Input Elimination Transformations for Scalable Verification and Trace Reconstruction

Raj Kumar Gajavelly, Jason Baumgartner, Alexander Ivrii, Robert L. Kanzelman, Shiladitya Ghosh
IBM Corporation

***Abstract*—We present two novel sound and complete netlist transformations, which substantially improve verification scalability while enabling very efficient trace reconstruction. First, we present a 2QBF variant of input reparameterization, capable of eliminating inputs without introducing new logic and without complete *range* computation. While weaker in reduction potential, it yields up to 4 orders of magnitude speedup to trace reconstruction when used as a fast-and-lossy preprocess to traditional reparameterization. Second, we present a novel scalable approach to leverage *sequential unateness* to merge selective inputs, in cases greatly reducing netlist size and verification complexity. Extensive benchmarking demonstrates the utility of these techniques. *Connectivity verification* particularly benefits from these reductions, up to 99.8%.**

I. Introduction

The use of model checking algorithms has proliferated throughout the semiconductor industry in recent years. This is due to several factors, including their ease of use, ever-growing capacity, and the advent of niche applications such as connectivity verification and sequential equivalence checking which require virtually no manual effort. Capacity improvements have largely been due to **(1)** improved core algorithms, such as SAT solvers; **(2)** a growing diversity of core solving algorithms, each able to exponentially outperform the other on different problems, offering tremendous portfolio benefit; **(3)** improved synergistic complexity-reduction techniques, able to substantially boost the scalability of later solver engines; **(4)** end-to-end software scalability boosts, such as techniques to accelerate the reconstruction of counterexample traces produced across a collection of reduction and solver engines.

Despite the growing capacity of model checkers, the size and complexity of systems requiring verification also continues to grow. Formal verification generally entails exponentially-growing complexity with respect to the size of the design under verification. Continued advances to the end-to-end scalability of model checkers are thus of substantial importance.

In this paper, we introduce two novel sound and complete netlist transformations to boost verification scalability. The first is a scalable 2QBF variant of input reparameterization [4], [10], oriented toward merging input variables to constants while preserving the *range* of logic cuts (Section V). The second is a novel approach for leveraging *sequential unateness* to merge selective input variables to constants while preserving representative counterexamples (Section VI). Both are scalable enough to include in a standard state-of-the-art logic optimization flow. While traditional reparameterization has been demonstrated as highly-beneficial to verification scalability [4], [10], unscalable trace reconstruction is an occasionally-severe byproduct risking end-to-end resource degradation and/or need for customized orchestration. Our techniques offer trivially-fast trace reconstruction; when used as a preprocess to traditional reparameterization, they offer up to 4 orders of magnitude speedup. *Connectivity verification* [11] particularly benefits from these techniques, as discussed in Section VII.

II. Related Work

The often-essential role of logic transformations to boost hardware verification scalability is well-established (e.g., [13], [1]), with competitive hardware model checkers leveraging a variety of synergistic optimizations to prevent common hardware artifacts from severely degrading verification scalability. Many useful transformations have been developed over the decades, ranging from area-reduction techniques from logic synthesis to abstractions and temporal transformations that apply only in a verification context. For example, *localization* is a highly-effective abstraction-refinement technique, consisting of replacing various gates in a netlist with *cutpoints* or unconstrained primary inputs (hereafter referred to as simply "inputs"). Because these cutpoints can simulate the behavior of the logic they replace and more, this abstraction is *sound* but *incomplete* in that spurious counterexamples may occur. Through refinement, spurious counterexamples and cutpoints may be eliminated to render a *complete* overall technique [16].

Our first contribution is a variant of the known input-elimination technique of *reparameterization*, tailored to merge selective inputs to constants while preserving overall netlist behavior. Borrowed from variable elimination techniques for symbolic simulation, the initial use of reparameterization for sequential netlist reduction used BDDs to compute the *range* of a logic cone adjacent to inputs selected using a min-cut algorithm, then synthesized replacement logic producing the same range using only a single new parametric input per cut node [4]. More-recent work proposes a faster yet lossier variant using truth-table analysis of single-output, eight-input dominator subcircuits [10].

Our experience confirms that both techniques are highly-beneficial to verification scalability, and synergistic: optimal reductions may be achieved with minimal resources by iterating the two. Regardless, these approaches have several drawbacks motivating our work: **(1)** trace reconstruction resources can be substantial, requiring BMC-like analysis to translate a trace over the reparameterized netlist to one consistent with the original netlist. Especially for multi-property testbenches

requiring multiple traces, *trace reconstruction resources may dominate end-to-end verification resource, hurting more than helping scalability*. **(2)** The size of the replacement logic is guaranteed to have fewer inputs, but may at times increase combinational logic; this may hurt vs. help different verification flows. **(3)** Replacement logic entails logic perturbation, with drawbacks such as degrading the ability to correlate logic regions in equivalence checking, and obfuscating user-guided case-splitting. In contrast, our technique has trivially-fast trace reconstruction; never increases logic size; and preserves every gate in the original netlist modulo merging. Though since weaker in reduction potential, we propose our technique as a fast-and-lossy preprocess to traditional reparameterization.

Our second contribution is a novel transformation leveraging *sequentially-unate* logic characteristics to merge selective inputs to constants while preserving representative counterexamples. A *unate* input is one which monotonically affects a function; i.e. the positive cofactor of the function w.r.t. that input implies the negative cofactor, or vice-versa. Various CAD applications may leverage unateness for improved scalability, such as functional-dependency analysis with applications in reverse-engineering [15], and Boolean matching with applications in incremental synthesis and certain types of equivalence checking [12]. Ours is the first to our knowledge to leverage unateness to accelerate *unbounded model checking*, using highly-scalable structural analysis to identify inputs affecting properties in only a singly polarity, even if appearing in different polarities across different next-state functions. Trace reconstruction is also highly-efficient for this technique.

Somewhat-related is *word-level bitwidth reduction*, applicable to netlists partitionable into a bit-level network (retained intact) vs. a word-level network (potentially reducible) [6]. The only supported feedback from the latter to the former is via word-level (in)equality comparisons between data vectors or uniform all-0 or all-1 constant vectors, and from the former to the latter is via multiplexor selectors. An input vector (possibly routed through multiplexors and/or latches) in the word-level network only influencing (in)equality comparisons to a single constant value is a special case of sequentially-unate logic; depending on netlist topology, bitwidth reduction may be able to shrink that input vector to fewer bits, but never to 0 bits. In contrast, sequentially-unate input reduction can reduce such input vectors to 0 bits, similar to *positive equality* [8], [19] reductions applicable to a combinational formulae derived from a netlist. As discussed in [6], extending formula-reduction techniques to sequential netlists and general unbounded model checking is far from trivial; in a sense, sequentially-unate input reduction could be viewed as a positive-equality extension of bitwidth reduction. Though generally, bitwidth reduction is most-useful to yield reductions beyond those possible with our techniques, and our techniques apply to bit-level netlists with no partitionability requirements; these are all synergistic.

Our transforms often significantly improve verification scalability. Their benefits combined with localization [16] is particularly strong, due to **(1)** cutpoints creating additional input- and unate-reduction opportunities, **(2)** this greater netlist reduction accelerating subsequent verification, and **(3)** enabling faster abstraction-refinement iterations due to greater efficiency of trace reconstruction vs. traditional reparameterization [4], [10], which benefits even provable properties.

III. PRELIMINARIES

We focus on netlists represented as And-Inverter Graphs (AIGs) [5]. AIG gates G comprise a constant-ZERO gate; primary inputs; 2-input AND gates; bit-level unclocked *registers* which reference two other gates, their *initial value* defining their time-0 value and their *next-state function* defining their time-$i+1$ value; and implicit inversions as edge attributes. A netlist may be reasoned about as a *Finite State Machine* $(S_0, I, \delta, \lambda, S, O)$, where I is the *input alphabet*, S is the set of *states*, $S_0 \subseteq S$ is the set of *initial states* consistent with initial values, $\delta : S \times I \rightarrow S$ is the *transition function* consistent with next-state functions, $O \subseteq G$ is the *output alphabet*, and $\lambda : S \times I \rightarrow O$ is the *output function*.

A $trace$ is a sequence of Boolean valuations to gates, starting with an initial state in S_0 and with successive timesteps consistent with δ. A *property* is an output representing a verification objective. A *safety property* p is an LTL objective $G\neg p$ of computing a counterexample trace showing an assertion of p, or of proving that no such trace exists. A *liveness property* l is an LTL objective $GF\neg l$ of computing a counterexample trace showing reachability of a repeating *lasso loop* state sequence wherein l remains asserted forever, or proving that no such trace exists. A *constraint* is an output c imposing a restriction that any trace must show c evaluate to 1 at every timestep. A *fairness constraint* f qualifies only livelocks, requiring any livelock counterexample to witness at least one assertion of f within the repeating lasso loop. Our techniques are applicable to both safety and liveness properties.

IV. INPUT REPARAMETERIZATION

Traditional reparameterization is performed as per Algorithm 1: first select a candidate structural cut of the netlist within the combinational fanout of existing inputs, then create replacement logic producing the same *range* as the cut gates using a new parametric input per cut gate. By choosing cuts whose set of dominated inputs is larger in cardinality than the cut width, a reduction in input count is guaranteed. Generally, the candidate cut may include registers or other non-dominated gates in its support, hence the range is generally a function of original gates which persist after replacing the candidate cut.

Algorithm 1 Input Reparameterization for Sequential Netlists

1: Select a netlist *cut* in the combinational fanout of inputs.
2: Compute the range of the *cut* as a function of registers and non-dominated gates in its combinational fanin.
3: Synthesize a *replacement circuit* producing identical range values over new parametric inputs.
4: Replace the original logic cut with the *replacement circuit*.

[4] uses a min-cut algorithm to identify the candidate cut, and BDDs to compute the range. [10] uses dominator

analysis [14] to identify small single-output cuts dominating two or more inputs (though limited to eight inputs and non-dominated gates), and exhaustive simulation to compute the range. Both techniques may be iterated, and applied synergistically for greater reductions. While primarily focused upon input reductions, they may also yield register and combinational logic reductions, contributing to verification speedups. Though generally, they introduce *new* combinational logic, which may outweigh the replaced logic.

If a counterexample is generated on the reparameterized netlist, *trace reconstruction (TR)* is necessary to map that trace to one consistent with the original netlist. This TR will first simulate the trace on the abstract netlist to populate Boolean values to the cut gates and to any registers or non-dominated gates upon which the synthesized range depends, then perform a bounded-model-checking (BMC)-like SAT query to compute a sequence of valuations to the original logic yielding identical valuations to the cut gates. As noted in [4], this process is generally less expensive than typical BMC, since the prior simulation decomposes it to a set of independent per-timestep SAT queries. Nonetheless, this process can be *extremely* computationally-expensive, especially for netlists for which SAT analysis is expensive (e.g., exclusive-or rich logic) or when the trace to be reconstructed is *much longer* than practical using BMC alone, as often happens with a semi-formal bug-hunting engine [17]. In practice, TR is often *more expensive* than reparameterization itself, especially for netlists with multiple failing properties. TR can dominate end-to-end verification resource, in cases rendering reparameterization more harmful than beneficial to a verification portfolio. *This end-to-end impact is a primary motivation for our techniques: to retain and improve upon reparameterization reductions, while containing TR resource.*

V. Reparameterization without Logic Insertion

In this section we present a novel netlist reduction for merging selective inputs to constants, after which constant propagation and other logic optimizations may further reduce netlist size. This technique identifies the set of inputs *dominated* by a candidate cut, then evaluates if they can be merged to 0/1 by comparing the range values producible at the cut with and without candidate reductions. Intuitively, we compare two ranges: first leaving the inputs intact, second after merging the inputs to 0 or 1. If equal, the corresponding inputs can be merged without observably altering cut behavior; soundness follows from [4]. This approach can be viewed as a generalization of *observability don't care*-based reductions [18], performing reductions not only if the behavior of the cut gates is unchanged, but if the *range* of the cut is preserved.

This computation can be expressed as 2QBF: *Quantified Boolean Formula* with two levels of quantification. Let $F(x, Y, Z)$ denote the logic cone defining the cut, where x is a given dominated input, Y is the remaining dominated inputs, and Z are non-dominated gates (which may be inputs or other gates). Then x can be merged to 0 iff

$$\forall Z, Y \ \exists Y' \left(F(1, Y, Z) = F(0, Y', Z) \right),$$

and x can be merged to 1 iff

$$\forall Z, Y \ \exists Y' \left(F(0, Y, Z) = F(1, Y', Z) \right).$$

I.e., if for all values of non-dominated gates Z, each value producible with $x = 1$ is producible with $x = 0$ given other valuations to Y, then x can be merged to 0, and vice versa.

A. Examples

(1) Consider a netlist with inputs x, y and *exclusive-or gate* $c = x \oplus y$. If neither x nor y have fanouts besides c, both are dominated inputs. x can be merged to 0 or to 1: varying y despite this merge produces all range values for c. (Alternatively, y can be merged to 0 or to 1 while retaining x as an input.) After merging x to 0, constant propagation will merge c with y. Trace reconstruction is trivial: merely propagate 0 for eliminated input x for every timestep.

(2) Assume in the above example that y has additional fanouts, i.e. is not dominated by c. Then our reduction does not apply. Traditional reparameterization [4], [10] may be able to eliminate x by introducing a new input, even though reparameterization is typically skipped for cuts where input count cannot be reduced. However, traditional reparameterization requires expensive SAT-based trace reconstruction, as the reconstructed value of x will depend on y and c.

(3) Suppose that $c = x \vee y \vee z_1 \vee \cdots \vee z_n$ is an *OR gate*, where x and y are dominated inputs and $z_1, \ldots, z_n$ are non-dominated inputs. There are two cases of interest: **(a)** if $z_1, \ldots, z_n$ all evaluate to 0, then both $c = 0$ and $c = 1$ are producible depending on values of x and y. **(b)** If at least one of $z_1, \ldots, z_n$ evaluate to 1, then only $c = 1$ is producible. In this example, either x or y can be merged to constant-0.

B. Input-Elimination Algorithm

Algorithm 2 receives a cut gate set C, a set of dominated inputs separated into a single input x and the remaining inputs Y, and a set of non-dominated gates Z. It then checks whether x can be merged to 0 while preserving the range of C. (Merging x to 1 is handled similarly.) The approach is SAT-based, and reminiscent of circuit-based 2QBF-approaches [3]. However, it contains an important optimization to enable *early termination without complete range computation, whether deducing that a merge is possible or not*. Instead, it only concentrates on different valuations to the cut gates with x cofactored to 0 vs 1.

On line 1, F' and F'' are two copies of the logic between C and x, Y, Z, with gates that do not depend on x common across the two copies, and with x set to 1 in F' and set to 0 in F'' (this focuses on the cofactored difference in behavior, and simplifies the SAT instance). We denote the outputs of F' and F'' by C' and C'' respectively. Thus the SAT-query on line 4 checks whether the value of at least one gate in C' can be different from the value of the corresponding gate in C'', and thus essentially checks if it possible to change the valuation of C by only changing the value of x, and not of Y and Z. If this is possible, line 5 extracts the assignment r that can be produced with $x = 1$, where r considers valuations

Algorithm 2 TryToMergeInputTo0(cut gates C, dominated inputs x, Y, non-dominated gates Z)

1: $F'(1, Y, Z) \leftarrow$ copy of logic driving C with x set to 1,
$F''(0, Y, Z) \leftarrow$ copy of logic driving C with x set to 0,
(C' and C'' denote outputs of F' and F'' respectively)
2: $enumerated \leftarrow \emptyset$
3: **while** (resource limits not reached) **do**
4: **if** $\texttt{IsSAT}((F' \neq F'') \wedge \neg enumerated) \equiv \texttt{SAT}$ **then**
5: // found a new satisfying assignment α
$r \leftarrow \alpha|_{C' \cup Z}$ // new range value produced with $x \equiv 1$
6: **if** $\texttt{IsSAT}(F'' \wedge r) \equiv \texttt{UNSAT}$ **then**
7: // r cannot be produced with $x \equiv 0$
return CANNOT MERGE
8: **else**
9: // r can be produced with $x \equiv 0$
$enumerated \leftarrow enumerated \cup r$
10: **end if**
11: **else**
12: // no remaining assignments
merge x to 0, **return** MERGED
13: **end if**
14: **end while**
15: **return** LIMITS-EXHAUSTED

to both C' and Z. Another SAT-query on line 6 checks if the same assignment on $C \cup Z$ can be also produced with $x = 0$ while allowing to vary the values of other dominated inputs Y. If not, then x cannot be merged to 0: doing so would change the values producible on the cut for at least one choice of values of Z, and the algorithm returns (line 7). Otherwise, this assignment is marked as $enumerated$ and blocked from further consideration (line 9), and the first SAT-query is repeated. If at any point this query is unsatisfiable (line 11), then changing the value of x no longer affects the set of possible assignments to C, and hence every assignment to $C \cup Z$ that can be obtained with $x = 1$ can be also obtained with $x = 0$. In this case, x is merged to constant-0 and the algorithm returns (line 12). For efficiency, two incremental SAT-solvers are used. The first is used for queries on line 4, and the new range values r producible with $x = 1$ are blocked by adding a corresponding clause to the solver. The second is used for queries on line 6, and the restrictions r on variables in $C \cup Z$ are passed to the solver via assumptions.

Example 1: Consider the example **(3)** from Section V-A. The first SAT-query is satisfiable, with the (only) satisfying assignment $\alpha = \{y = 0, z_1 = \cdots = z_n = 0, c = 1, c' = 0\}$. In other words, the value $c = 1$ is producible with $x = 1$ whenever $y = 0, z_1 = \cdots = z_n = 0$, but would change to 0 if x is changed to 0. The second SAT-query checks if $c = 1$ is also producible with $x = 0$ and with $z_1 = \cdots = z_n = 0$, and it is as y can be assigned to 1. The assignment $c = 1, z_1 = \cdots = z_n = 0$ is added to $enumerated$ and the first SAT-query is repeated. This time the query is unsatisfiable, and hence x can be merged to 0. Note that overall

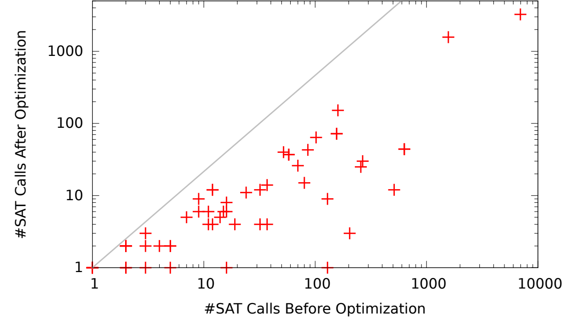


Fig. 1: Number of SAT Calls for Successful Merges

there are 2^n assignments to $\{z_1, \ldots, z_n\}$ and avoiding the full range computation decreased the number of SAT-queries from exponential to linear (or constant).

We demonstrate the impact of the incomplete range computation optimization in Figure 1, for various HWMCC benchmarks [2] where an input is successfully merged. This figure illustrates the total number of SAT calls with partial (after optimization) vs. complete (before optimization) range computation. The average improvement is 2.2×, up to 129.0×. A similar experiment for unsuccessful merge attempts shows an even larger average improvement of 9.4×. Intuitively, our technique can leverage this optimization because it does not attempt to synthesize new range-equivalent logic, only to reduce existing logic while preserving range values. Compared to min-cut-based traditional reparameterization [4], this shortcut often achieves (different) reductions with lesser resource: our approach averages 16.5× faster than traditional reparameterization even with modest BDD-size limits, yielding an average of 13.5% as much input reduction though occasionally yielding identical reductions. Dominator-based reparameterization [10], in contrast, is substantially faster: an average of 13.4×, though lacking multi-output-cut reduction capability.

C. Cut Selection and Resource-Bounding

Selecting candidate cuts for Algorithm 2 is done similarly to [4], [10]: using min-cut analysis [4], or single-output dominator analysis [10], [14] of cuts dominating at least two inputs. To optimize reductions vs. runtime, we first analyze purely-combinational min-cuts, then dominator-based cuts, then (optionally) min-cuts including registers in their fanin. The motivation: cuts with a larger number of dominated inputs vs. cut width, and fewer nondominated gates, are more-likely reducible; skipping irreducible cuts vastly improves runtime.

As with traditional reparameterization, our technique is beneficial to apply iteratively to reduce increasingly-deep logic cones: shallower reductions simplify the analysis of deeper cones. For best runtime with minimal forgone reduction, unlike [10] we do not limit the number of dominated inputs, but instead the number of iterations of the **while**-loop (line 3) defaulting in our experiments to 1024, with SAT propagations per iteration limited to 100000. If a cut exceeds these resources, deeper cuts are also likely to exceed; our preferred

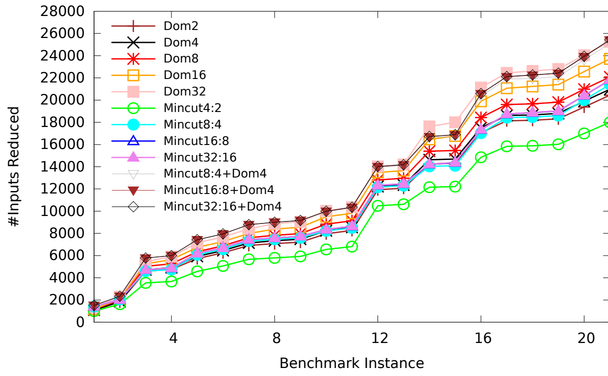


Fig. 2: Algorithm 2 Input Reduction

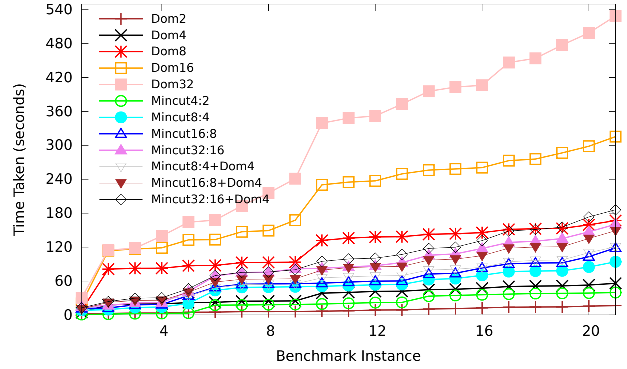


Fig. 3: Algorithm 2 Runtime

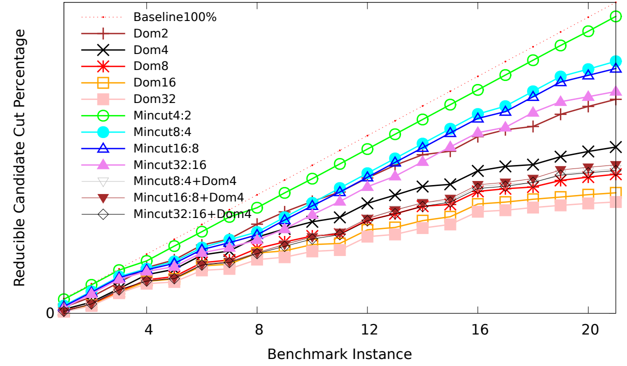


Fig. 4: Algorithm 2 Success Rate

flow thus marks the fanout of resource-exceeded cuts, and avoids analyzing them later. The number of non-dominated gates is practically useful to bound, with a heavy tradeoff between speed and optimality of reductions: larger resources obviously yield greater reductions, though at a significant runtime cost and diminishing success rate.

After achieving the majority of reductions with a fast-and-lossy configuration, our experiments defer more-aggressive input reductions to traditional reparameterization.[1] With this configuration, Algorithm 2 is fast enough to include in a standard logic optimization flow, which benefits everything downstream. Practically this offers the best overall verification speedup using generic orchestration, with Algorithm 2 achieving some but not all of its input reduction potential, leveraging the additional reduction capability of traditional reparameterization while containing its trace reconstruction overhead. *By iterating our approach with [10] and [4], we achieve* 22.2% *average greater input reduction than traditional reparameterization alone in* 43.6% *lesser runtime, while enabling the trace reconstruction speedups of Section V-E.*

D. Experiments

The impact of candidate cut selection options on reduction is shown in Figure 2, with runtime in Figure 3 and success rate (percentage of candidate cuts yielding at least one eliminated input) in Figure 4. These benchmarks are a subset of those of Section V-E, chosen as large highly-reducible netlists where option variation substantially affects results. These are cactus plots, where the ith column sums all prior columns.

The *Dom#* lines reflect single-output dominator cuts, with unlimited dominated inputs but nondominated gates bounded by the respective number. Note that runtime increases sharply with this parameter; limiting to 2, runtime is very fast (0.79 seconds average) achieving 80.9% of the reductions of setting 32 (23.7 seconds average).

The *Mincut#:#* lines reflect combinational min-cuts, where the first number limits cut width and the second bounds non-dominated gates. Even large settings are substantially faster than dominator-based (*Mincut32:16* averages 7.6 seconds), yielding much but not all reduction achievable with dominator-based. Faster runtime is largely due to greater success rate. Weaker reduction potential is primarily due to dominator-based allowing a small number of registers in candidate cuts, whereas in these experiments min-cuts do not.

The *Mincut#:#+Dom#* lines run the corresponding dominator-based reduction *after* (vs. *instead of*) the corresponding min-cut reduction. Both are able to reduce some common and some unique inputs, yielding synergistic value. Since min-cut is substantially faster than dominator-based, our preferred flow used in later experiments is *Mincut32:16+Dom4*.

E. Synergy with Traditional Reparameterization

A primary motivation for the proposed technique is its trivially-fast trace reconstruction: merely populate inputs by the constant values they were merged to, with no dedicated SAT or simulation analysis. This process requires only a fraction of a second even on the largest and heaviest-reduced netlists. In cases, Algorithm 2 can reduce as many inputs as traditional reparameterization. Though in general, traditional reparameterization can perform additional reductions even after the proposed technique is exhausted or reaches cost-effective resource bounds. A synergistic application of both techniques yields optimal results in practice, with the proposed technique applied *before* traditional reparameterization to limit its scope and thereby offset its sometimes-expensive trace reconstruction. The benefit of this strategy is illustrated in

[1]"Traditional reparameterization" in our experiments consists of "strong" dominator-based reparameterization [10], followed by min-cut BDD-based reparameterization [4] with BDD limit 2^{15}. The former is very fast; the latter enables multi-output reductions impossible with the former. Both are applied to increasingly-deep logic cones, truncating at cuts exceeding resource limits.

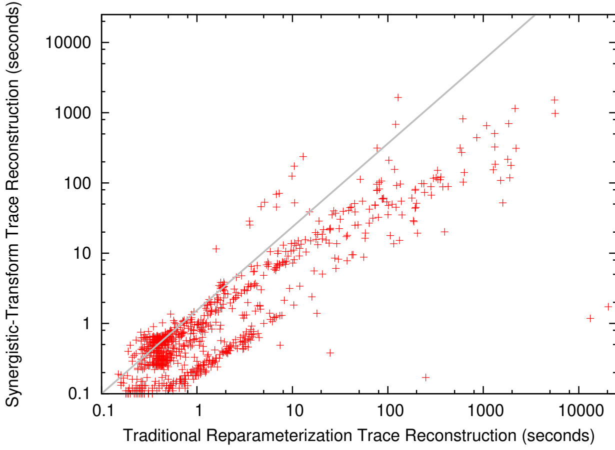


Fig. 5: Synergistic Input Elimination vs. Traditional Reparameterization Trace Reconstruction

Figure 5, offering comparable netlist reductions to accelerate semi-formal bug-hunting while yielding the trace reconstruction speedup depicted in this plot: a 29.6× average per-benchmark speedup, with a maximum speedup of 11808.3×.

VI. Sequentially-Unate Input Reduction

In this section we present a different yet synergistic netlist reduction, *sequentially-unate input reduction* (SUR). From literature, a combinational Boolean function $f(x_i, X)$ is called *positive unate* with respect to input x_i if $\forall X \ f(1, X) \geq f(0, X)$, and *negative unate* if $\forall X \ f(0, X) \geq f(1, X)$. If neither positive nor negative unate, f is *binate* with respect to x_i. E.g., given inputs x and y, *or gate* $x \vee y$ and *and gate* $x \wedge y$ are positive unate with respect to both x and y, whereas *exclusive-or gate* $x \oplus y$ is *binate* with respect to both.

Note that counterexamples require demonstrating assertions (not deassertions) of properties, constraints, and fairness constraints. For combinational netlists, unateness lends itself to a fairly-obvious reduction:

Theorem 1: When verifying a combinational netlist, if an input is found positive (negative) unate with respect to every property and constraint and fairness gate, it is sound and complete to merge that input to constant-1 (0).

Proof: Any input not appearing in the cone-of-influence of any property, constraint, or fairness gate is both positive and negative unate, and can obviously be merged to any value as it cannot influence counterexample validity.

For other inputs, any original-netlist trace evaluating an input to its merged-to constant value is producible on the reduced netlist, proving completeness. To prove soundness: consider an original trace c_1 evaluating an input i_1 to the opposite of its merged-to constant. By the definition of unateness, modifying i_1 in c_1 to its merged-to value will only evaluate the property, constraint, and fairness gates to 1 monotonically more often, and thus render a valid counterexample. ■

Unate input reduction in sequential netlists is not as straightforward. For example, consider a netlist with input x occurring only in (and *as*) the next-state function of register r, and property logic that counts two positive-edge transitions of r as a counterexample. While x appears positive-unately in every combinational function in the netlist, merging x to constant is unsound due to its influence on sequential fanout logic: x must toggle in any counterexample.

To leverage unateness in a sequential netlist, it is essential to track the number of inversions modulo-two along every structural path between each property, constraint, and fairness gate and each input, as illustrated in Algorithm 3. In line 5, the gate-inputs of a register refer to its initial-value and next-state function. We refer to any input unmarked as "negative" as *sequentially-positive unate*, and any input unmarked as "positive" as *sequentially-negative unate*. This algorithm has linear runtime, marking each gate at most once per polarity.[2]

Algorithm 3 Sequentially-Unate Input Reduction

1: markPolarityAIG(gate g, polarity p) {
2: **if** isInverted(g) **then** $p \leftarrow \neg p$; $g \leftarrow$ uninvert(g)
3: **if** getPolarity(g) $\supseteq p$ **then** return // already marked
4: addPolarity(g, p)
5: **for each** gate-input i of g
6: markPolarityAIG(i, p)
7: }
8:
9: sequentiallyUnateInputReduction(netlist N) {
10: **for each** property, constraint, fairness gate g
11: markPolarityAIG(g, positive)
12: **for each** input x
13: **if** getPolarity(x) $\neq$ {positive, negative} **then**
14: // x is sequentially-unate
15: merge x to (getPolarity(x) $\equiv$ {positive}) constant
16: }

Theorem 2: Algorithm sequentiallyUnateInputReduction of Figure 3 is a sound and complete netlist transformation.

Proof: Completeness follows for any transform that only performs input merging: every reduced-netlist trace is producible on the original netlist. To prove soundness: because every structural path from a sequentially-unate input i_1 to every property, constraint, and fairness gate passes through an identical-modulo-two number of inversions, changing the valuation of i_1 to its merged-to value in original trace c_1 can only cause these gates to evaluate to 1 monotonically more often in the modified trace c_1'. Such modified traces are producible on the reduced netlist, hence all original counterexamples are preserved modulo this modification. ■

Intuitively, Theorem 2 holds by the definition of combinational unateness because every bounded-model-checking unfolding timestep would be compatibly positive or negative unate for every unfolded instance of a sequentially-unate input. However, note that a sequentially-unate input is not necessarily compatibly-unate in every next-state function in

[2]While semantic extensions to this purely-structural algorithm are possible, their computational expense often risks degrading verification runtime. We defer this as a future research direction, e.g. as an extension of [9].

TABLE I: SUR Verification Speedup

Benchmark	Inputs	ANDs	Unate Inputs	Unate ANDs	Solving Time (sec)	Reduced Solving Time (sec)
$oski2ub5i$	13231	174682	16	24	12250.4	2908.4
$6s429$	13335	419135	18	38	9994.1	2930.4
$oski4ui$	11836	122438	24	48	11153.8	8950.5
$oski5ui$	3175	24689	6	12	7627.5	6917.7
$6s301$	1048	101655	8	22	1046.1	647.9
$6s115$	1966	121473	684	44545	831.5	673.0
$bob1u05cu$	100	17647	61	183	248.7	115.9
$mentorbm1$	164	24996	93	711	346.1	240.2
$bob05$	100	17647	61	183	201.4	98.9
$6s143$	425	13928	3	3	1061.9	970.5
$6s310r$	86	3014	8	15	155.6	121.1
$oski1rub03$	13074	109711	40	48	90.0	56.4
$oski1rub07$	13071	109665	40	64	95.0	64.5
$6s8$	86	3016	8	15	212.9	184.6
$nusmvtcastp2$	146	2744	2	2	35.8	18.8
$nusmv.tcast2.B$	146	2744	2	2	34.9	19.6

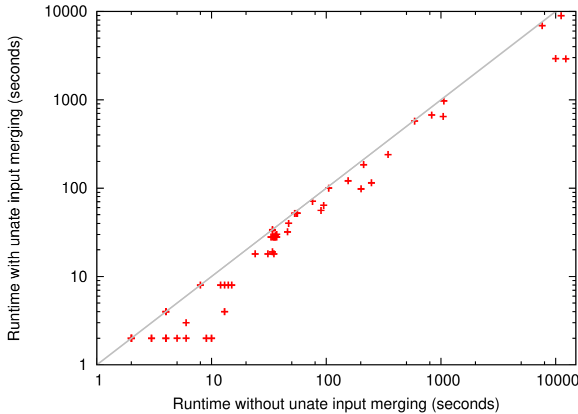


Fig. 6: Verification Runtime Impact of SUR

which it appears. For example, given an input i which is the next-state function of register r_1, and *NAND gate* $\neg(i \wedge r_1)$ which is the next-state function of r_2 which is labeled as a property: i appears positive-unately in the next-state function of r_1, and negative-unately in the next-state function of r_2, and is sequentially-negative unate with respect to the property. i may be merged to 0 while preserving verification results, regardless of choice of subsequent verification algorithms.

Beyond its reduction benefits, SUR can improve semi-formal bug-hunting [17] by increasing the probability that randomly-generated input valuations yield counterexamples.

A. Experimental Results

Table I and Figure 6 present HWMCC [2] benchmarks where SUR (Algorithm 3) substantially accelerates verification. Columns 2 - 3 illustrate original netlist size. Columns 4 - 5 present the number of inputs and AND gates eliminated by SUR. Columns 6 - 7 show end-to-end verification runtime with vs. without SUR using IC3 [7]: up to 4.2× faster, with an average speedup of 35.6% per benchmark. The percentage of unate inputs is at times substantial: 93 of 164 inputs on mentorbm1, and 56.0% average on mentorbm* benchmarks.

VII. Connectivity Verification

In addition to their primary functionality, modern semiconductor designs also include a variety of *pervasive logic* intertwined with the functional logic to enable practically-usable chips. For example, *trace bus logic* enables full-clock-speed post-silicon observability and performance monitoring, comprising internal routing and monitoring logic allowing the value of various internal signals to be configurably sampled onto an internal *trace bus* and ultimately propagated off-chip.

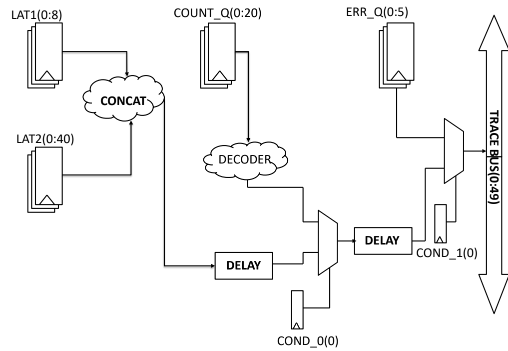


Fig. 7: Trace Bus Testbench Example

Verification of such logic is referred to as *trace bus verification* or *connectivity verification*, and sometimes mandates chip-level vs. lower block-level verification to capture its end-to-end behavior. A methodology is presented in [11] for automatic testbench creation from the *templates* used as post-silicon recipes for observing various internal sample points. In such templates, a vector of internal signal values are sampled with specified polarities and bit-positions and propagated onto a trace bus under a specified delay. The testbench effectively creates a reference trace bus against which the implementation is compared, on a per-sampled-vector basis.

An example of trace bus testbench logic is depicted in Figure 7, where vectors LAT1() and LAT2() are sampled and concatenated. Alternate trace bus sample points irrelevant to this specific property (likely checked by different properties) include ERR_Q, and a decoding of vector COUNT_Q. If the implementation logic can be isolated for standalone verification, bitwidth reduction [6] may greatly boost its scalability. In practice, such logic must be verified as instrumented into the chip, since most hardware bugs involve instrumentation details such as clocking/latching/power-saving problems, subtle problems with hierarchy-spanning selector logic and/or VHDL2008 “external signal” references, etc. Unfortunately, design logic irrelevant to trace-buses often renders bitvector reduction powerless (fragmenting all vectors to bit-level), at least until localization and heavy reductions eliminate that unnecessary logic. By that point, bitvector reductions can often identify reducible partitions, though most achievable reductions have already been exhausted by our techniques.

Because the amount of logic in the cone-of-influence of a specific property may be very large, a collection of synergistic algorithms including localization is often necessary to render tractable problems for core solvers such as IC3. Localization can often place cutpoints at the sampled signals relevant to a given property without affecting its provability, and also place cutpoints at side multiplexor inputs irrelevant to a given property. Coupled with synergistic logic rewriting optimizations and reparameterization, many of these cutpoints become sequentially-unate inputs reducible by Algorithm 3. These

TABLE II: Traditional Reparam. vs. Algo. 2 Reductions

Bench-mark	Inputs	ANDs	Registers	Traditional Reparam. Inputs / Time (sec)	Algo. 2 Inputs / Time (sec)
DBV1	22256	1138339	112161	6154 / 62.3	1546 / 19.1
DBV2	21435	4429287	456243	1799 / 2.2	400 / 1.5
DBV3	24111	1209308	115676	7406 / 63.2	1580 / 18.1
DBV4	22456	1136918	111966	6416 / 79.3	1547 / 19.5
DBV5	20457	1110112	109463	5670 / 49.7	1472 / 27.7
DBV6	24320	5351515	1980356	2212 / 4.7	333 / 0.5
DBV7	24168	5391085	2035734	2219 / 12.1	319 / 0.6

TABLE III: Post-Localization Reductions and Runtime

Bench-mark	Localized Inputs	Localized ANDs	Localized Registers	Unate Inputs	Unate ANDs	Unate Registers	Solving Time (sec)	Reduced Solving Time (sec)
DBV1	13728	115163	12531	12839	112849	12465	14906.1	7451.2
DBV3	16064	157329	15612	11923	156636	14923	29799.8	14037.5
DBV4	14395	122873	12664	13486	120279	12591	25213.4	8721.7
DBV5	14078	135856	13110	10431	135511	13043	10629.4	9797.1

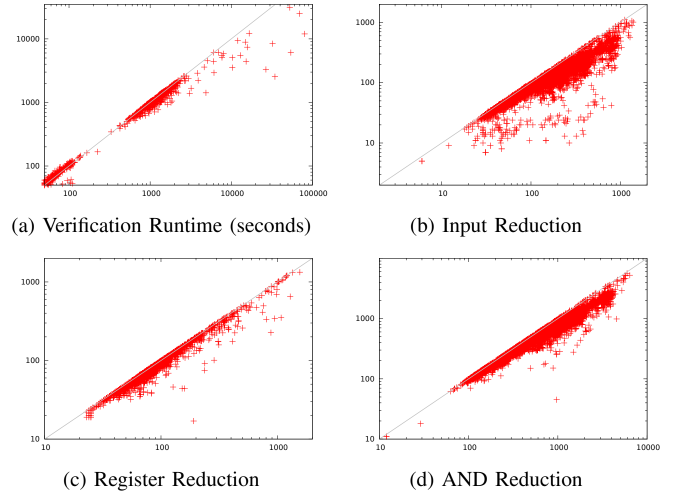
netlist transforms can often substantially reduce logic size, offering significant end-to-end verification speedups especially for larger and more-challenging properties.

We present reduction results in Table II using traditional reparameterization vs. our variant on several connectivity verification testcases, *before localization*. Columns 2 - 4 show original netlist size. Column 5 shows the number of inputs reduced by traditional reparameterization (and runtime), without our techniques. Column 6 shows the number of inputs reduced by Algorithm 2 (and runtime) *before* traditional reparameterization. Despite the large number of properties over disjoint fanin logic inherent in these benchmarks, these techniques yield significant reductions. Though virtually no sequentially-unate inputs exist w.r.t. all properties. Property-specific localization and netlist reductions are thus often necessary for scalable verification, as discussed next.

A. End-to-End Connectivity Verification Experiments

Table III presents the effectiveness of SUR *after property-specific localization*, for some difficult properties of Table II. Columns 2 - 4 show netlist size *after localization*. The number of gates reduced by SUR are presented in columns 5 - 7, up to 93.6% with an average of 83.8%. When synergistically combined with our and traditional reparameterization, 99.8% total reduction is achieved on DBV1 and DBV4. Columns 8 - 9 present our best solving times with vs. without our techniques; both include traditional reparameterization. Solving time includes reductions and IC3: localization runtime itself is not itself reported, though for properties requiring multiple abstraction-refinement iterations, their solving time is summed. SUR is trivially fast, <1 second on these runs. In these experiments, our techniques eliminate all inputs that traditional reparameterization could; due to their fast trace reconstruction, this greatly improves end-to-end scalability especially when spurious localization counterexamples occur.

In Figure 8a, we present more per-property end-to-end connectivity verification runtime comparisons with vs. without our reduction techniques, *after localization*. Overall verification runtime improved on average by 6.2% with a maximum of $13.6\times$. Note that for simpler properties that are easier to solve even without reductions, end-to-end runtime is on-par with or even slightly worse with the additional reductions given their runtime; speedup increases for more-difficult properties.



(a) Verification Runtime (seconds) (b) Input Reduction

(c) Register Reduction (d) AND Reduction

Fig. 8: Connectivity Verification Unate Reductions and Runtime: With SUR (y-axis) vs. Without SUR (x-axis)

In Figure 8b we show the number of inputs passing into IC3 with vs. without SUR, yielding up to $27.6\times$ greater reduction. In Figure 8c we show the number of registers, yielding up to $11.2\times$ greater reduction. In Figure 8d we show the number of AND gates, yielding up to $21.6\times$ greater reduction. These post-localization algorithm flows include an application of min-area retiming [13], reparameterization, and combinational optimizations before IC3, depicting reduced netlist size with vs. without SUR. The former enable greater sequentially-unate input reduction, contributing to faster solving runtime especially for more-difficult properties.

VIII. Conclusions

We presented two novel sound and complete netlist transformations, which substantially improve verification scalability while enabling highly-efficient trace reconstruction. The first is a scalable 2QBF variant of input reparameterization, capable of eliminating inputs without introducing logic and without full *range* computation. While traditional reparameterization has greater reduction potential, our technique can yield up to 4 orders of magnitude speedup to trace reconstruction when used as a preprocess to traditional reparameterization. Practically, this largely solves an occasionally-fatal risk where traditional reparameterization severely degrades end-to-end verification resource. The second is sequentially-unate input reduction, capable of substantial sequential logic reduction and boosting the probability of random input stimulus generation yielding valid counterexamples. Both are scalable enough to include in a standard state-of-the-art logic optimization flow, whether used for proving or bug-hunting. *Connectivity verification* particularly benefits from these reductions, up to 99.8%.

Acknowledgments: The authors wish to thank Pradeep Kumar Nalla for his insights in brainstorming discussions of the proposed techniques.

REFERENCES

[1] ABC: An academic industrial-strength verification tool. In *Computer Aided Verification*, July 2010.

[2] Hardware model checking competition, 2017.

[3] V. Balabanov, J. R. Jiang, A. Mishchenko, and C. Scholl. Clauses versus gates in CEGAR-based 2QBF solving. In *Beyond NP, Papers from the 2016 AAAI Workshop.*, 2016.

[4] J. Baumgartner and H. Mony. Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies. In *Correct Hardware Design and Verification Methods*, October 2005.

[5] A. Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, 2007.

[6] P. Bjesse. Word level bitwidth reduction for unbounded hardware model checking. In *Formal Methods in System Design*, August 2009.

[7] A. Bradley. SAT-based model checking without unrolling. In *international Conference on Verification, Model Checking, and Abstract Interpretation*, January 2011.

[8] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer Aided Verification*, July 1999.

[9] G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, and S. Quer. Speeding up model checking by exploiting explicit and hidden verification constraints. In *Design, Automation and Test in Europe*, April 2009.

[10] N. Eén and A. Mishchenko. A fast reparameterization procedure. In *International Workshop on Design and Implementation of Formal Tools and Systems*, October 2013.

[11] T. Glökler, J. Baumgartner, D. Shanmugam, R. Seigler, G. V. Huben, B. Ramanandray, H. Mony, and P. Roessler. Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning. In *Formal Methods in Computer Aided Design*, November 2006.

[12] H. Katebi and I. L. Markov. Large-scale Boolean matching. In *Design, Automation and Test in Europe*, March 2010.

[13] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. In *Computer Aided Verification*, July 2001.

[14] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1979.

[15] M. Soeken, P. Raiola, B. Sterin, B. Becker, G. De Micheli, and M. Sauer. SAT-based combinational and sequential dependency computation. In *Hardware and Software: Verification and Testing*, November 2016.

[16] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Formal Methods in Computer-Aided Design*, November 2002.

[17] P. Nalla, R. K. Gajavelly, J. Baumgartner, H. Mony, R. Kanzelman, and A. Ivrii. The art of semi-formal bug hunting. In *International Conference on Computer-Aided Design*, November 2016.

[18] S. M. Plaza, K. Chang, I. L. Markov, and V. Bertacco. Node mergers in the presence of don't cares. In *Asia and South Pacific Design Automation Conference*, January 2007.

[19] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? In *Information and Computation*, October 2002.