Formal Methods in Computer-Aided Design 2024

Toward Exhaustive Sequential Redundancy Removal

Rohit Dureja*, Jason Baumgartner*, Raj Kumar Gajavelly*, Robert Kanzelman*, and Kristin Y. Rozier†
*IBM Corporation †Iowa State University

***Abstract*—Hardware designs often contain logical redundancies: pairs of behaviorally-equivalent gates. *Sequential redundancy removal* is the process of removing gates that are behaviorally-equivalent within the reachable states of a design. It has many applications in the hardware design process, including logic optimization, equivalence checking, accelerating functional verification, and engineering change-order optimization.**

Redundancy removal is an intricate process, orchestrating various algorithms to compute equivalence-classes of potentially-equivalent gates, then to prove their validity. In this paper, we introduce techniques to enable *exhaustive* redundancy removal on practical designs, such as resource-balancing the underlying algorithms; self-tailoring them to sequentially-deep logic; and detailing an orders-of-magnitude optimization to the *Proof Graph* essential to proving non-inductive redundancies. We integrate these techniques within a state-of-the-art redundancy removal framework, illustrating their efficacy on various benchmarks.

I. Introduction

Hardware designs are often rife with logical redundancies. Some are deliberate, e.g. to improve circuit timing or implement error-resilience features. Many are unexpected and undesired; including them in semiconductor devices degrades cost and circuit performance, and increases power consumption.

In verification, logical redundancies are even more prevalent, e.g. due to input constraints disabling various functionality, and redundancies arising between design and testbench logic. *Equivalence checking* (EC) and *engineering change order* (ECO) tools compare two related designs; significant redundancy is common between those designs. Redundancy removal is highly-beneficial to verification scalability, solving some properties outright [1]–[3]; is the core solving procedure of EC [4, 5]; and can yield smaller ECOs [6].

Sequential redundancy removal frameworks (Fig. 1) identify, then eliminate, functionally-equivalent gates. Each suspected redundancy requires proving a property, called a *miter*, confirming that a pair of gates behave identically in the reachable states of a design. Simulation is used to refine incorrect equivalence-classes of gates, correcting inaccurate miters [4, 7]. Once a miter is proven, design size and power can be reduced by replacing one of its gates by the other [5]. The choice of which gate to eliminate can be delay- and placement-aware yielding higher-performance circuits.

Many techniques have been proposed to accelerate redundancy removal. For example: combinational redundancy removal solves miters from topologically-shallowest to deepest, reducing effort for deeper miters by leveraging early-merging and prior refinements [8, 9]. *Speculative reduction* models assumptions through structural logic simplifications, enabling a *transformation-based verification* (TBV) suite of model-checking algorithms to benefit from those assumptions to solve the non-inductive miters [2]. A *Proof Graph* enables early-merging of selective miters even before a fixedpoint of all-miters-proven is achieved, minimizing the number of proofs necessary to converge, and yielding reductions even if a resource-limit precludes convergence [10].

Contributions: We introduce various improvements to sequential redundancy removal in the pursuit of *exhaustiveness*. **(1)** We present *sequential resource-sweeping* (Sec. III-A) to self-tailor SAT-based bounded model-checking (BMC) [11] and induction to the sequential depth of the design, enabling them to solve deeper miters. This yields $\approx 5\%$ greater redundancy removal in less runtime via induction, and $\approx 20\%$ fewer incorrect miters deferred to TBV. **(2)** We propose techniques to balance counterexample simulation runtime with solving effort (Sec. III-B), yielding $\approx 30\%$ overall speedup (Sec. III-B). **(3)** We address scalability challenges of deep-counterexample generation and simulation, via: separate *eager shallow* vs. *lazy deep simulation* phases to accelerate $\approx 16\%$ additional deep–logic refinement (Sec. III-C); obtaining $\approx 34\%$ complementary deep refinements via seeded-state BMC (Sec. III-D); and minimally-lossy techniques to approximate pathologically-deep miters impractical to simulate (Sec. III-E). **(4)** We present a near-linear-runtime algorithm to construct a *Proof Graph* [10], improving scalability by orders of magnitude (Sec. III-F). Experiments in Sec. IV show our techniques yielding $2.1\times$ speedup to EC, 32.4% speedup with 16.9% more solves in model-checking, and enabling *exhaustive* redundancy removal on netlists up to 857110 AND gates, 75952 registers.

II. Preliminaries and Related Work

We represent a hardware design as a *netlist* N, comprising a directed graph $G = \langle V, E \rangle$. Vertices V represent logic gates of different types: constants, primary inputs, combinational primitives such as AND gates, and sequential primitives such as *registers*. Edges $E \subseteq V \times V$ represent interconnections between gates. The *fanin* (*fanout*) of gate u is the set of gates reachable by traversing edges backward (forward) from u. A *strongly-connected component* (SCC) is a set of gates having a directed path between every pair of gates within the SCC.

Registers have *initial values* defining their time-0 behavior, and *next-state functions* defining their time-$i{+}1$ behavior. A *trace* is a sequence of Boolean valuations to gates over timesteps, beginning from an *initial state* consistent with initial-values at time 0. A *state* is a Boolean valuation to the registers; a *reachable state* is one reachable along a trace. Certain gates may be labeled as *properties*, representing a verification objective to obtain a *counterexample trace* illustrating an assertion of that gate, or to prove the absence of any

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_27

```
exhaustive_sequential_redundancy_removal (Netlist $N$)
1: Create equivalence-classes of gates in $N$, where gate $u$ in class $Q(u)$
   is suspected functionally-equivalent to every other gate in $Q(u)$.
2: Select a representative gate $R(Q(u))$ from each class $Q(u)$.
3: Construct speculatively-reduced netlist $N'$ (§II-A2) replacing the
   source gate $u$ of each $(u,v) \in E$ by $R(Q(u))$; else copy $N' = N$.
4: For each gate $v$, add a miter to $N'$ falsified when $v \not\equiv R(q(v))$.
5: Construct a Proof Graph $P$ from $N'$ and the set of miters $M$.
6: Attempt to falsify or prove each miter (§II-A1).
7: If $P$ was computed or speculative-reduction was not used, merge the
   soundly-proven-miter gates onto their representatives (§II-A3).
8: If a miter was not proven, refine equivalence classes to separate their
   gates (simulating available counterexamples, §II-A4); goto Step 2.
9: Merge proven miter gates onto their representatives.
```

Fig. 1. Exhaustive sequential redundancy removal algorithm.

counterexample. During redundancy removal, properties called *miters* are created confirming the equivalence of postulated gate redundancies. Each miter is represented as an XOR over the suspected gate equivalence. Certain gates may be labeled as *observables*, e.g. primary outputs of the design. A *merge* of gate u onto gate v consists of moving output edges of u onto v, then deleting u. Any gate not in the fanin of a property or observable is irrelevant to netlist behavior, outside of its *cone of influence* (COI). After merging a pair of redundant gates, a secondary set of gates may become irrelevant in this way.

A. Redundancy Removal

Fig. 1 shows an exhaustive-capable sequential redundancy removal algorithm. It first overapproximates the redundancy candidates, represented as *equivalence classes* of gates suspected as pairwise functionally-equivalent in all reachable states. Initial equivalence-classes are constructed via: **(1)** runtime options and syntactic information, e.g. whether to compute redundancies only over registers vs. all gate types, and whether to pre-filter e.g. via *corresponded signal name pairs* in equivalence-checking; **(2)** compatible random-simulation signatures [4, 7]. (While not depicted for brevity, redundancy-removal can efficiently be performed *modulo inversion*.)

The *miter* properties M are then created and solved, comparing each gate to its equivalence-class representative. If a miter is falsified, its counterexample shows a miscompare of the corresponding gates (and/or fanin gates, due to *speculative reduction*). Simulating a counterexample on the original netlist often refines additional incorrect equivalence-classes. If a miter is unsolved due to resource limits or incomplete proof technique, it must be pessimistically pruned from its equivalence-class. If *all* miters are proven, the equivalence-classes represent valid redundancies; the netlist may be optimized by *merging* equivalent gate-pairs. Else the process repeats until this fixedpoint is achieved, or global timeout.

1) Induction: k-Induction [12] is commonly used to prove miters [1, 4]. The *base case* is bounded model checking (BMC), validating miters during the first *k* timesteps from the initial states. The *inductive step* verifies miters in *k* timesteps from any state (reachable or not) which satisfies the set of

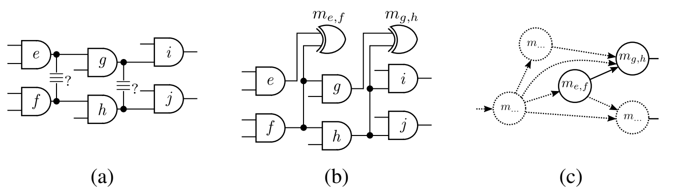


Fig. 2. Assumption modeling for scalable redundancy removal. (a) Suspected redundancies over gates. (b) Speculatively-reduced netlist with miters over suspected redundancies. (c) Proof Graph representing miter dependencies.

mutually-postulated equivalences within fewer timesteps. Using a sufficiently-large k and unique-state constraints, induction can be a *complete* proof technique [1]. Though BMC and induction solve NP-complete problems and become unscalable as k increases, rendering them *incomplete* in practice.

Redundancy removal of large netlists often requires solving millions of miters, partially because each gate may be compared to a sequence of varying representatives across refinements. BMC can efficiently falsify, and *k*-induction can efficiently prove, many miters. Though practical netlists often comprise non-inductive miters. Discarding even a single non-inductive yet accurate miter precludes *exhaustive* redundancy removal, and often causes an iterative avalanche of fanout miters to become unscalable as equivalence-classes are refined. Stronger algorithms (logic reduction, abstraction, alternate proof and falsification techniques) are thus useful to solve the non-inductive miters to enable exhaustiveness [2, 3].

2) Speculative reduction: An *assume-then-prove* paradigm enables assuming certain miters when proving others [1, 4]. *Speculative reduction* [2] models assumptions by replacing the fanout edge from each gate to an edge from its equivalence-class representative, aside from input edges to the miters themselves (necessary for soundness). Fig. 2a shows suspected-redundant gate pairs $\{e, f\}$ and $\{g, h\}$; Fig. 2b shows the speculatively-reduced netlist with miters $m_{e,f}$ and $m_{g,h}$.

Speculative reduction reduces the amount of logic in the fanin of each miter, making many inductive and often yielding orders of magnitude speedup. However, speculatively-reducing an incorrect miter alters fanout behavior of its non-representative *speculatively-reduced gate*, i.e. e and g in Fig. 2b. Therefore, this technique hinders the ability to early-merge a proven miter's gates until *all of its fanin speculatively-reduced gates* have been proven accurate. Therefore, speculative reduction may require a fixedpoint of proving all remaining miters, before any merge can be safely performed.

3) Early merging: A *Proof Graph* records the set of miters whose speculatively-reduced gates may alter the behavior of fanout miters, allowing to *early-merge* certain proven miters before converging a fixedpoint [10]. The Proof Graph is a directed graph $P = \langle M, D \rangle$ with one vertex in M per miter, and edges $D \subseteq M \times M$ representing dependencies of fanout miters upon the speculative-reduced gate of each miter. Miter m_1 is dependent on miter m_2 if the speculatively-merged gate of m_2 is in the fanin of either gate of m_1. Fig. 2c shows a Proof Graph for the speculatively-reduced netlist of Fig. 2b.

A miter proof is *sound* when the miters of all of its fanin speculatively-merged gates are proven. When a miter is soundly proven, its gates may be safely merged, regardless of other falsified or unproven miters. Sound proofs are propagated through fanout nodes of the Proof Graph, which may enable proven fanout miters to become soundly-proven. For k-induction, the Proof Graph only needs to record miter-dependencies relevant to the k-timestep unrolled netlist, along with dynamically-added SAT proof dependencies upon postulated induction-hypothesis constraints. For TBV, *all* transitive fanin dependencies are recorded, to ensure soundness using arbitrary model-checking algorithms [10].

The Proof Graph does not alter the set of miters to be proven. It improves scalability by: **(1)** prioritizing miter-solution order, generalizing combinational topological ordering [8, 9] to cyclic sequential netlists. *Leaves* can be solved first, minimizing effort wasted solving possibly-unsound miters. With parallel orchestration, leaf miters can be stubbornly solved, in parallel to time-balanced iteration among others [13]. **(2)** More redundancy is identifiable before global timeout. **(3)** The number of times each miter is repeatedly solved across refinements is reduced. This is especially important when using stronger model-checking algorithms to solve non-inductive miters: a PSPACE-complete problem. **(4)** Early-merging within the original netlist yields other speedups, e.g. faster simulation, unrolling, SAT, and future Proof Graph reconstruction.

4) Trace simulation: When a miter is falsified, simulating its counterexample on the original netlist identifies a set of inaccurate miters. Failed induction proofs yield counterexamples starting from possibly-unreachable *induction leak* states; those may be simulated to refine other non-inductive miters.

Simulation can consume significant runtime, and thus requires careful orchestration. *Bit-parallel simulation* atomically simulates 64 independent patterns in each 64-bit machine word. Counterexamples from BMC, induction, and TBV may be accumulated into machine words [7, 14], allowing each simulation to refine multiple counterexamples. Additional proposed improvements include packing *compatible* counterexamples into the same machine-word pattern [14]; randomizing unimportant input values; and permuting copies of counterexamples across patterns, e.g. with distance-1 modifications [15].

With shallow analysis, e.g. k-induction with small k, each miter affects only a local fanout region. This allows to decompose the netlist into slightly-overlapping components to be analyzed in parallel using fixed-depth simulation [5, 16, 17]. *Exhaustive* redundancy removal *additionally* requires deep analysis across more timesteps. As the depth of analysis increases, the inter-dependence of miters extends toward the entire cone-of-influence; windowing becomes ineffective, and simulation of the sequential netlist becomes inevitable.

III. Exhaustive Redundancy Removal

We describe the main contributions and experimentally evaluate their isolated impact on exhaustive redundancy removal

```
sequential_resource_sweeping (Netlist $N$, Miters $M$)
Miters $M_u := \emptyset$, $M_i := \emptyset$, $M_s := \emptyset$  # sets of miters
for $k \in 0, 1, 2, 3, \dots$ :  # iterate over increasing $k$-depth
    for satLimit $\in$ min, ..., max :  # iterate over SAT limits
        # run BMC to increase bounded-proof depth $\geq k$
        ⟨proved, falsified, unsolved⟩ := BASECASE($k$, $M$, $N$, satLimit)  # BMC
            ↪ Check all miters with proof-depth $< k$ using BMC
            ↪ Update proof-depth of newly-BMC-proved
            ↪ Simulate falsified miters to refine equivalence-classes
        $M := M \setminus$ falsified  # discard BMC-falsified miters
        $M_u := M_u \setminus \{$ proved $\cup$ falsified $\}$  # update unsolved miters
        if satLimit $\equiv$ max : $M_u := M_u \cup$ unsolved  # cache unsolved miters
        # run induction on miters adequately checked by BMC
        $M_i := M$  # snapshot active-miters for later rollback from induction leaks
        if $M_s \not\equiv \emptyset$ :
            $M := M_s \cup$ newly-BMC-proved, $M_s := \emptyset$  # restore snapshotted miters
        else
            $M := M \setminus$ (miters with proof-depth $< k$)  # base-case inconclusive
            $M := M \setminus M_u$  # drop miters unsolved in prior induction steps
        repeat  # fixedpoint (FP) iterations
            $N'$ := SPECREDUCE($N$, $M$), Graph $G$ := PROOFGRAPH($N'$, $M$)
            ⟨proved, falsified, unsolved⟩ := INDUCTIVESTEP($k$, $G$, $N'$, satLimit)
                ↪ Check miters in leaves of Proof Graph $G$
                ↪ Update Proof Graph for proven miters  # enable early-merging
                ↪ Simulate falsified miters to refine equivalence-classes
            $N$ := EARLYMERGE($N$, $G$) and remove merged miters from $M$, $M_i$
            $M := M \setminus$ falsified  # drop falsified miters (induction leaks)
            if satLimit $\equiv$ max : $M_u := M_u \cup$ unsolved  # cache unsolved miters
            else if $|$unsolved$| > 0$ and $M_s \equiv \emptyset$ :  # FP iteration with unsolved
                $M_s := M$  # snapshot miters for next SAT iteration
            $M := M \setminus$ unsolved  # drop inconclusive miters
        until fixedpoint (no unsolved or falsified) for $k$ at satLimit
        $M := M_i$  # restore active-miter snapshot to roll-back induction leaks
    if $n$-steps with no merging or timeout : break  # self-tailor depth
```

Fig. 3. Sequential resource-sweeping using BMC and k-induction

in this section. End-to-end experimental results for various formal applications appear in Section IV.

A. Sequential resource-sweeping

Most miters are easy to solve at shallow BMC or induction depth, becoming unscalable as depth increases. Because satisfiability checking is NP-complete, some miters are pathologically-difficult, even at shallow depth. Large netlists often contain a diversity of logic, often comprising a mix of easier and difficult miters.

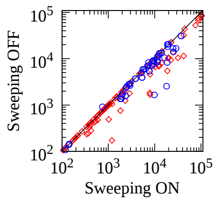
Borrowing from combinational equivalence checking [8, 9], sequential redundancy removal frameworks typically solve unfolded miters from topologically-shallower to deeper. Shallow miters are often easier; their solution simplifies fanout miters through merging unfolded gates, and refining incorrect equivalence-classes. Resource-limits may be applied, and another fixedpoint iteration attempted after refining unsolved miters. Due to diversity of miter-difficulty, combinational netlist simplification via BDD- and SAT-sweeping may benefit from iterating unsolved miters with increasing resource-limits, solving gradually-more-difficult miters without indefinite delays caused by pathological miters [7, 9].

For *exhaustive* sequential redundancy removal, additional resource-sweeping controls and equivalence-class management are necessary across k values, and to optimally defer miters into TBV. We introduce *sequential resource-sweeping* in Fig. 3 to manage these intricacies. For each k-depth, each miter is checked using increasing SAT-resource limits (lines 3–32). The induction base-case is validated by BMC (lines 5–8). The miters falsified with BMC are inaccurate, and permanently discarded (line 9). Any miter unsolved by BMC is skipped for that resource-limit, as induction would be unsound (line 17). The remaining miters are checked using induction (lines 19–31). Miters are solved from topologically-shallowest to deepest, using an inductive Proof Graph [10]; early-merging of soundly-proven miters is performed after iterating its leaves (line 25), offering runtime benefits (Sec. II-A3). Any miter unproven by induction is discarded (line 26 and 30). Because k-induction is incomplete, it may *spuriously* falsify accurate miters. For exhaustiveness, it is thus essential to snapshot and restore *active* miters M_i (line 13 and 32, resp.) to prevent induction leaks from permanently discarding accurate miters.

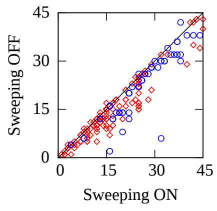
After completing a lossy fixedpoint, SAT resource is incremented up to a configurable maximum value, and previously-unsolved miters M_s are snapshotted to check anew (line 29). Any miter unsolved by the maximum SAT-limit during BMC (line 11) or induction (line 27) is deferred to TBV (line 18). Miters are then restored from M_s if available, incrementally reusing prior effort (line 15). Note that any proofs obtained using a subset of miters are valid for a superset. Thus induction-*proved* results when M_s was snapshotted may be re-applied after restoring, despite adding any newly-BMC-*proved* miters. However, the induction-hypotheses upon which reused proofs relied must also be annotated onto the inductive Proof Graph. Prior refinements reflected in M_s may be pessimistic within the current SAT-limit and k if any newly-BMC-*proved* miters are added. While spuriously-refined miters will be checked at higher k values or by TBV, this temporary lossiness may be compensated for, by: **(i)** using a dedicated pre-induction BMC phase, which often helps overall scalability anyway; **(ii)** skipping the restoration of M_s if newly-BMC-*proved* is non-empty; or **(iii)** selectively restoring previously-falsified miters that share logic with newly-BMC-*proved* to check anew.

The algorithm terminates after n timesteps without conclusive result, or global timeout (line 33). *Exhaustive* redundancy removal often benefits from deeper BMC than induction, to minimize wasted effort trying to prove inaccurate miters and to accelerate early-merging; $n \geq 5$ for BMC, and $n \geq 2$ for induction, are used in all experiments. While many miters are sequentially shallow, robust redundancy removal requires self-tailoring to deeper netlists. Solving easier miters at greater BMC and induction depth is often faster than deferring those to TBV: inductive Proof Graphs have fewer dependencies, enabling earlier merging; counterexample generation for BMC requires less reconstruction effort than through a sequence of TBV engines. Thus it is beneficial to defer only the difficult miters to TBV, not the easier deep miters.

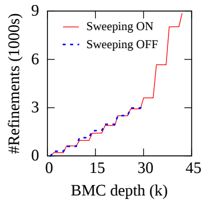
Most prior work imposes SAT resource-limits via bounding backtracks [9] or decisions [7]. While effective, those do not closely align with actual runtime. Time-limits are volatile and hurt reproducibility. We have found the number of propagations as a reproducible runtime-aligned metric, which also allows balancing simulation and solver runtime (Sec. III-B).



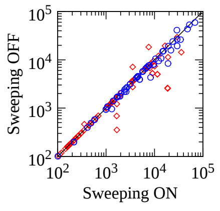
(a) #BMC refinements



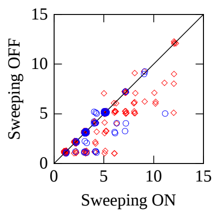
(b) BMC depth



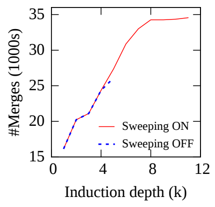
(c) #Refinements vs k



(d) #Induction merges



(e) Induction depth



(f) #Merges vs k

Fig. 4. Sequential resource-sweeping effectiveness: BMC and Induction for industrial logic optimization (○) and model-checking benchmarks [18] (◇)

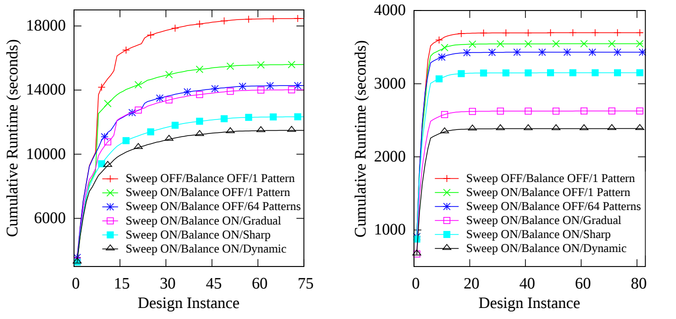
We evaluate sequential resource-sweeping in Fig. 4 for difficult industrial logic optimization and model-checking benchmarks [18]. Redundancy removal begins with a *refinement phase* of increasing-depth BMC, with 1800-second timeout early-terminated by 5 timesteps without refinement; then a *proof phase* of increasing-depth induction with 36000-second timeout early-terminated by 2 timesteps without merging.[1] SAT propagation-limits increment by $10\times$ from 10000 to 10000000 when resource-sweeping, and are unbounded otherwise. For industrial benchmarks, sequential resource-sweeping enables 30.3% deeper BMC on average (32.4% for model-checking) (Fig. 4b), yielding up to 20.7% more refinements (23.1% for model-checking) (Fig. 4a). It also enables deeper induction (Fig. 4e), yielding 5.2% more merged gates on average (3.4% for model-checking) (Fig. 4d). Note that unbounded SAT-resource can sometimes solve a *lucky miter* with modestly more than the maximum resource-sweeping SAT-limit, and thus occasional modest losses for resource-sweeping occur. These are offset by more-frequent, larger wins by deferring difficult-for-SAT miters to TBV, while enabling BMC and induction to solve easier deeper miters.

Fig. 4c and Fig. 4f show detailed per-k refinement and merging respectively for a single deep benchmark. BMC completes 12 more steps, and induction goes 6 steps deeper, with resource-sweeping enabled. While some common success is achieved at shallower k, self-tailoring depth when resource-sweeping yields 201.5% more refinements and 32.4% more merged gates at greater depth.

B. *Resource-balanced simulation*

Bit-parallel simulation allows atomically refining multiple counterexamples, packed into each bit of a machine word.

[1]The time-limits used in these experiments are rarely encountered, though are imposed for uncommon yet inevitable pathological scenarios.



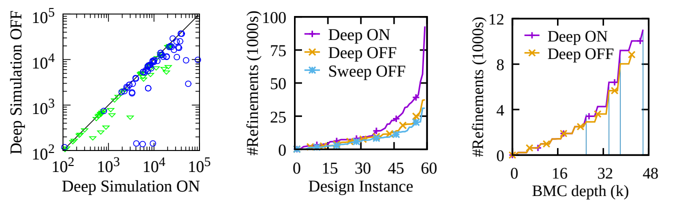
(a) SEC Benchmarks (b) Model-checking benchmarks [18]

Fig. 5. Cumulative runtime with resource-balanced simulation

Resource-sweeping ensures that easier miters are solved early, while difficult miters are deferred. When SAT-limits are small, miters are solved quickly: accumulating more traces before simulating is often faster. As SAT-limits increase, miter solving is slower: simulating fewer traces more-frequently is often faster. For greatest scalability, it thus is beneficial to tailor simulation frequency as miters are solved within the resource-sweeping loop of Fig. 3 (lines 8 and 24).

Predictive tuning of how many traces to accumulate vs. SAT-limit can be improved via *dynamic resource-balancing*, comparing runtime of the prior simulation to runtime of solvers since the last simulation. When either exceeds the other by more than a configurable threshold, simulation can be deferred or expedited. (For better reproducibility, runtime can be estimated by comparing number of gates and timesteps simulated vs. SAT-propagations since the last simulation.) While resource-sweeping ensures somewhat-balanced resource per miter, dynamic balancing adjusts for factors such as the percentage of miters proven vs. falsified vs. resource-exceeded, or unexpected solver-resource variance using less-predictable model-checking algorithms via TBV.

We evaluate resource-balancing in Fig. 5a on industrial sequential equivalence-checking (SEC) benchmarks, and Fig. 5b on model-checking benchmarks. SAT-limits increment by $10\times$ from 10000 to 1000000 propagations. Cumulative runtime is plotted for: **(1)** resource-sweeping disabled (baseline); simulation-resource balancing disabled with **(2)** each trace simulated vs. **(3)** 64 traces accumulated; resource-balancing enabled with **(4)** gradual tapering of 64 patterns at 10000 propagations, 32 at 100000, and 1 at 1000000, **(5)** sharp tapering from 64 patterns at 10000 propagations to 1 above; **(6)** dynamic runtime balancing. Note that unsolved benchmarks are not plotted, for clarity. Settings **(3)-(6)** are 8.38%, 10.07%, 20.87% and 27.28% faster, respectively, than **(2)** in Fig. 5a; vs. 3.24%, 25.92%, 11.18% and 32.71% in Fig. 5b. The reason that sharp tapering wins for SEC, and gradual for model-checking, is due to the percentage of miters falsified: corresponded signal-name pairing for SEC ensures fewer incorrect non-inductive miters. Dynamic balancing adjusts effectively to either scenario. Setting **(1)** for SEC without resource-sweeping is 18.4% and 60.6% slower compared to setting **(2)** and **(6)**,



(a) #Refinements (log) (b) #Refinements (c) #Refinements vs k

Fig. 6. Deep simulation BMC for logic optimization on industrial ($\circ$), and public QUIP [20] and ITC99 [21] benchmarks ($\triangledown$)

respectively; vs. 4.23% and 54.9% slower for these easier model-checking benchmarks.

C. Lazy deep simulation

In *lossy* redundancy removal frameworks using only shallow fixed-depth induction [5, 16], deep simulation is unnecessary: incorrect miters are discarded along with accurate non-inductive miters. *Exhaustive* redundancy removal requires equivalence-classes to be corrected before miters can be proven. Deep simulation may arise due to deep counterexamples found by BMC or TBV, or simulating beyond shallower trace length to propagate refinements through fanout logic. While deep simulation is expensive, its refinements offset the expense of explicitly computing long miter counterexamples as NP- or PSPACE-complete problems.

Simulation consumes nearly identical runtime per timestep. When balancing solver vs. *imminent* simulation runtime considering *prior* simulation runtime, depth should be considered. We introduce the concept of *lazy deep simulation* using its own *deep-simulation trace storage*, distinct from traditional *eager shallow simulation* for minimal-depth simulation. *Eager shallow simulation* is faster, and should occur more-frequently using fewer traces. Very-shallow simulation may be parallelized via simulating slightly-overlapping windowed components [5, 16, 17]. Lazy deep-simulation is slower, evaluating the sequential netlist. It thus should occur less-frequently, sometimes accumulating more traces into *multiple* machine words to leverage multi-word simulation speedup [19].

Shallow vs. deep simulation benefit from different resource-balanced parameters affecting how many timesteps beyond trace length to simulate: a *maximum extension* parameter, and an *inactivity limit* to early-terminate the extension if insufficient refinements occur during the prior n timesteps.

We evaluate lazy deep simulation during BMC in Fig. 6 for industrial and public logic optimization benchmarks. Deep simulation is run when 640 patterns are accumulated, with a maximum overall 900-second timeout early-terminated with maximum extension of 2048 (vs. 10 for shallow), and inactivity-limit of 100 (vs. 4 for shallow). Deep simulation enables a median 15.78% more refinements (Fig. 6a), and median 30.15% compared to disabled sequential resource-sweeping (Fig. 6b). Fig. 6c shows per-k refinement for the deep benchmark of Fig. 4c. Vertical lines indicate running deep simulation, thrice after accumulating 640 patterns and once as

BMC terminates. Each deep simulation provides 12.56% more refinements on average, improving BMC (and subsequent proof) scalability. BMC completes 6 additional timesteps via this speedup, yielding 39.48% additional refinements.

D. Seeded BMC

When a miter is falsified, simulating its counterexample often refines additional miters. Randomizing and permuting bit-parallel patterns, and simulating beyond trace length, increase the number of secondary refinements. Though the return on runtime investment, and probability of secondary refinements, via additional simulation often quickly saturates.

Secondary refinements may be obtained via semi-formal methods: when simulation encounters an interesting state (e.g. one that refines an equivalence-class), one can seed BMC into that state, trying to falsify additional miters [19]. If miters are falsified by seeded BMC, eager simulation of those counterexamples can be accelerated starting from the seeded state vs. initial state; especially valuable for very-deep TBV traces. Seeded-BMC traces may also be appended to the *deep-simulation trace storage* to increase the probability of secondary refinements during later deep simulation. To balance overall runtime, seeded BMC runtime may be configured to a fraction of time-elapsed to obtain those counterexamples.

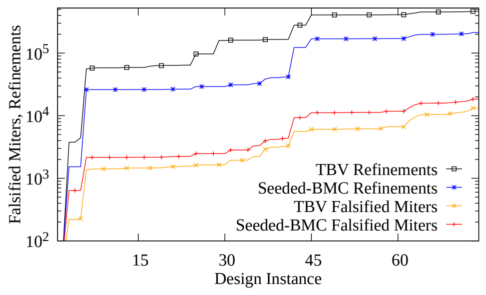
One challenge to maximizing seeded BMC refinements is that controllability to propagate the new refinement scenario into adjacent logic may be limited by prior input valuations locked into the seeded state. It thus is often useful to seed BMC into a state several timesteps *before* the refinement of interest. The optimal number of timesteps varies by netlist and by miter. It can be approximated by the number of registers along simple paths between the refined equivalence class and primary inputs. It can also be varied, dynamically adjusting to a setting suitable for the netlist.

We evaluate seeded BMC in Fig. 7a during logic-optimization of public design benchmarks. Seeded BMC is run *directly after* refining TBV counterexamples, starting from already-refined equivalence-classes; their refinements are thus complementary. Seeded BMC runtime is limited to the lesser of 200 seconds, 25% of TBV runtime to produce the trace, or 50 timesteps without refinement. Seeded BMC thus consumes at most 25% of TBV runtime; in these experiments it consumed less than 3%. Despite the advantages given to TBV, seeded BMC yields 39.2% more falsified miters than TBV, while TBV generates 2.18× as many refinements. Fig. 7b identically evaluates larger proprietary industrial designs. Seeded BMC yields 11.1% more falsified miters than TBV, while TBV generates 1.7× as many refinements.

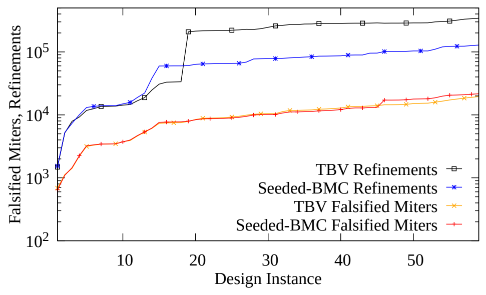
E. Approximating pathologically-deep logic

Industrial netlists often comprise sequentially-deep logic components, such as large counters or linear-feedback shift-registers (LFSRs). Incorrect miters dependent upon such components might be falsifiable with traces of minimum-length exponential with respect to their register count. Such components pose two challenges. **(i)** BMC and induction become unscalable too-shallowly to converge very-deep miters. TBV is often necessary, using heavier model-checking algorithms to solve PSPACE-complete miters, including localization [22] and logic reductions to reduce miter size, phase abstraction [23] to reduce sequential depth, well-tuned IC3 [24, 25] and BDD-based reachability [26] as general solvers, and semi-formal bug-hunting [19]. **(ii)** Simulating a very-deep trace may be prohibitively slow. While TBV may use tailored techniques to analyze deep miters (e.g. parameterized traces [27]), speculative-reduction and logic transformations applied within TBV require simulating traces on the original netlist (not only its deep components) to enable precise refinements. For extremely-deep traces, potentially billions of timesteps long, lossy shortcuts are inevitable. The following are useful minimally-lossy strategies.

(1). As refinement depth or BMC bound exceed a configurable threshold, a *depth-compensation graph* can annotate gates with: **(i)** most-recent refinement depth; **(ii)** most-recent bounded proof depth; **(iii)** the snapshotted equivalence-class for which the former were obtained. The behavior of known-deep gates can be randomly permuted when simulating bit-parallel *copies* of a trace. Because the trace pattern itself is not permuted, precise refinements occur. The random-permutation may cause pessimistic refinements, discarding valid miters. To limit the pessimism, permutation can be synchronized across the snapshotted equivalence-class gates, similar to induction counterexamples [28]. If new refinements occur, simulation may be rolled-back, and newly-mismatched candidates may be permuted for continued simulation. This process can flatten a pathological sequence of double-length counterexamples across refinements to near-linear simulation depth, similar



(a) QUIP [20] and ITC99 [21] benchmarks



(b) Industrial benchmarks

Fig. 7. Cumulative seeded BMC vs. TBV refinements for logic optimization

to forcing X-pessimism to accelerate convergence in three-valued approximate reachability analysis [29].

(2). Seeded BMC (Sec. III-D) can falsify deeper miters from a simulated state, though often becomes unscalable too shallowly to converge on large counters. It is sometimes useful to *overapproximate* seeded BMC, randomly permuting the value of known-deep gates, to be less dependent upon simulation probabilities to propagate permuted values.

(3). In cases, the first concerningly-deep counterexample is too deep to simulate, or even to generate. E.g., if a design has a 64-bit LFSR, each bit may toggle shallowly, though a compare-to value may only be reached incredibly-deeply. If simulating a counterexample is impractical, the offending miter may be blindly refined as if unsolved, losing the ability to propagate refinements to fanout logic. Simulation permutation of deep gates may nonetheless be useful in other ways, e.g. reusing previous (or future) counterexamples [14].

F. *Linear Proof Graph construction*

The Proof Graph [10] has one node per miter. Fanout edges represent the miters whose behavior is compromised by speculative-reduction until that node's miter is proven. The miters correlating to a node may be merged as soon as its fanin miters are proven, regardless of other unproven miters. Early merging has many runtime benefits (Sec. II-A3).

The Proof Graph is reconstructed whenever a new speculatively-reduced netlist is created, at each iteration of Fig. 1. Scalable construction is thus critical. *Lossy* redundancy removal, e.g. using shallow fixed-depth induction [5], may choose to not use a Proof Graph, instead deferring all merging until fixed-point. Despite the overhead of deferred merging, e.g. causing repeated proving of accurate miters across refinements, this shortcut is motivated by the traditional runtime overhead of constructing the Proof Graph vs. lossy-solver runtime. Early merging is a practical necessity to enable *exhaustive* redundancy removal on large netlists, which requires solving PSPACE-complete non-inductive miters via general model-checking algorithms. In this section, we describe a scalable graph-labeling algorithm to compute a minimally-sized Proof Graph.

When using arbitrary model-checking algorithms to solve miters, *all* transitive fanin dependencies are recorded in the Proof Graph. Traditional iterative construction (e.g. [10] Alg. 7) traverses the fanin of each miter m' to find the speculatively-reduced gates M' which affect its behavior (stopping at vs. recursing through M', to contain runtime); edges from M' to m' are iteratively added to the Proof Graph. This initial Proof Graph can be vastly larger than a condensed version due to duplicate and transitively-implied edges, risking memout. Postprocessing is proposed to *condense* the Proof Graph to be irredundant and acyclic [10], reclaiming memory before TBV. Though iterative fanin traversal and compaction are often a runtime bottleneck on large netlists.

We propose a method to directly compute an optimally-sized Proof Graph (Fig. 8), using a single netlist traversal. Our algorithm uses an efficient graph-labeling approach [31, 32], propagating miter-dependency information as a *bitvector*. Each speculatively-reduced gate is represented with a unique bit-index, though all miters within an SCC reuse a single bit-index, yielding a massive practical compaction.

```
createProofGraph_graphLabeling (Spec-Reduced Netlist $N'$, Miters $M$)
 1: Compute SCC within $N'$  # Tarjan's linear algorithm [30]
 2: for each gate $g \in$ topologically-sorted gates in $N'$ :
 3:   if $g$ is not in a multi-gate SCC :
 4:     if $g$ is speculatively-reduced :  # $g$ is the non-rep. gate of a miter
 5:       Miter $m$ := miter corresponding to $g$
 6:       # add dependencies of $m$ in the fanin of $g$
 7:       for each miter $n$ with index i such that bitvector($g$)[i] $\equiv$ 1 :
 8:         add_edge($n$, $m$)  # add dependency $n \rightarrow m$ to graph
 9:       # create singleton bitvector for miter $m$
10:       unsigned idx := get unique index for miter $m$
11:       clear bitvector($g$); bitvector($g$)[idx] := 1  # singleton bitvector
12:     # copy / union bitvector to fanout gates
13:     for each gate $h$ in the fanout of $g$ :  # propagate to fanout
14:       if $h$ is part of SCC $S$ : $h$ := representative gate of SCC $S$
15:       bitvector($h$) := bitvector($h$) $\cup$ bitvector($g$)  # copy / union
16:     delete bitvector($g$)  # cleanup
17:   else if $g$ is representative gate of SCC $S$ :
18:     if $S$ contains miters :
19:       # add cyclic dependencies between miters in SCC $S$
20:       Miters $M$[ ] := get all miters in $S$, unsigned j := 0
21:       while j+1 < size($M$) :
22:         Miter $n$ := $M$[j], Miter $m$ := $M$[j+1]
23:         add_edge($n$, $m$)  # add dependency $n \rightarrow m$ to graph
24:       Miter $n$ := $M$[size($M$)], Miter $m$ := $M$[0]
25:       add_edge($n$, $m$)  # add dependency $n \rightarrow m$ to graph
26:       # add dependencies for miters in the fanin of SCC $S$
27:       Miter $m$ := miter corresponding to representative gate of SCC $S$
28:       for each miter $n$ with index i such that bitvector($g$) [i] $\equiv$ 1 :
29:         add_edge($n$, $m$)  # add dependency $n \rightarrow m$ to graph
30:       # create singleton bitvector for miter $m$
31:       unsigned idx := get unique index for miter $m$
32:       clear bitvector($g$); bitvector($g$) [idx] := 1  # singleton bitvector
33:     # copy / union bitvector to fanout gates
34:     for each gate $h$ in the fanout of gates in $S$ :  # traverse to fanout
35:       if $h$ is part of SCC $T$ : $h$ := representative gate of SCC $T$
36:       bitvector($h$) := bitvector($h$) $\cup$ bitvector($g$)
37:     delete bitvector($g$)  # cleanup
```

Fig. 8. Graph labeling algorithm for Proof Graph construction.

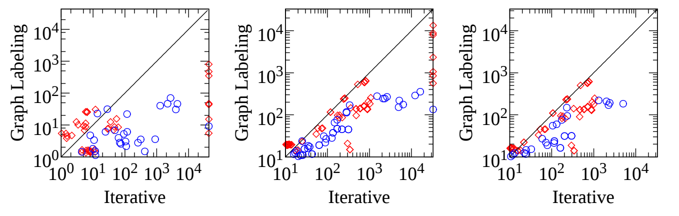
Linear SCC identification [30] identifies the strongly-connected components; each gate within an SCC is given the bit-index of a *representative* miter therein (line 1). The algorithm then iterates gates in a topological order, propagating fully-populated bitvectors denoting miter dependencies to fanout logic. When gate g is traversed, bitvector copy or union operators propagate g's bitvector to fanout gates (lines 13–15 and 34–36), accumulating their fanin dependencies. If g is a speculatively-reduced gate, all miters with indexed bits set to 1 in its bitvector are added as Proof Graph fanin edges to the node for g's miter (lines 7–8). Fanout edges of speculatively-reduced gates propagate only that gate's corresponding bit-index vs. all transitive fanin dependencies (lines 10–11).

A single bitvector is maintained for all nodes in an SCC, at its representative gate g (line 14, 35). If the SCC contains miters, a single unique bitvector index for a representative miter m therein is associated with all of that SCC's miters (line 27). The dependencies (asserted bitvector indices) of all SCC inputs become Proof Graph fanin edges of the SCC-

TABLE I
PROOF GRAPH CONSTRUCTION RESOURCES FOR LOGIC OPTIMIZATION

#	#Gates	#Miters	Iterative [10]		Graph Labeling (Fig. 8)	
			Time (s)	Memory	Time (s)	Memory
b1	4,476,762	192,445	4368.8*	**340 GB***	9.12	134.5 MB
b2	1,043,224	969,135	889.3	15.8 GB	3.58	353.8 MB
b3	833,584	517,979	420.1	11.9 GB	1.49	188.6 MB
6s125	3,232,742	2,885,658	>684	>**32 GB**	15.13	1.03 GB
6s350	3,774,149	1,511,759	>14400	5.6 GB	47.02	559.78 MB

BOLD: 32GB memout. [*]: runtime, memory estimate for complete construction.



(a) Time (seconds) (b) Peak memory (MB) (c) Graph size (MB)

Fig. 9. Proof Graph construction resources for logic optimization on industrial logic optimization (○) and model-checking benchmarks [33] (◇)

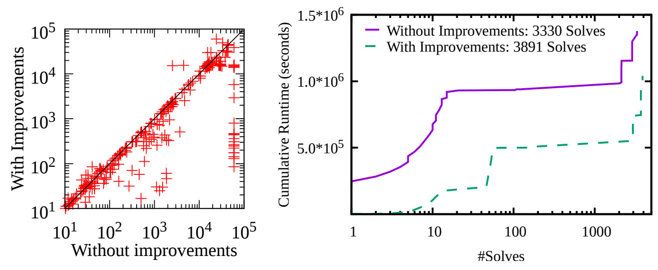
representative miter m (lines 28–29). The one-hot bitvector for m is then propagated to all SCC outputs (lines 31–32). Miters inside an SCC are added to the Proof Graph in lines 20–25 as a cyclic chain through representative m, to minimally record their inter-dependency.

Propagating a one-hot bitvector to the fanout of speculatively-reduced gates ensures a minimally-sized Proof Graph without unnecessary transitively-implied edges. When later annotating the Proof Graph with solver results, transitive traversal is generally necessary anyway [10]. Adding transitively-implied edges tends to degrade performance, bloating graph size and requiring more unsuccessful fanout-edge traversals: until a miter is proven, soundly-proven results for fanin miters cannot propagate through that miter anyway.

Theorem 1 (Sound)**.** *Given speculatively-reduced netlist N' and miters M, Fig. 8 generates a* sound *Proof Graph: every fanout dependency of speculatively-reduced gate g is transitively reachable via fanout traversal of g's Proof Graph node.*

Theorem 2 (Optimal)**.** *Proof Graph $P = \langle M, E\rangle$ generated by Fig. 8 has minimal size. I.e., there does not exist a Proof Graph P' with fewer nodes or edges correctly representing the dependencies of N' and M.*

Table I shows resources to construct the Proof Graph for selected large industrial and model-checking benchmarks, using the traditional iterative approach [10] vs. graph-labeling (Fig. 8). This highlights the scalability limitation of the former, precluding TBV on the largest netlists. Fig. 9 shows resources across more industrial and model-checking benchmarks; each run with 32GB memory-limit and 14400-second timeout. Graph-labeling is up to 98.34× (Fig. 9a) faster with a 53.77% lower peak memory requirement on average (Fig. 9b), and enables the Proof Graph creation for 17 more benchmarks within resource-limits. While the iterative approach postprocesses the



(a) SEC runtime (seconds) (b) Runtime vs # solves, model-checking

Fig. 10. Runtime for SEC and model-checking [33] benchmarks

Proof Graph by SCC compaction and pruning transitively-implied edges, graph labeling generates a 59.25% smaller Proof Graph without costly postprocessing (Fig.9c).

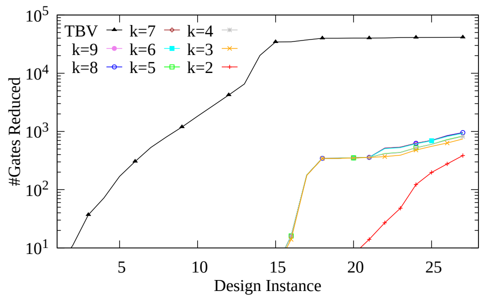
IV. EXPERIMENTAL RESULTS

In this section, we show end-to-end results for various redundancy removal applications, using the contributions of Sec. III. Though to minimize memouts, these all benefit from the scalable Proof Graph presented in Sec. III-F. Our techniques are implemented within RuleBase: Sixthsense Edition [34], building upon the techniques presented in [13, 32].

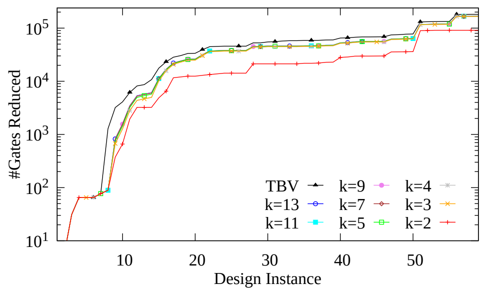
1) Property- and Equivalence-Checking: Fig. 10a compares end-to-end runtime with *vs.* without our improvements on 384 SEC benchmarks, using the refinement phase of Fig. 6 with 15-second seeded BMC, 4-hour induction, and 12-hour overall time-limits using 1-process TBV. Our improvements solve 362 common benchmarks 2.1× faster on average, with 22 unique solves. Fig. 10b shows model-checking speedup, using a similar configuration though without corresponded signal-name filtering. Cumulative runtime is 32.4% faster, and 16.9% more properties are solved using our improvements.

2) Logic Optimization: We present logic optimization results in Fig. 11a for public design benchmarks, and Fig. 11b for industrial designs. *Exhaustive* logic optimization is a global concern [35], and particularly challenging: **(1)** Unlike SEC, name-correlation cannot fragment equivalence-classes to pairs. Large equivalence-classes require *many* refinements to correct, iteratively comparing a gate to *many* different representatives. **(2)** Redundancy removal is not early-terminated when properties are solved. These benchmarks were preprocessed using 1-induction. On public benchmarks, preprocessing removes 1442832 gates. Deeper-k induction removes an additional 959 gates, then TBV an additional 40494, for 2.9% greater redundancy removal overall. Redundancy-removal was *exhaustive* for 65 of 74 netlists (containing up to 373338 AND gates and 8809 registers), capping resource at 64GB 3-day 5-process. No-reduction points are not plotted. Miters of depth ≥ 32768 (up to 250125) were encountered on 3 netlists; synchronized random-permutation (Sec. III-E) losslessly aided convergence.

The industrial benchmarks are evolving components, already redundancy-removed at prior logic iterations. Leveraging our contributions, deeper-k induction removes an ad-



(a) QUIP [20] and ITC99 [21] benchmarks



(b) Industrial benchmarks

Fig. 11. Cumulative deep-k induction and TBV logic optimization

ditional 167462 gates (76259 with $k \geq 3$), and TBV an additional 14248. Practical benefits were significantly greater due to multiple on-chip copies of each component. Exhaustiveness was achieved for 11 of 59 netlists (containing up to 857110 AND gates and 75952 registers). Miters of depth ≥ 32768 (some above 100M) were encountered on 21 netlists. Without the contributions presented in this paper, exhaustive redundancy removal was achievable for only 2 netlists.

V. Conclusions

Global energy use of semiconductor devices doubles every three years, mandating a new type of Moore's Law for energy efficiency [35]. Sequential redundancy removal is one of many remedies, and has many other applications.

We introduce various techniques to improve the scalability of sequential redundancy removal, eliminating bottlenecks to *exhaustiveness*. *Sequential resource-sweeping* self-tailors to netlist depth, yielding $> 20\%$ greater equivalence-class refinement via BMC and $\approx 5\%$ greater inductive redundancy removal, on average. *Resource-balanced simulation* accelerates redundancy-removal by $\approx 30\%$ on average. *Lazy deep simulation* yields $\approx 16\%$, and *seeded-state BMC* $\approx 34\%$, additional refinements with minor runtime overhead. Graph-labeling *Proof Graph* construction boosts scalability by two orders of magnitude, making TBV practical on very-large netlists. Heuristics are introduced to prevent pathologically-deep logic from derailing convergence. Overall, our techniques yield 2.1$\times$ speedup to SEC, 32.4% speedup with 16.9% more solves on large difficult model-checking problems, and enabled *exhaustive* redundancy removal on large netlists up to 857110 AND gates, 75952 registers.

Acknowledgments

We would like to thank the reviewers for their valuable feedback and comments. Kristin Y. Rozier is supported by NSF:CCRI grant 2016592 and NSF:CAREER grant 1664356.

References

[1] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 409–426, Oct 2000.

[2] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *Design Automation Conference (DAC)*, pp. 463–466, Jun 2005.

[3] R. Brayton, N. Een, and A. Mishchenko, "Using speculation for sequential equivalence checking," in *International Workshop on Logic and Synthesis (IWLS)*, Jun 2012.

[4] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *Design, Automation and Test in Europe (DATE)*, pp. 618–623, Feb 1998.

[5] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *International Conference on Computer-Aided Design (ICCAD)*, 2008.

[6] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri, "DeltaSyn: An efficient logic difference optimizer for ECO synthesis," in *2009 International Conference on Computer-Aided Design (ICCAD)*, pp. 789–796, 2009.

[7] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *International Conference on Computer-Aided Design (ICCAD)*, November 2004.

[8] D. Brand, "Verification of large synthesized designs," in *International Conference on Computer-Aided Design (ICCAD)*, November 1993.

[9] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, 2002.

[10] M. Case, J. Baumgartner, H. Mony, and R. Kanzelman, "Optimal redundancy removal without fixedpoint computation," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 101–108, Oct 2011.

[11] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems* (W. R. Cleaveland, ed.), (Berlin, Heidelberg), pp. 193–207, Springer Berlin Heidelberg, 1999.

[12] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 127–144, Nov 2000.

[13] R. Dureja, J. Baumgartner, R. Kanzelman, M. Williams, and K. Y. Rozier, "Accelerating parallel verification via complementary property partitioning and strategy exploration," in *Formal Methods in Computer-Aided Design (FMCAD)*, Sep 2020.

[14] S. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. D. Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2022.

[15] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *International Conference on Computer-Aided Design (ICCAD)*, pp. 836–843, November 2006.

[16] A. Mishchenko and R. Brayton, "Integrating an AIG Package, Simulator, and SAT Solver," in *International Workshop on Logic and Synthesis (IWLS)*, June 2018.

[17] V. N. Possani, A. Mishchenko, R. P. Ribas, and A. I. Reis, "Parallel combinational equivalence checking," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Oct 2019.

[18] "Hardware Model Checking Competition 2017, Single property benchmarks," *http://fmv.jku.at/hwmcc17*.

[19] P. K. 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 (ICCAD)*, ACM, 2016.

[20] "Altera corporation/intel: Quartus II University Interface Program," *https://github.com/ispras/hdl-benchmarks/tree/master/hdl/quip*.

[21] F. Corno, M. S. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *IEEE Design & Test of Computers, https://github.com/cad-polito-it/I99T*, vol. 17, no. 3, 2000.

[22] 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 (FMCAD)*, pp. 33–51, 2002.

[23] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *International Conference on Computer-Aided Design (ICCAD)*, pp. 1076–1082, November 2005.

[24] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, p. 70–87, 2011.

[25] R. Dureja, A. Gurfinkel, A. Ivrii, and Y. Vizel, "IC3 with Interal Signals," in *Formal Methods in Computer-Aided Design (FMCAD)*, (New Haven, CT, USA), IEEE/ACM, Oct. 2021.

[26] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 401–424, April 1994.

[27] C. Wang, A. Gupta, and F. Ivancic, "Induction in CEGAR for detecting counterexamples," in *Formal Methods in Computer Aided Design (FMCAD)*, pp. 77–84, 2007.

[28] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification," in *Design, Automation and Test in Europe (DATE)*, pp. 1674–1679, Apr 2009.

[29] M. L. Case, J. Baumgartner, H. Mony, and R. Kanzelman, "Approximate reachability with combined symbolic and ternary simulation," in *Formal Methods in Computer-Aided Design (FMCAD'11)*, pp. 109–115, 2011.

[30] R. Tarjan, "Depth first search and linear graph algorithms," in *SIAM Journal on Computing*, 1972.

[31] G. Cabodi, P. Camurati, and S. Quer, "A graph-labeling approach for efficient cone-of-influence computation in model-checking problems with multiple properties," *Software: Practice and Experience*, vol. 46, no. 4, pp. 493–511, 2016.

[32] R. Dureja, J. Baumgartner, A. Ivrii, R. Kanzelman, and K. Y. Rozier, "Boosting verification scalability via structural grouping and semantic partitioning of properties," in *Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–9, Oct 2019.

[33] "Hardware Model Checking Competition 2010-2017," *http://fmv.jku.at*, Selected all non-liveness benchmarks, filtered to largest file sizes. Fig. 10b prunes timeout-vs-timeout.

[34] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 159–173, 2004.

[35] M. Mann and V. Putsche, "Semiconductor: Supply chain deep dive assessment," February 24, 2022. United States. https://www.osti.gov/biblio/1871585 *"The global energy use of products featuring semiconductors has doubled every three years since 2010 primarily due to the accelerating use of semiconductors in all facets of our modern economy and the deceleration of energy efficiency increases due to miniaturization. This exponential growth in energy use is projected to accelerate..."*.