorithm 1 depicts a traditional framework for reachability analysis using hints [16]. In a traditional application, the hints are manually provided to the reachability process, and the final hint must be *true* (or *constant 1*) to ensure that the original transition relation will be *restored* for a complete reachability computation.

There are several limitations of the use of hints in practice which we address in this paper.

1. Requiring manual specification of hints diminishes their utility, and enabling automation only for problems of suitable Verilog syntax [17] is limiting in practice. We thus introduce in Section III-A an effective automated hint generation algorithm which operates directly upon the transition relation, and in Section III-B an algorithm which iterates through the generated hints.

2. In cases, hints may degrade performance of the reachability computation because they increase the number of image computations, while not significantly reducing effort vs. unconstrained image computation. For easier problems, this is a risk because the image computations are already efficient. For complex problems, a fixed set of hints may not adequately simplify image computation, whereas a more aggressive set of hints may be helpful. To address this issue, we introduce in Section III-C a reachability framework which introduces hints upon demand, when BDDs exceed configurable thresholds.

3. In rare cases, hints may result in convergence problems for reachability computation. A pathological example is for a counter with a *parallel load* port, where any arbitrary state may be loaded into the counter under control of a particular input – otherwise it may take an exponential number of steps to transition from one reachable state of the counter to another. If a hint disables that parallel load, it may dramatically increase the number of necessary image computations for a fixedpoint

Algorithm 2 Hint Introduction Algorithm

```
    function GENERATE_HINTS(TR, hints, reached, reduction_limit, var_limit)
 2:     vars = all variables that are not in hints

        for all BDD variable var in vars do
 4:         compute rank of positive and negative literals of var
            add best rank literal to array ranks
 6:     end for

        sort ranks // ranks used to generate the initial hint cube
 8:     hint_cube = bdd_1 ∧ ⋀ hints // form cube for already-selected hints

        while (|ranks|) do
10:         literal = best candidate from rank
            prune rank from ranks
12:         new_cube = hint_cube ∧ literal

            if (new_cube contradicts TR or reached) then
14:             compute rank for opposite literal polarity; add to ranks
                re-sort ranks; continue

16:         else
                hint_cube = new_cube
18:             hints = hints ∪ literal
                compute TR size reduction of TR from new_cube

20:             if ((TR reduction exceeds reduction_limit) or (|hints| exceeds
    var_limit)) then
                    break
22:             end if
            end if
24:     end while

        return hints
26: end function
```

computation, slowing overall progress. We thus introduce in Section III-D a mechanism to truncate the use of a specific hint prior to fixedpoint if necessary, for overall robustness.

A. Automated Hint Generation Algorithm

Algorithm 2 outlines our automated hint-generation technique. The hints that we have found most effective are *BDD cubes* over input variables and/or current state variables. A BDD cube is a conjunction of BDD literals (positive or negative) over a set of BDD variables.

There are several heuristics that we have found effective for selecting the best BDD literals to include in hints. One heuristic is to first select a BDD variable using the *use_count* of that variable as its ranking measure; i.e., the number of BDD nodes associated with a given variable, then to select the positive or negative literal of that variable based on the amount of reduction to the transition relation each literal provided. The intuition of using this metric is that it provides an indication that asymmetries over the corresponding variable may be the cause of intermediate BDD growth. Another heuristic is to rank all the BDD literals according to the criteria of how much reduction a given variable cofactoring provides to the transition relation, which in turn provides an estimate of how much they may speed up image computation. We have empirically found that the former works best. The ranking metrics, along with the most promising variable polarity, are recorded in the *ranks* data structure against which each BDD variable will be sorted.

After ranking the BDD variables the next task is to select a set of BDD literals to form the first hint cube. This is not merely a matter of choosing the k highest-ranked literals from the sorted *ranks* data structure, as the result may yield a "contradicting" hint cube which has an empty intersection with the transition relation or *reached* set, which begins as the initial states. We thus perform a consistencycheck on the candidate *hint_cube* before adding a literal to it, and in case of a contradiction, we flip the polarity of that variable and re-rank. To avoid adding more literals to the first hint cube than necessary, we use two termination criteria: **(1)** *reduction_limit* measures the degree to which the given *hint_cube* reduces the *TR*, i.e. $\big(sizeof(TR) - sizeof(TR \wedge hint\_cube)\big)/sizeof(TR)$, and **(2)** *var_limit* which provides an upper-bound on the number of literals to be added to *hints*. Our experience shows that *reduction_limit* may be left large, on the order of 100% since a small transition relation will be fast for reachability computation anyway, and a *var_limit* of between 10 and 15 literals yields the best results for larger netlists (see further discussion in Section IV and Figure 3). We use the *bdd_and* operation to constrain the transition relation with the hint cube. Since the hint is merely a cube, the *bdd_and* operation is as effective as other BDD constraining operations.

It is noteworthy that the generated hints are highly dependent upon BDD variable ordering. This algorithm may be called multiple times in the overall reachability framework as per Algorithm 4, possibly adding additional hints to a non-empty set of previously-generated hints. It is likely that dynamic variable ordering was invoked between these calls, hence the added hints will reflect the best choice under the current ordering.

B. Hint Iteration Algorithm

In addition to deciding the set of literals that will be used for the hints, it is important to decide the *sequence* of hints that will be applied given this set. We have found the most consistently-effective hint-successor strategy to be iteratively eliminating literals from the original hint cube, thus starting the computation with a maximally-restrictive constraint and gradually relaxing that constraint. This observation was formed over years of relying upon the use of manual hints for BDD-based reachability analysis in practice, prior to the availability of more scalable alternative proof techniques. It is consistent with the intuitive notion that hints should be introduced to decompose an overly-complex fixedpoint computation from following many disparate design behaviors to focusing on a smaller yet growing set of behaviors. This process is depicted in Algorithm 3.

The *gen_next_hint* function is called as part of the overall reachability framework in Algorithm 4. When reachability analysis exceeds a complexity threshold and hint literals are generated, this algorithm determines the sequence in which literals are removed from that set for successive hints. Note also that the sequence of applied hints is *dynamically* determined, vs. merely deciding a fixed order when hint literals are generated via Algorithm 2, as variable ordering may have changed between those points.

Algorithm 3 Hint Successor Algorithm

```
    function GEN_NEXT_HINT(TR, hints, first, reached)
        cur_hint = bdd_1 ∧ ⋀ hints

3:      if (first) then
            return cur_hint
        end if

6:      re-rank and re-sort hints

        while (|hints|) do
            remove lowest-rank literal from hints
9:          next_hint = bdd_1 ∧ ⋀ hints

            if ((next_hint ∧ reached) ⊆ (cur_hint ∧ reached)) then
                continue // next_hint is vacuous

12:         else
                return next_hint
            end if
15:     end while

        return bdd_1
    end function
```

In our experiments, we observed occasional occurrences of "*vacuous*" hints which do not add any new states to the reached set. Rather than waste resources performing a useless image and frontier computation in such cases, we developed an inexpensive test to detect most vacuous hints and avoid generating them. This test consists of computing the conjunction I_c of *cur_hint* with the reached set, and checking if I_c contains the conjunction I_n of the candidate *next_hint* with the reached set. If so, *next_hint* is vacuous and we proceed to the next literal. Since our hints are cubes, this computation is efficient in practice. Empirically, we found that approximately 20% of candidate hints are vacuous, and this step results in approximately 15% improvement in overall performance.

C. Dynamic Hint Introduction

In practice, a *monolithic* application of hints may not be ideal for several reasons. First, for easier problems, the use of hints often degrades performance of the reachability computation because they increase the number of image computations, while not significantly reducing effort compared to the unconstrained image computations. In other cases, the application of hints is inadequate to reduce the complexity and make image computation tractable. We thus have developed a framework which introduces hints only upon demand, as BDD sizes exceed configurable thresholds.

We exploit a "node limit" feature provided by our BDD package which limits the peak number of nodes it is allowed to generate within a BDD operation. If an operation exceeds this limit, a special *UNKNOWN* handle is returned, which is treated similarly to the *X* value in ternary analysis. Every image computation is performed using a node limit, which allows that computation to add at most a fixed number of BDD nodes. If the image computation returns *UNKNOWN*, additional hint literals are generated to mitigate the BDD explosion, and the constrained image computation is repeated. Our practical experience is that the threshold should not be too small, nor overly large; an allowance of 350000 nodes is the best setting we have practically found. To allow convergence on very complex problems, whenever we generate hints, we increase this threshold by a configurable factor (50% is effective) to avoid future hints from being triggered too frequently on problems that intrinsically need large BDDs. This process is depicted in our overall reachability flow in Algorithm 4, under control of variable *bdd_threshold*.

D. Hint Truncation

In a traditional hint application as per Algorithm 1, a full fixedpoint of states reachable under the corresponding hint-constrained transition relation is performed for each hint. However, in cases, a hint may dramatically increase the number of necessary image computations as per the example of a counter with *parallel load* capability discussed in Section III. For robustness, we thus have found it useful to place a limit on the maximum number of image computations that are allowed for a given hint. Because it is difficult to predict the number of image computations which would be necessary without hints (i.e., the *diameter* of the design), this metric in practice can be kept quite large (on the order of 10000), and optionally increased every time this limit is encountered. Using such a facility, one may thus ensure that the use of hints increases the number of image computations vs. reachability without hints by at most a linear factor. This process is depicted in our overall reachability flow in Algorithm 4, under control of variable *hint_iters*.

E. Overall Enhanced Reachability Algorithm

Algorithm 4 summarizes our overall enhanced reachability framework, combining aspects described in prior sections. Compared to Algorithm 1, the primary differences are: **(1)** automated generation of hints (Section III-A); **(2)** dynamically-prioritized iteration among the generated hints, taking into account current variable ordering (Section III-B); **(3)** dynamic triggering of hint introduction (Section III-C); and **(4)** truncation of a hint if too many image computations are required under that hint (Section III-D).

IV. Experimental Results

In this section we provide experimental results to illustrate the effectiveness of our techniques. These experiments are all derived from the Hardware Model Checking Competition 2011 benchmarks [20], pruned to the 92 that: **(1)** were not trivially solved by light-weight logic optimizations or random simulation; **(2)** could complete a reachability computation either with or without hints within a 4 hour time limit and 4GB memory: and **(3)** our dynamic hint-generation algorithm was invoked due to resource requirements. Because these benchmarks are provided in AIGER form, not in behavioral Verilog syntax, the technique of [17] is not applicable. Furthermore, the benchmarks used in [17] are a small set that are not publicly available, hence we could not readily contrast our approaches. restrict our focus to those of [20].

We implemented our techniques in the reachability engine included in the IBM verification tool *SixthSense* [12]. This

Algorithm 4 **Reachability using Dynamic Automated Hints**

```
 1: function FORWARDREACH(TR, init_states, var_limit, bdd_threshold,
    bdd_growth_factor, depth_threshold, reduction_limit, hint_iters)
 2:     reached = init_states
 3:     hints = emptyset
 4:     first = true

 5:     while (hint = GET_NEXT_HINT(TR, hints, first)) do
 6:         first = false
 7:         hint_iters = 0
 8:         hint_TR = apply_hint(TR, hint) // constrain TR with hint
 9:         frontier = reached

10:         while (true) do
11:             image = compute_image(hint_TR, frontier, bdd_threshold)

12:             if (image ≡ UNKNOWN) then // aborted due to bdd_threshold
13:                 bdd_threshold = bdd_threshold * bdd_growth_factor
14:                 hints = GENERATE_HINTS(TR, hints, reached, reduc-
    tion_limit, var_limit)
15:                 first = true
16:                 break
17:             end if

18:             hint_iters++

19:             if (hint_iters ≥ depth_threshold) then break // goto next hint
20:             end if

21:             frontier = compute_frontier(image, reached)

22:             if (frontier is empty) then
23:                 if (hints ≡ emptyset) then return // fixedpoint complete
24:                 end if
25:                 break // goto next hint
26:             end if

27:             reached = bdd_or(reached, frontier)
28:         end while
29:     end while
30: end function
```

engine uses an internally-developed BDD package [21], with standard features such as dynamic variable ordering, as well as more advanced techniques such as support for multiple distinct BDD "managers" with the ability to cast BDDs from one to the other as long as they share the same set of variables, though not necessarily in the same order. One occasion to use a different BDD manager is for on-the-fly counterexample generation when concurrently solving multiple properties, to avoid trace generation from triggering a dynamic variable ordering which hurts continued reachability analysis. An initial ordering of the variables is computed using the interleaved approach described in [22]. We use the transition relation partitioning techniques of [2] by default. Cutpointing is supported in both the transition relation and the BDD representing the property.

A number of optimizations are used during the reachability computation to reduce BDD size, including backward and forward pruning of the transition relation as described in [23]. In addition, we make use of the BDD reduction operations described in [15] when computing frontiers.

Figures 1 and 2 summarize our experiments for runtime and memory, respectively, of performing a reachability computation after light-weight logic optimization techniques, with and without our dynamic hint generation approach. Note that we forced a complete reachability computation on each of

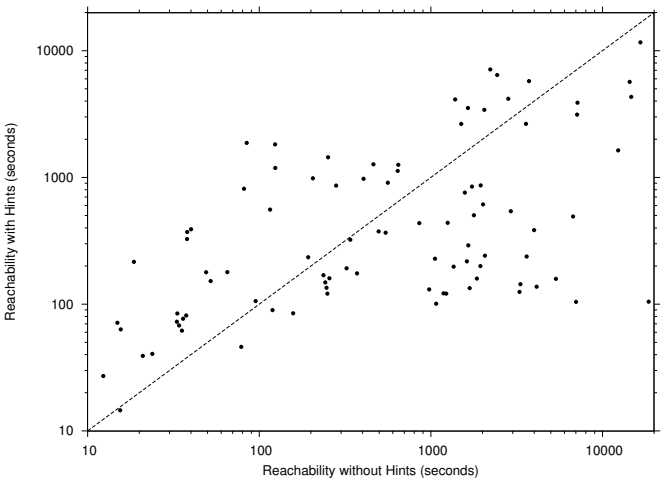

Fig. 1. Reachability Computation Runtime with vs. without Hints

these, even if an on-the-fly failure could have enabled early termination. None of these experiments exhausted memory, though there were timeouts which are omitted from Figure 2. These results demonstrate that hints do introduce a computational overhead for simpler problems – primarily those which complete within several minutes. However, for a majority of the complex problems, hints significantly improve runtime and memory requirements. In fact, the benefit achieved by hints is largely proportional to the complexity of the verification problems: those which would otherwise require approximately 1000 seconds often speed up to within one order of magnitude, and those which otherwise require approximately 10000 seconds often speed up to approaching two orders of magnitude. There are several examples which timeout without hints, yet which complete with hints. This is a promising result, as the practical need to improve runtimes of complex, if not "otherwise unsolvable," problems is at the forefront of industrial relevance.

Note that the memory plot exhibits a fair amount of clustering of data points, caused by thresholds at which dynamic

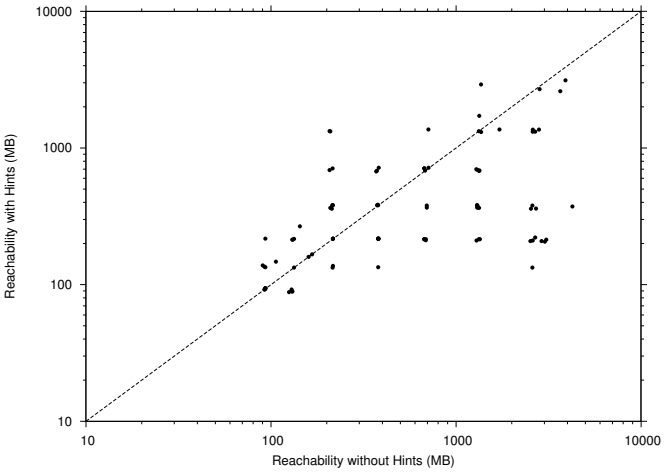

Fig. 2. Reachability Computation Memory with vs. without hints

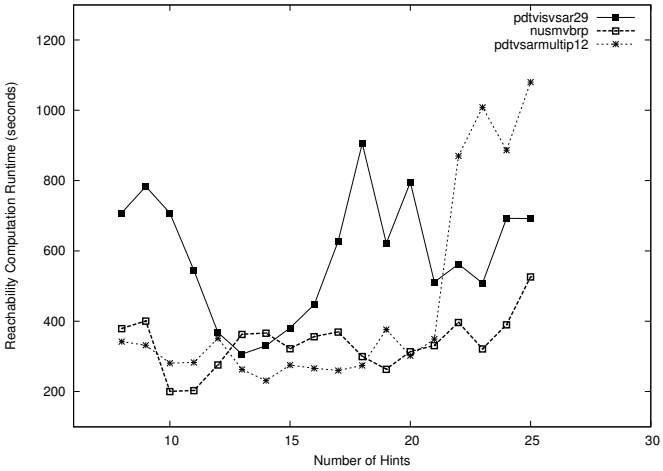

Fig. 3. Impact of number of hints on runtime

variable ordering is invoked. Similarly to the runtime analysis, there are frequent benefits of one to two orders of magnitude for more complex problems, though some penalties primarily for simpler problems.

The observation that hints often entail an overhead for simpler problems prompts the question of whether the introduction of hints should be delayed until a larger threshold. We performed significant experimentation to assess the validity of this strategy and found that delaying the onset of hints almost uniformly hurt complex problems. Invoking hints when the reached set has grown to millions of BDD nodes often requires more expensive dynamic variable ordering calls, and applying a large number of hints at that point degrades performance. Adjusting our hint heuristics to improve performance of simpler problems would compromise performance on complex problems, where hints yield the biggest advantage. We also note that in an industrial-strength multi-engine verification flow, slowdown of simpler problems is not as serious of a concern since within that runtime, one likely would have spent comparable resources trying various alternate algorithms such as bounded model checking and IC3.

Figure 3 illustrates the impact of number of generated hints on runtime. In the experiment, reachability analysis was performed varying the number of hints from 8 to 25. These experiments demonstrate that it is disadvantageous to use too few or too many hints. With too few, the hints do not adequately simplify image computation, while with too many there is too large of an overall increase in the number of computed images. Recall from Algorithm 2 that we use a parameter *var_limit* to limit the maximum number of literals that may be included in a hint. Practically, we have found it useful to bound this parameter based upon netlist size (a percentage of the total number of inputs and state elements) to preclude introducing too many hints for smaller problems.

To justify the importance of highly-tuned reachability engine in a state-of-the-art verification tool, we ran light-weight logic optimization techniques followed by our implementation of IC3 [4], interpolation [3], and k-step unique-state induction engines [24], which are all highly-tuned and competitive with the best academic solvers. Of the 92 benchmarks, 11 resulted in counterexamples for all their properties hence the SAT-based techniques terminated upon finding these counterexamples, whereas we disabled early-termination in our reachability engine for these experiments. We thus omit these 11 from the following experiments. We illustrate the runtimes for the remaining 81 benchmarks using reachability without hints, reachability with hints, and the three SAT-based techniques mentioned above, in Table I. The runtime for the technique which solves most quickly is shown in bold.

Note that 5 benchmarks are solved most quickly using reachability with hints, whereas 14 are solved most quickly using reachability without hints. This collectively represents 23.4% of the benchmarks which are solved more quickly using BDD-based reachability than SAT-based techniques, often by orders of magnitude. The converse is not surprisingly true as well; the SAT-based techniques inherently reason about the design in an abstract manner vs. precisely computing the reachable states, often resulting in much faster runtimes. If we preceded reachability computation by abstraction techniques such as localization [8] or phase abstraction [10], or sequential reductions such as redundancy removal [9] or retiming [11], this would have enabled reachability computation on a larger fragment of the benchmark suite, and have narrowed the precise vs. abstract penalty imposed by these experiments. IC3 solves 27 most quickly (33.3%), and induction solves 35 (43.2%)most quickly, where we broke ties in favor of induction given the maturity and simplicity of that technique. Interpolation was somewhat surprisingly not the winning engine in any of these benchmarks. While we have found IC3 to very often outperform interpolation in practice, there are industrial cases where interpolation is the winning technique.

To further emphasize the role of reachability analysis and hints, we note the following.

1. We only included examples for which hints were generated in these experiments, thus omitted numerous easy wins for reachability in this benchmark suite.

2. Reachability with hints solved all these benchmarks, whereas reachability without hints has 3 timeouts, IC3 has 13, and interpolation and induction each have 41.

3. Reachability using hints outperformed reachability without hints in 46 of these examples (56.8%). As per Figure 1, hints offers greater benefits for more complex benchmarks; if we increase the timeout period, our practical experience is that hints and BDD-based techniques overall play a larger role.

4. In a state-of-the verification tool, lighter-weight algorithms are often leveraged with a moderate resource limit before heavier-weight techniques. If we discount benchmarks solvable within 10 seconds, only 29 of these benchmarks remain: 19 are solved most quickly using reachability (65.5%), 7 using IC3 (24.1%), and 3 using induction (10.3%).

5. Cumulative runtime for reachability with hints is much lesser than for the other techniques while counting timeouts at 4 hours, and even outperforms reachability without hints by a factor of 1.77 when discounting the 3 timeouts for the latter.

Benchmark Name	Inputs / Ands / Registers	Reachability w/o Hints	Reachability with Hints	IC3	Interpolation	Induction
6s4	209 / 2448 / 201	**405.4** (3918)	976.1 (7738)	TO	TO	6081.3
6s48	72 / 796 / 66	7151.8 (17)	**3884.1** (31)	TO	TO	TO
6s48p0	72 / 795 / 66	2434.8 (17)	10620.2 (57)	TO	TO	**430.4**
6s52	35 / 1226 / 207	**65.1** (52)	179.1 (245)	TO	TO	TO
6s53	35 / 1228 / 207	**115.7** (259)	557.6 (1052)	TO	TO	TO
bjrb07amba10andenv	23 / 62516 / 58	**36.0** (41)	76.8 (69)	240.1	TO	TO
bjrb07amba7andenv	17 / 22312 / 45	**15.6** (33)	63.2 (98)	26.2	TO	TO
bjrb07amba9andenv	21 / 45216 / 52	**33.3** (41)	84.5 (188)	75.1	TO	TO
boblivea	5 / 540 / 102	7117.3 (49)	3130.9 (104)	**8.0**	TO	TO
boblivear	5 / 321 / 77	2220.7 (49)	7109.3 (119)	**68.8**	7638.8	TO
eijkbs1512	29 / 817 / 123	TO (544)	5675.1 (1300)	**1.3**	TO	TO
eijkbs3330	37 / 1407 / 166	TO (4)	11637.8 (48)	**23.2**	TO	TO
intel055	222 / 3847 / 124	561.5 (24)	908.6 (75)	**16.6**	TO	TO
intel059	280 / 1955 / 140	646.2 (24)	1257.5 (83)	**13.7**	TO	TO
intel063	288 / 1773 / 240	34.1 (7)	67.9 (19)	**0.6**	0.7	TO
nusmvbrp	11 / 378 / 51	1952.2 (57)	864.8 (158)	**2.2**	3492.6	TO
nusmvdme1d3multi	54 / 236 / 61	TO (38)	**104.6** (270)	TO	TO	TO
nusmvqueue	82 / 1200 / 84	**462.4** (45)	1270.0 (136)	5246.5	TO	TO
pdtfifo1to0	6 / 860 / 142	**251.6** (62)	1440.6 (398)	5517.9	TO	TO
pdtpmsbufferalloc	6 / 477 / 66	78.6 (31)	**46.0** (57)	TO	TO	TO
pdtpmseisenberg	3 / 1765 / 125	544.9 (90)	**366.2** (223)	TO	TO	TO
pdtpmsfpmult	17 / 929 / 166	49.0 (7)	178.6 (37)	**1.0**	14176.8	TO
pdtpmsgigamax	22 / 681 / 85	6.9 (8)	22.5 (37)	**0.3**	4.5	TO
pdtpmsns2	16 / 1742 / 278	339.1 (16)	322.7 (38)	**56.2**	TO	TO
pdtpmstimeout	10 / 922 / 80	**12.3** (28)	27.1 (64)	TO	TO	TO
pdtswvibs8x8p1	9 / 1039 / 96	81.5 (83)	813.3 (523)	5.1	47.3	**4.6**
pdtswvqis10x6p1	7 / 1609 / 92	124.0 (99)	1187.1 (487)	**81.0**	TO	TO
pdtswvqis10x6p2	7 / 1771 / 88	**84.5** (99)	1871.4 (489)	TO	TO	TO
pdtswvqis8x8p1	9 / 1685 / 98	**18.6** (79)	215.5 (325)	48.6	2492.3	6612.2
pdtswvqis8x8p2	9 / 1866 / 94	**37.9** (79)	326.4 (349)	TO	TO	TO
pdtswvrod6x8p1	9 / 1314 / 74	40.0 (132)	**39.2** (748)	100.4	TO	TO
pdtswvrod6x8p2	9 / 1331 / 70	**38.0** (132)	371.0 (772)	TO	TO	TO
pdtswvroz10x6p1	7 / 926 / 73	52.1 (87)	152.1 (367)	**3.5**	3413.2	27.6
pdtswvroz10x6p2	7 / 941 / 73	192.9 (87)	234.6 (391)	**13.3**	TO	2262.8
pdtswvsam6x8p4	9 / 2003 / 116	1385.7 (69)	4125.2 (453)	TO	TO	**264.9**
pdtswvtma6x4p2	5 / 457 / 42	37.5 (60)	81.4 (159)	92.4	TO	**8.3**
pdtswvtma6x4p3	5 / 459 / 42	**14.9** (60)	24.1 (164)	918.4	TO	42.2
pdtswvtma6x6p1	7 / 640 / 58	205.4 (60)	984.1 (235)	48.6	904.2	**6.7**
pdtswvtma6x6p2	7 / 607 / 58	280.3 (60)	861.9 (242)	1002.1	TO	**49.0**
pdtvisns3p00	21 / 1210 / 100	371.3 (25)	174.8 (60)	**3.6**	TO	TO
pdtvisns3p01	21 / 1220 / 100	157.7 (25)	84.7 (83)	**5.6**	TO	TO
pdtvisns3p02	21 / 1206 / 100	322.8 (25)	191.8 (130)	**3.0**	TO	TO
pdtvisns3p03	21 / 1200 / 100	249.4 (25)	121.1 (43)	**2.2**	TO	TO
pdtvisns3p04	21 / 1183 / 100	246.5 (25)	134.6 (146)	**3.9**	TO	TO
pdtvisns3p05	21 / 1179 / 100	95.3 (25)	105.7 (150)	**3.3**	TO	TO
pdtvisns3p06	21 / 1181 / 100	256.4 (25)	159.9 (42)	**6.7**	TO	TO
pdtvisns3p07	21 / 1190 / 100	236.6 (25)	169.2 (78)	**3.9**	TO	TO
pdtvisns3p08	21 / 1176 / 100	242.4 (25)	148.5 (127)	**0.8**	TO	TO
pdtvisns3p09	21 / 1178 / 100	119.5 (25)	89.7 (90)	**0.9**	TO	TO
pdtvissoap1	11 / 1510 / 124	23.8 (46)	40.5 (77)	**1.7**	TO	TO
pdtvissoap2	11 / 1548 / 124	21.0 (46)	39.0 (118)	**1.2**	149.7	TO
pdtvisvsar27	17 / 898 / 62	1622.1 (36)	217.5 (192)	0.1	0.3	**0.1**
pdtvisvsar29	17 / 1081 / 61	3994.4 (36)	383.9 (111)	120.2	5049.6	**0.3**
pdtvsarmultip	17 / 1473 / 77	2922.6 (36)	541.6 (116)	65.2	1891.5	**0.8**
pdtvsarmultip00	17 / 860 / 61	1683.8 (36)	133.8 (139)	0.1	0.2	**0.1**
pdtvsarmultip03	17 / 873 / 61	1942.7 (36)	199.7 (118)	0.1	0.1	**0.1**
pdtvsarmultip04	17 / 873 / 61	3285.1 (36)	125.1 (237)	0.1	0.1	**0.1**
pdtvsarmultip05	17 / 850 / 61	855.9 (36)	914.6 (187)	0.5	0.3	**0.1**
pdtvsarmultip06	17 / 862 / 61	7017.1 (36)	104.3 (261)	0.2	0.2	**0.1**
pdtvsarmultip07	17 / 890 / 61	4134.4 (36)	137.3 (94)	0.4	0.4	**0.1**
pdtvsarmultip08	17 / 857 / 61	3584.0 (36)	2650.1 (478)	0.1	0.2	**0.1**
pdtvsarmultip09	17 / 852 / 61	1653.3 (36)	291.2 (180)	0.1	0.2	**0.1**
pdtvsarmultip10	17 / 852 / 61	2065.2 (36)	241.1 (131)	0.4	0.3	**0.1**
pdtvsarmultip11	17 / 870 / 62	1252.5 (36)	438.4 (132)	0.1	0.1	**0.1**
pdtvsarmultip12	17 / 866 / 61	1229.7 (36)	121.1 (214)	0.1	0.1	**0.1**
pdtvsarmultip13	17 / 869 / 64	3613.9 (36)	237.8 (173)	0.1	0.1	**0.1**
pdtvsarmultip14	17 / 900 / 61	1074.4 (36)	100.9 (170)	0.1	0.1	**0.1**
pdtvsarmultip15	17 / 880 / 61	1057.6 (36)	228.4 (124)	0.1	0.1	**0.1**
pdtvsarmultip17	17 / 879 / 63	3326.2 (36)	143.7 (121)	0.1	0.1	**0.1**
pdtvsarmultip19	17 / 876 / 62	977.3 (36)	130.7 (109)	0.1	0.1	**0.1**

continued on next page

Benchmark Name	Inputs / Ands / Registers	Reachability w/o Hints	Reachability with Hints	IC3	Interpolation	Induction
pdtvsarmultip21	17 / 874 / 62	496.3 (36)	375.0 (254)	0.1	0.1	**0.1**
pdtvsarmultip22	17 / 846 / 62	1356.7 (36)	197.6 (115)	0.1	0.1	**0.1**
pdtvsarmultip23	17 / 865 / 62	1852.7 (36)	159.1 (121)	0.1	0.1	**0.1**
pdtvsarmultip24	17 / 861 / 62	5350.6 (36)	158.4 (170)	0.1	0.1	**0.1**
pdtvsarmultip26	17 / 865 / 62	2016.4 (36)	612.8 (234)	0.1	0.1	**0.1**
pdtvsarmultip27	17 / 882 / 62	1186.4 (36)	121.8 (220)	0.1	0.3	**0.1**
pdtvsarmultip29	17 / 1064 / 61	1735.2 (36)	1747.5 (176)	802.6	2636.6	**0.5**
pdtvsarmultip31	17 / 1002 / 62	1781.9 (36)	502.7 (167)	0.1	0.1	**0.1**
pdtvsarmultip32	17 / 983 / 61	6739.3 (36)	491.7 (179)	25.5	86.0	**0.2**
pj2009	304 / 7498 / 269	3734.3 (31)	5748.3 (159)	**4.3**	20.4	TO
sm98a7multi	82 / 3337 / 89	12346.1 (37)	1632.9 (161)	2.8	1.4	**1.1**
Cumulative		158558.6	**82707.6**	201872.0	632409.5	606195.3

TABLE I. Runtimes for various proof engines. Column 2 provides size of the benchmark after light-weight reductions. Subsequent columns list runtimes in seconds; TO refers to 4-hour timeout. The number in parenthesis in Columns 3 and 4 indicates the number of image computations until fixedpoint or TO.

While points **2** and **3** above are skewed by the selection of benchmarks for which reachability in some form converges, these experiments do emphasize that reachability often outperforms SAT-based techniques, and hints increase the overall robustness of reachability computation.

V. Conclusion

Despite many advances in SAT-based proof techniques, BDD-based reachability remains a critical technology which is able to significantly outperform alternative proof techniques on numerous classes of problems. In this paper, we introduce a novel technique to increase the scalability of reachability computation: automated dynamic netlist-based hint generation. Experiments demonstrate that this approach is able to reduce resources well over an order of magnitude on many complex verification problems, outperforming SAT-based techniques in many cases. These techniques have played a vital role in revitalizing reachability analysis as a core industrial-strength proof technique in our multi-algorithm verification toolsuite.

References

[1] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations," in *VLSI*, pp. 49–58, Aug. 1991.

[2] I.-H. Moon, G. D. Hachtel, and F. Somenzi, "Border-block triangular form and conjunction schedule in image computation," in *FMCAD*, Nov. 2000.

[3] K. McMillan, "Interpolation and SAT-based model checking," in *CAV*, 2003.

[4] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, Jan. 2011.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, March 1999.

[6] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *ICCAD*, Nov. 2000.

[7] I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi, "To split or to conjoin: the question in image computation," in *DAC*, June 2000.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, July 2000.

[9] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification," in *DATE*, 2009.

[10] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.

[11] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *CAV*, July 2001.

[12] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[13] Berkeley Logic and Synthesis Group, *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/alanmi/abc.

[14] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," *Formal Methods in System Design*, vol. 39, no. 2, pp. 205–227, 2011.

[15] P. A. Beerel, J. R. Burch, and K. L. McMillan, "Sibling-substitution-based BDD minimization using don't cares," *TCAD*, vol. 19, Jan. 2000.

[16] K. Ravi and F. Somenzi, "Hints to accelerate symbolic traversal," in *CHARME*, Oct. 1999.

[17] D. Ward and F. Somenzi, "Automatic generation of hints for symbolic traversal," in *CHARME*, Sept. 2005.

[18] D. Ward and F. Somenzi, "Decomposing image computation for symbolic reachability analysis using control flow information," in *ICCAD*, Nov. 2006.

[19] K. Ravi and F. Somenzi, "High-density reachability analysis," in *ICCAD*, Nov. 1995.

[20] Hardware Model Checking Competition 2011. http://fmv.jku.at/hwmcc11.

[21] G. Janssen, "Design of a pointerless BDD package.," in *IWLS*, 2001.

[22] H. Fujii, G. Ootomo, and C. Hori, "Interleaving based variable ordering methods for ordered binary decision diagrams," in *ICCAD*, Nov. 1993.

[23] H. Jin, A. Kuehlmann, and F. Somenzi, "Fine-grain conjunction scheduling for symbolic reachability analysis," in *Tools and Algos. Construction and Analysis of Systems*, April 2002.

[24] N. Eén and N. Sörennson, "Temporal induction by incremental SAT solving," in *Workshop on Bounded Model Checking*, 2003.