Noname manuscript No.
(will be inserted by the editor)

Scalable Reachability Analysis via Automated Dynamic Netlist-Based Hint Generation

Jiazhao Xu and Mark Williams and Hari Mony and 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.

1 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 [5–7], a variety of simplification and abstraction techniques to reduce problem complexity [8–10], 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

Address(es) of author(s) should be given

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; it is certainly not the case that SAT-based methods have subsumed the utility of BDD-based methods, and each technique has its strengths and weaknesses. While it is difficult to precisely characterize problems which favor BDDs, examples include those which require extremely deep sequential analysis (e.g., with counterexamples tens of thousands of timesteps long), and designs with a large number of highly-correlated state variables that are difficult to model with succinct invariants. A well-tuned BDD-based reachability engine is thus an essential component of a state-of-the-art verification tool for overall robustness.

Numerous techniques have been developed to boost the scalability of BDD-based reachability engines. Examples include use of a partitioned transition relation (TR) instead of a monolithic representation [1], advanced quantification and conjunction scheduling based upon metrics such as variable dependency [2], and heuristics to balance splitting and conjoining strategies [7]. The application of BDD-reduction operators such as *bdd_constrain*, *bdd_restrict* and *bdd_compact* on the transition relation [15] 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 by computing states reachable (along the constrained transition relation) from those reached using prior hints, finally restoring the original transition relation to ensure completeness. Despite requiring more (pre-)image computation steps, this compaction of intermediate BDDs often enables a significant reduction in peak memory requirements. It also invokes less expensive dynamic variable orderings thus often in turn reduces runtime for complex problems. In extreme cases it provides a solution for problems which would otherwise exhaust reasonable memory or time limitations. As noted in [16], as concurs with our practical experience: even *arbitrary* hints often reduce complexity for difficult problems. This preliminary work was focused upon using 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 applicability since they require a high-level design format which may not be available. They 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, or more generally, of designs lacking proper behavioral syntax. Such a methodology is additionally a practical challenge to apply in application domains such as sequential equivalence checking, which may require analysis of post-synthesis netlists. In contrast, our work is focused upon generating a high-quality set of hints from arbitrary netlist representations, and is triggered on-demand only when reachability computation exceeds a resource threshold.

Algorithm 1 Breadth-First Reachability Analysis

```
1: function FORWARDREACH(TR, init_states)
2:     frontier = reached = init_states
3:     while (true) do
4:         image = compute_image(TR, frontier)
5:         frontier = compute_frontier(image, reached)    // subtract reached from image
6:         if (frontier is empty) then break
7:         end if
8:         reached = bdd_or(reached, frontier)
9:     end while
10: end function
```

It has been successfully applied in sequential equivalence checking as well as in the property checking domain. It also fully exploits a transformation-based verification paradigm where the netlist may be reduced, abstracted, rewritten, and decomposed prior to deploying a reachability engine. The automatically-computed hints are derived dynamically from the transformed netlist during reachability computation instead of from the original netlist. In contrast, hints pre-computed from the original netlist may be ineffective if not completely irrelevant on the transformed netlist upon which reachability analysis is to be performed.

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. Our specific contributions, as detailed in Section 3, 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 4 confirm that they often significantly boost performance for complex problems, and in many cases outperform SAT-based techniques.

2 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* $TR(\bar{x}, \bar{i}, \bar{y})$ 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 $\exists \bar{i}. \exists \bar{x}. TR(\bar{x}, \bar{i}, \bar{y}) \wedge s$. A *reachability computation* may be performed by first setting the

Algorithm 2 Reachability using Hints

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

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 as shown in Algorithm 1

3 Enhanced Reachability Algorithms

In this section we present our automated hint generation algorithms. Algorithm 2 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 practical limitations of the traditional use of hints which we address in this paper.

i) Hint generation automation only for problems of suitable Verilog syntax [17] is limiting in practice. The manual specification of hints for general classes of netlists often diminishes their applicability and utility. We thus introduce in Section 3.1 an effective automated hint generation algorithm which operates directly upon the transition relation, and in Section 3.2 an algorithm to iterate through the generated hints. These algorithms control the sequence of hints provided by Step 4 of Algorithm 2 in our fully-automated framework.

ii) 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 over unconstrained image computations. To address this issue, we introduce in Section 3.3 a reachability framework which introduces hints upon demand, only when BDDs exceed configurable thresholds, thus avoiding using hints on simple problems.

iii) In some rare cases, hints may result in convergence problems for a 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 computation, slowing overall progress. We thus introduce in Section 3.4 a mechanism to truncate the use of a specific hint prior to fixedpoint if necessary, for overall robustness.

Algorithm 3 Hint Introduction Algorithm

```
    function GENERATE_HINTS(TR, hint_literals, reached, reduction_limit, var_limit)
 2:     vars = all variables that are not in hint_literals
        for all BDD variable var in vars do
 4:         compute the rank of each var; note best candidate between the positive and negative literals
            add best rank to array ranks
 6:     end for

 8:     sort ranks
        // Use ranks info to generate the initial hint cube
10:     hint_cube = bdd_1 ∧ ⋀ hint_literals // Form cube for already-selected hints
        for all rank in ranks do
12:         literal = best candidate from rank
            new_cube = bdd_and(hint_cube, literal);
14:         if (new_cube contradicts TR or reached) then
                invalidate chosen polarity for the variable; compute rank for opposite polarity
16:             re-sort ranks and goto line 10
            else
18:             hint_cube = new_cube
                hint_literals = hint_literals ∪ literal
20:             prune rank from ranks
                compute TR size reduction of TR from new_cube
22:             if ((TR reduction exceeds reduction_limit) or (cardinality of hint_literals exceeds var_limit))
    then
                    break
24:             end if
            end if
26:     end for
        return hint_literals
28: end function
```

In this paper we present the use of hints for forward reachability analysis. For forward reachability, hints constrain present-state variables of the transition relation. This means that an image may include states that do not satisfy the current hint, though successors of such states will not be considered until the corresponding hint is eliminated. Use of hints for backward reachability is also possible in a straight-forward manner, by symmetrically swapping their transition-relation constraints from current- to next-state variables.

3.1 Automated Hint Generation Algorithm

Algorithm 3 outlines our automated hint introduction 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. Several heuristics may be used to select which input and state literals will form effective hints, as will be discussed below.

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 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 hint cube which contradicts the transition relation or *reached* set, which begins as the initial states. We thus perform a consistency-check 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: *reduction_limit* measures the degree to which the given *hint_cube* reduces the *TR*, and *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 *hint_limit* of between 10 and 15 literals yields the best results for larger netlists (see further discussion in Section 4 and Figure 3).

It is noteworthy that the generated hints are highly dependent upon BDD variable ordering. This algorithm may be called multiple times in an overall reachability framework to iteratively constrain a complex problem, possibly adding additional hints to a non-empty set of previously-generated hints. As it is very likely that dynamic variable ordering will have been invoked between these calls, the newly-added hints will always reflect the best choices under the current ordering.

3.2 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 directing the computation starting from applying a most restrictive constraint and gradually relaxing those constraints. These observations were 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 4.

When reachability analysis exceeds a complexity threshold and a set of hint literals have been generated using function *generate_hints* from Algorithm 3, this algorithm determines the sequence of hints which will be applied from that set. The first hint to be applied after generating literals will be the conjunction of all hint literals. The remaining sequence of hints iteratively eliminates one literal from that set. Note that the sequence of applied hints is *dynamically* determined, vs. merely deciding a fixed order when hint literals are generated via Algorithm 3, to exploit the fact that variable ordering may have changed between those points.

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_h of the current hint h with the reached set, and checking if I_h contains the conjunction

Algorithm 4 Hint Successor Algorithm

```
    function GET_NEXT_HINT(TR, hints, first, reached)
        cur_hint = true ∧ ⋀ hints
 3:     if (first) then
            return cur_hint
        end if
 6:     re-rank and re-sort hints
        // Vacuous hint detection
        while (|hints|) do
 9:         remove lowest-rank literal from hints
            next_hint = true ∧ ⋀ hints
            if ((next_hint ∧ reached) ⊆ (cur_hint ∧ reached)) then
12:             continue // next_hint is vacuous
            else
                return next_hint
15:         end if
        end while
        return true
18: end function
```

I_c of the candidate next hint c with the reached set. If so, the candidate next hint c is vacuous and we skip it by removing another 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.

3.3 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 does not inadequately reduce complexity to 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 will be generated to mitigate the BDD explosion, and a constrained image computation will be attempted. 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 often effective) to avoid future hints from being triggered too frequently on problems that intrinsically need large BDDs.

3.4 Hint Truncation

In a traditional hint application as per Algorithm 2, a full fixedpoint of states reachable under the corresponding hint-constrained transition relation is performed for each hint. However,

Algorithm 5 **Enhanced Reachability using Dynamic Automated Hints**

```
1: function FORWARDREACH(TR, init_states, var_limit, bdd_threshold, bdd_growth_factor,
   depth_threshold, reduction_limit)
2:     frontier = 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:         while (true) do
10:            image = compute_image(hint_TR, frontier, bdd_threshold)
11:            if (image == UNKNOWN) then // computation aborted due to bdd_threshold
12:                bdd_threshold = bdd_threshold * bdd_growth_factor
13:                hints = GENERATE_HINTS(TR, hints, reached, reduction_limit, var_limit)
14:                first = true
15:                break
16:            end if
17:            hint_iters++
18:            if (hint_iters ≥ depth_threshold) then break // move to next hint; optionally increase threshold
19:            end if
20:            frontier = compute_frontier(image, reached)
21:            if (frontier is empty) then
22:                if (hints == emptyset) then return // fixedpoint complete
23:                end if
24:                break // move onto next hint
25:            end if
26:            reached = bdd_or(reached, frontier)
27:        end while
28:    end while
29: end function
```

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 3. 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.

3.5 Overall Algorithm for Reachability under Dynamic Hint Introduction

In this section we provide our overall reachability framework depicted in Algorithm 5, combining aspects described in prior sections. This algorithm details the various steps which would be coalesced as *get_hint* in Step 4 of Algorithm 2. The primary differences are: **(1)** automated introduction of hints in Step 13 (Algorithm 3 from Section 3.1); **(2)** dynamically-prioritized iteration among the generated hints, taking into account current variable ordering in Step 5 (Algorithm 4 from Section 3.2); **(3)** dynamic triggering of hint introduction in Step 11 (Section 3.3); and **(4)** truncation of hint-constrained reachability if too many image computations are required in Step 18 (Section 3.4).

4 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 cap, as an arbitrary yet fairly aggressive resource setting (the Hardware Model Checking Competition limits to 15 minutes); 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, we are unable to compare our approach against those of [17]. Furthermore, the benchmarks used in [17] are a small set that are not publicly available, hence we restrict our focus to those of [20].

We implemented our techniques in the reachability engine included in the IBM verification tool *SixthSense* [12]. This 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 prevent trace generation from spoiling a good variable order for 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. Note that we forced an entire reachability computation on each of these, even if an on-the-fly failure could have been detected earlier.

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. Those which timed-out or exhausted memory in either approach are omitted from these plots. There are several examples which exceed resource limitations without hints for which we are able to complete reachability using hints, though none of these runs exhausted resources with hints yet completed without hints. As demonstrated, 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. 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 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 for simpler problems.

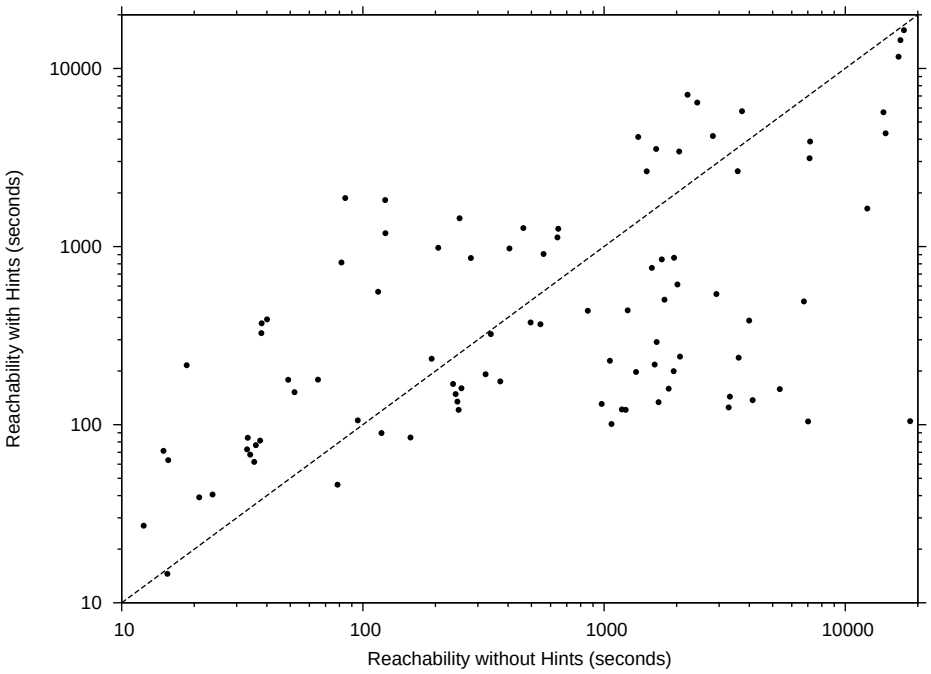

Fig. 1 Reachability Computation Runtime with vs. without Hints

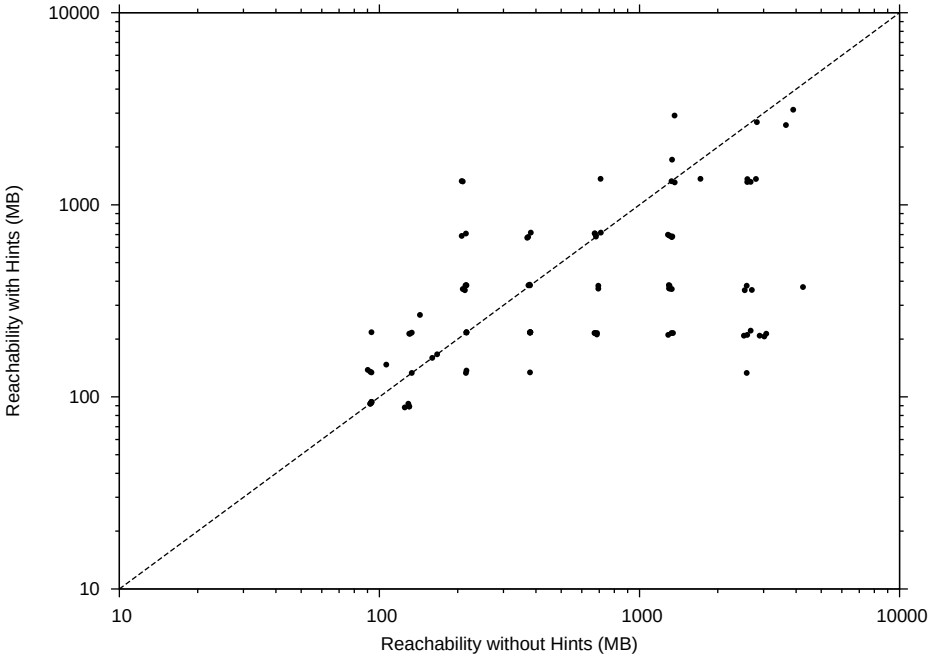

Fig. 2 Reachability Computation Memory with vs. without Hints

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

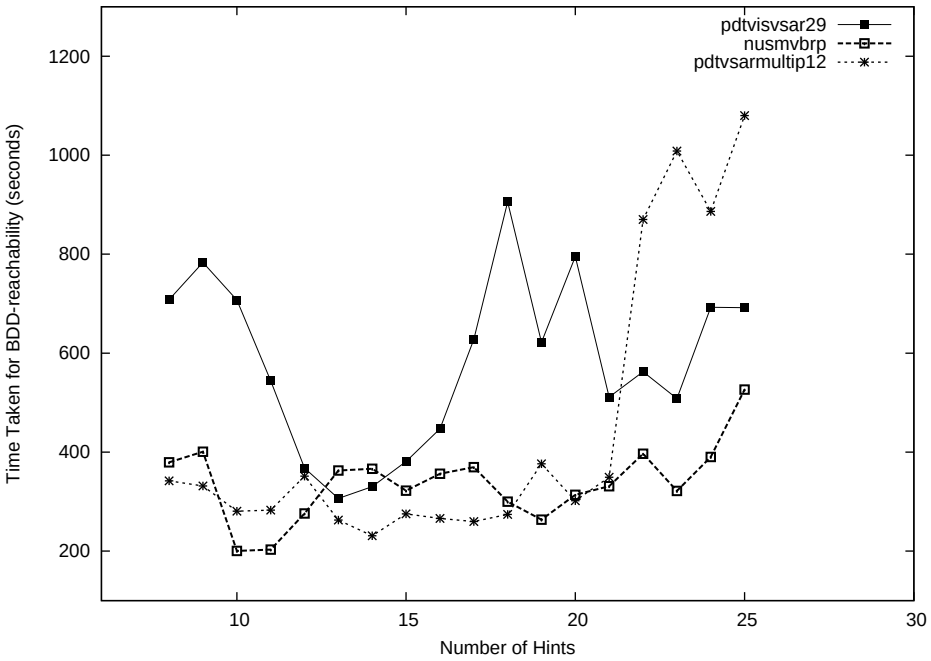

Fig. 3 Impact of number of hints on runtime

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 BDD has grown to millions of nodes often requires more expensive dynamic variable ordering calls and applying a large number of hints will further degrade the performance. We observe that in a practical industrial-strength multi-engine flow, slowdown of simpler problems is not as serious of a concern since within that runtime, one likely would have spent comparable time trying various alternate algorithms such as bounded model checking and IC3. Similarly, one could use the strategy of performing reachability without hints using small resource limits and if resources are exceeded, to proceed using reachability with hints to better manage the complexity of the problem – or to attempt both in parallel.

Figure 3 illustrates the impact of the number of hints generated on runtime. In the experiment, reachability analysis was performed by varying the number of hints from 8 to 25. Experiments demonstrate that it is disadvantageous to have too few or too many hints. With too few, the hints cannot make an adequate impact on simplifying reachability analysis, while with too many there is too much overhead to complete the increased number of image computations. We have two options to achieve a better selection of this number: the *hint-variable-limit* will dictate an upper bound on the number of literals allowed in a hint; the *hint-variable-percentage* will ensure that the number of literals in a hint does not exceed a percentage of the total number of inputs and state variables.

To justify the importance of highly-tuned reachability analysis in a state-of-the-art 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 and 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 1. The runtime for the technique which solves most quickly is shown in bold. Of the remaining 81 cases, 14 more could not be solved by either IC3 or interpolation within a 4 hour timeout, whereas each of these could be solved by our reachability flow – some as quickly as 12 seconds. We thus note that 18.4% of these provable benchmarks were solved more quickly by BDD-based reachability than contemporary SAT-based proof techniques, often by orders of magnitude runtime. We furthermore note that these SAT-based proof techniques intrinsically yield abstract overapproximations of the reachable states of a netlist, whereas reachability computation may only do so if preceded by an abstraction technique such as localization – which was not included in these experiments. More of these benchmarks would have been candidates for reachability analysis had we included localization or other sequential reductions in these experiments.

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 problems 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 by any method, 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**

continued on next page

Benchmark Name	Inputs / Ands / Registers	Reachability w/o Hints	Reachability with Hints	IC3	Interpolation	Induction
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**
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**
# Completed		78	**81**	68	40	40
Cumulative Time		158558.6	**82707.6**	201872.0	632409.5	606195.3

Table 1 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, and is counted as such in the final *cumulative time* row. 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.

5 Comparison of Dynamic Hints and Dynamic Variable Splitting

The dynamic hint generation procedure is invoked at an image computation iteration where the intermediate BDD of the partial image exceeds resource limitations. Another method to deal with BDD size explosion is to apply dynamic variable splitting [7]. The idea of dynamic variable splitting is to choose a BDD variable, for example z, and to split the image calculation into two parts through BDD cofactor operations: one part of the image calculation assumes the value $z = 1$, and the other part assumes $z = 0$.

$$\begin{aligned} & \exists \bar{x}.\exists \bar{i}.TR(\bar{x},\bar{i},\bar{y}) \wedge s(\bar{x}) \\ = \; & \exists \{\bar{x}\backslash z\}, \exists \{\bar{i}\backslash z\}.TR_{z=1}(\bar{x},\bar{i},\bar{y}) \wedge s_{z=1}(\bar{x}) \vee \\ & \exists \{\bar{x}\backslash z\}.\exists \{\bar{i}\backslash z\}.TR_{z=0}(\bar{x},\bar{i},\bar{y}) \wedge s_{z=0}(\bar{x}) \end{aligned} \tag{1}$$

In the above equation, $TR_{z=1}$ and $s_{z=1}$ respectively represent the positive cofactors of the transition relation TR and the current state s, and $TR_{z=0}$ and $s_{z=0}$ represent the negative cofactors of the transition relation TR and the current state s. The final image is the union of the subimage computed from the positive cofactor and the subimage computed from the negative cofactor. The BDD operation of the union does not always have to be carried out; instead, the set of two subimages may be used to implicitly represent the resulting image.

We compare dynamic hint generation with the alternative approach of dynamic variable splitting to evaluate their relative effectiveness in handling BDD explosion in an image step. Both methods use heuristics to specify a threshold regarding when an image step is deemed to be too difficult for standard breadth-first search. Our experiments show that the threshold for invoking dynamic hint generation should not be set too high, otherwise the initial BDDs become too large and hints do not adequately simplify the image computation. The threshold for invoking dynamic variable splitting, on the other hand, should not be set too low, else dynamic variable splitting will be invoked unnecessarily often and thus slow down the image computation. Practically, if both methods are used synergistically during reachability analysis, dynamic hint generation should often be invoked first because it has a lower invocation threshold.

Both methods are very sensitive to BDD variable ordering due to the following factors:

i) BDD sizes are highly dependent on BDD variable ordering, and BDD sizes impact when the threshold to invoke these methods is met.
ii) The computation of the initial hint cube (via choosing the hint literals) in dynamic hint generation is dependent upon the BDD variable ordering at the time of invocation. Similarly, the selection of the variable to split upon in dynamic variable splitting also depends upon BDD variable ordering. The heuristics for choosing the hint variables and the heuristics for choosing the splitting variables share some common features. We rank the BDD variables according to their influence on reducing the transition relation given the current BDD ordering, and choose the highest-ranked variables. The ranking mechanism can be further tailored. For hints, the BDD literals that can reduce the transition relation the most are considered first to enable efficient fixedpoints. For dynamic variable splitting, [7] suggests that the variable which appears in the support of the largest number of conjuncts is favored. In our implementation, we primarily use the BDD *use count* in this choice, followed by the reduction on the TR. These candidates can be chosen from the input variables as well as the state variables.

In our implementation, dynamic BDD variable reordering is enabled throughout the image calculation except during hint variable selection and splitting variable selection. Some implementations of dynamic variable splitting might disable BDD variable reordering within the image step to avoid creating variable orders during cofactoring which are ineffective at later stages in the computation. The effect of BDD ordering at the time that dynamic variable ordering is invoked would thus be more temporary because it will only affect the selection of the splitting variable and the subimage computation at that image step. In comparison, the effect of the BDD ordering in dynamic hint generation is more enduring because the initial hint cube is usually computed only once and it will affect the remainder of the reachability computation.

The computation paths between these two methods are drastically different. Dynamic variable splitting does not depart from the breadth-first search paradigm. It splits a difficult image computation into two halves, the union of the which yields the same set of states as a traditional image computation. Dynamic hint generation departs from breadth-first search and explores the subspaces of reachable states that satisfy the hints in order before an unconstrained reachability fixedpoint is attempted. Furthermore, our automated hint algorithms guarantee that the subspace explored using a hint cube is contained in the subspace explored using the subsequent hint cube due to the inclusion relations between the hints. This often renders the BDDs for these subspaces more compact in comparison to a hint computation using a set of hints with more complex expressions and little relation between them.

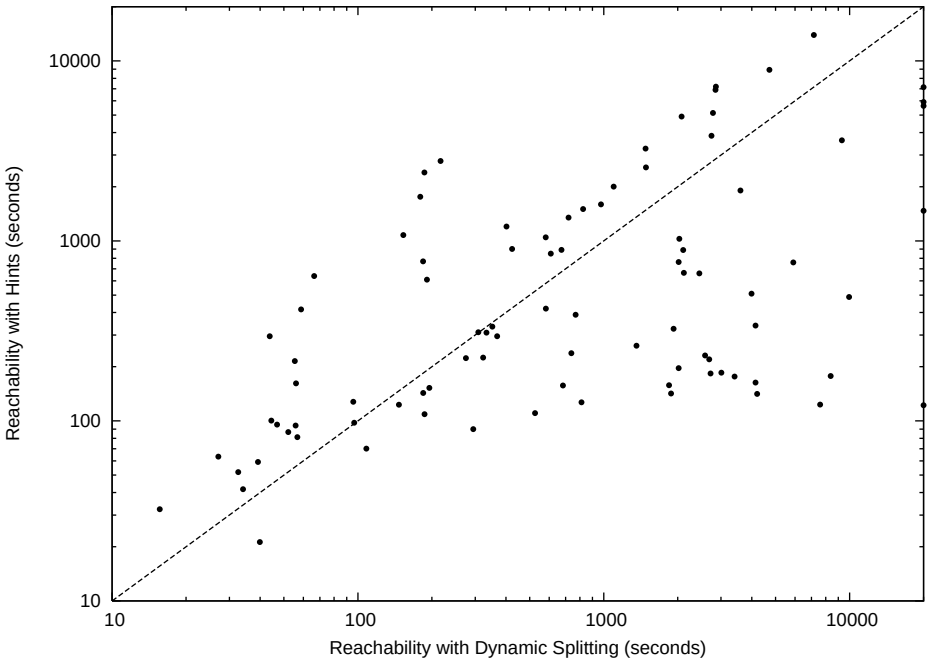

Fig. 4 Reachability Computation Runtime Using Dynamic Hints Generation vs. Dynamic Variable Splitting

The most challenging step in reachability computation under hints is often the final refinement step where the original transition relation is restored. The starting state set for this computation is the set of states reached using the prior hints. This state set itself could become very large and complex. However, in practice, it is often fairly compact due to the dynamic introduction of hints. There are nonetheless a few potential causes of difficulty for the final refinement step:

i) The initial selection of BDD literals to build the hint cube is highly dependent on BDD variable ordering; so in some cases, the choice of the BDD literals might not be optimal if evaluated at a different time of the computation.
ii) The subspace explored by the hints could be quite limited and in turn the final refinement step has to enumerate the vast majority of the reachable state space.

We have found that dynamic variable splitting should be applied to any difficult hint-constrained image step during the refinement before additional hints are computed to enhance the overall fixedpoint algorithm. Thereafter, continued hint generation, variable splitting, and dynamic variable reordering will be used to cope with the complexity of the final reachability computation.

We have experimentally compared the performance of the dynamic hint generation method with that of dynamic variable splitting. The result is shown in Table 4. Of the 92 testcases run, the dynamic hint method solved all the testcases while dynamic variable splitting left 5 testcases unsolved. For the remaining testcases, 38 of them have a better performance with dynamic variable splitting than with dynamic hint generation. The following are some of our observations:

– For difficult reachability testcases, dynamic hint generation has a better chance to tunnel through the complex image steps where BDDs otherwise explode.

- Dynamic hint generation provides robust performance over a variety of testcases, thus is a more robust method for hardware verification.
- Dynamic hints entail nontrivial overhead due to a larger amount of image steps needed to perform a fixedpoint across all hints. Therefore, for simpler testcases, it is often outperformed by dynamic variable splitting which only increases the number of image steps by a factor of two in its cofactored breadth-first search.
- Some circuit structures are more suitable for dynamic variable splitting than dynamic hint generation. In [7], it is noted that circuits with a more "full" dependency matrix are good candidates for dynamic variable splitting.
- Dynamic hint generation could sometimes generate overly restrictive hints which explore only a very limited subspace of the reachable states and leave the rest to be explored in the final refinement step. In such cases, allowing dynamic variable splitting during the refinement step will help decompose the resulting complex image computations.

This experiment shows that dynamic hint generation is a viable approach that complements other techniques such as dynamic variable splitting in overcoming difficult image computations during reachability analysis. These two techniques may be synergistically deployed during the same reachability computation; our current implementation allows splitting during the final refinement step of hint processing. However, as pointed out in [7], during the course of reachability analysis it is often more advantageous to have a flexible strategy to choose a best technique given the nature of the problem at hand. It thus might be interesting to explore more dynamic interactions between these two methods, such as allowing splitting within a hint computation, or deploying hint computation if cofactoring does not adequately simplify subimage computations. Future research will focus on improved heuristics to better complementarily leverage these techniques. It is also expected that the dynamic hint algorithm should work in backward reachability computation as well. It might be interesting to study if higher priority should be given to the variables in the support of the property when generating the hints.

6 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 various classes of problems. A robust industrial toolset thus practically requires the availability of a highly-tuned BDD-based reachability engine for overall robustness. 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. These techniques have played a vital role in revitalizing reachability analysis as a core industrial-strength proof technique in our multi-algorithm verification toolsuite. We also compared and studied the performance of the dynamic hint generation with the alternative yet complementary method of dynamic variable splitting. This further demonstrates that hint generation is robust and can solve difficult cases more reliably.

References

1. J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations," in *International Conference on Very Large Scale Integration*, pp. 49–58, August 1991.

2. I.-H. Moon, G. D. Hachtel, and F. Somenzi, "Border-block triangular form and conjunction schedule in image computation," in *International Conference on Formal Methods in Computer-Aided Design*, pp. 73–90, November 2000.
3. K. McMillan, "Interpolation and SAT-based model checking," in *International Conference on Computer-Aided Verification*, pp. 1–13, July 2003.
4. A. Bradley, "SAT-based model checking without unrolling," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 70–87, January 2011.
5. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 193–207, 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 *International Conference on Computer-Aided Design*, pp. 120–126, November 2000.
7. I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi, "To split or to conjoin: the question in image computation," in *Design Automation Conference*, pp. 23–28, June 2000.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer-Aided Verification*, pp. 154–169, July 2000.
9. H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification," in *Design, Automation and Test in Europe*, pp. 1674–1679, April 2009.
10. P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *International Conference on Computer-Aided Design*, pp. 1076–1082, November 2005.
11. A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *International Conference on Computer-Aided Verification*, pp. 104–117, July 2001.
12. H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *International Conference on Formal Methods in Computer-Aided Design*, pp. 159–173, November 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," *IEEE Transactions on Computer-Aided Design*, vol. 19, pp. 44–55, January 2000.
16. K. Ravi and F. Somenzi, "Hints to accelerate symbolic traversal," in *Correct Hardware Design and Verification Methods*, pp. 250–266, October 1999.
17. D. Ward and F. Somenzi, "Automatic generation of hints for symbolic traversal," in *Correct Hardware Design and Verification Methods*, pp. 207–221, October 2005.
18. D. Ward and F. Somenzi, "Decomposing image computation for symbolic reachability analysis using control flow information," in *International Conference on Computer-Aided Design*, pp. 779 – 785, November 2006.
19. K. Ravi and F. Somenzi, "High-density reachability analysis," in *International Conference on Computer-Aided Design*, pp. 154 – 158, November 1995.
20. Hardware Model Checking Competition 2011. http://fmv.jku.at/hwmcc11.
21. G. Janssen, "Design of a pointerless BDD package," in *International Workshop on Logic Synthesis*, June 2001.
22. H. Fujii, G. Ootomo, and C. Hori, "Interleaving based variable ordering methods for ordered binary decision diagrams," in *International Conference on Computer-Aided Design*, pp. 38–41, November 1993.
23. H. Jin, A. Kuehlmann, and F. Somenzi, "Fine-grain conjunction scheduling for symbolic reachability analysis," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 312–326, April 2002.
24. N. Eén and N. Sörennson, "Temporal induction by incremental SAT solving," in *Workshop on Bounded Model Checking*, December 2003.