

Optimal Redundancy Removal without Fixedpoint Computation

Michael Case Jason Baumgartner Hari Mony Robert Kanzelman
IBM Systems and Technology Group

Abstract—Industrial verification and synthesis tools routinely identify and eliminate redundancies from logic designs. In the former case, redundancy removal yields critical speedups to the overall verification process. In the latter case, redundancy removal constitutes a primary mechanism to optimize the final fabricated circuit. Redundancy identification frameworks often utilize a greatest-fixedpoint iteration, initially postulating a set of candidate redundancies to be conjunctively proved then refining candidates based upon failed proof attempts. Such procedures generally do not yield *any* soundly-proved redundancies until a fixedpoint is achieved. In this paper, we overcome this drawback by augmenting the fixedpoint procedure with a set of efficient techniques to track dependencies between candidate redundancies. This approach enables the identification of an optimal subset of valid redundancies before the fixedpoint is reached, and may also be used to reduce the number of computations within the fixedpoint procedure. We apply our techniques to enhance k -induction as well as a more general transformation-based verification flow. For induction, we demonstrate up to 75% reduction in runtime and 97% reduction in the number of inductive proofs. For the more general flow, we demonstrate up to 90% reduction in runtime and 80% reduction in the total number of proof obligations.

I. INTRODUCTION

Industrial gate-level designs are often rife with redundancy. Logic synthesis tools attempt to eliminate redundant structure as a way of improving the area, delay, or power of the final fabricated circuit. Verification tools eliminate redundancy to reduce the size of the design under verification, often yielding dramatic speedups to the overall verification process, e.g. [1], [2]. In *equivalence checking* frameworks, internal equivalences between two designs can be viewed as a set of redundancies, which once identified and eliminated, effectively decompose an otherwise intractable monolithic problem for greater scalability.

Redundancy identification frameworks often operate through a greatest-fixedpoint iteration to yield a maximal set of equivalent gates which can be proved to assume identical values in every reachable state¹. Such frameworks often postulate a superset of candidate equivalences, e.g. identified using simulation signatures or structural heuristics, then iteratively attempt to prove the conjunction of this postulated set. Any inaccurate or unprovable equivalences are discarded, and the process repeats until a fixedpoint is achieved. The benefit of the fixedpoint procedure is that it enables cross-leveraging postulated equivalences, i.e., *assuming* one set of postulated equivalences when *proving*

another. This often yields dramatic speedups, e.g., through enabling inductive proofs of redundancy which otherwise may require reachability analysis [3], [4], [1]. *Speculative reduction* may leverage assumptions to further reduce proof complexity by *merging* fanout references of postulated-equivalent gates, trivializing many proof obligations and simplifying the remainder [2], [5]. The drawback of cross-leveraging equivalences in this manner is that until a fixedpoint is achieved, no redundancy may be inferred because any successfully-completed equivalence proofs may be jeopardized by inaccurate candidate equivalences.

k -Induction is commonly used to scalably prove candidate equivalences [3], [1]. k -Induction first validates the *base case* by checking that the postulated equivalences hold on every state reachable in k or less steps from the initial states. Next, the *inductive step* validates that for all sequences of k consecutive states on which the postulated equivalences hold, they also hold in all successor states. If either check fails, the inaccurate or unprovable equivalences are discarded, and the fixedpoint process is repeated on the remaining equivalences.

More generally, redundancies may hold in a design which cannot be readily proved using induction. To identify such redundancies, one may need to leverage an arbitrary sequence of reduction, abstraction, and proof techniques to adequately simplify and ultimately prove postulated gate equivalences – often represented as verification properties termed *miters*. The use of verification-oriented transformations such as min-area retiming [6] and temporal decomposition [7] are particularly valuable in a redundancy removal framework, as they may eliminate structural differences between the logic being checked for equivalence. Speculative reduction is furthermore often critical to simplify the resulting set of proof obligations, both in enhancing the utility of other transformations and abstractions, as well as in simplifying the final proof obligation for a technique such as interpolation [8]. We refer to such a verification paradigm as *Transformation-Based Verification (TBV)* [6]. As with induction, if any miter is falsified or unproved by a given TBV algorithm sequence, the corresponding equivalences must be discarded, and the fixedpoint process is repeated on the remaining equivalences.

In this work we address the optimal identification of redundancies in an assume-then-prove framework without requiring fixedpoint computations. In particular, we present efficient techniques to track proof dependencies within inductive and TBV-based redundancy identification frameworks. Our techniques enable the identification of a subset of true redundancies before the fixedpoint is reached, despite any

¹Constant gates and antivalent gates may be identified using a straightforward extension of such frameworks.

Alg. 1. Redundancy Removal Fixedpoint Algorithm

```

1: function identifyRedundancyFixedpoint()
2:   Postulate redundancy candidates, represented as equivalence classes
3:   loop
4:     Attempt to prove each redundancy candidate as accurate
5:     if (all redundancy candidates are proved) then
6:       return equivalence classes as redundancies that may be merged
7:     else
8:       refine the equivalence classes
9:     end if
10:  end loop
11: end function

```

cross-leveraged assumptions. This has several benefits: **(1)** we can soundly identify redundancies even when resource limits prevent *every* candidate equivalence from being proved or disproved; **(2)** we reduce effort within a fixedpoint procedure by not requiring candidate equivalences to be repetitively proved across iterations; and **(3)** within each iteration of the fixedpoint computation, we allow the proofs of unsolved equivalences to be deferred or discarded when we detect that it is not possible to mark this redundancy as soundly proved.

Section II describes the preliminaries. Section III describes the Proof Graph, our datastructure which tracks dependencies among equivalences. Sections IV and V describe the integration of Proof Graph techniques in inductive and TBV frameworks, respectively. In Section VI we provide proofs that our techniques are sound and optimal. Lastly, we provide experimental results in Section VII.

II. PRELIMINARIES

We assume that the design under analysis is represented as a gate-level netlist, consisting of combinational gates of various types as well as sequential elements with associated *initial values* and *next-state functions*. Our implementation uses an And/Inverter Graph [9], [10] format, though our techniques are applicable to other netlist formats as well. In a verification setting, the netlist may also comprise logic expressing environmental assumptions and correctness properties. In an equivalence checking setting, the netlist may represent the composition of two designs being compared, with safety properties checking pairwise equivalence of primary outputs.

Algorithm 1 illustrates a traditional redundancy removal fixedpoint algorithm [3], [1], [5]. Such algorithms first postulate a superset of redundancy candidates, represented as *equivalence classes* wherein all gates within the same class are postulated to behave identically in all reachable states. A set of safety properties termed *miters* is constructed which represent the underlying equivalences candidates, and a set of proof techniques is then used to establish the validity of the miters and the candidates they represent. If any candidates are demonstrated to be inaccurate, or if the chosen proof techniques cannot yield a conclusive result for some candidates, the equivalence classes are *refined* by discarding the unprovable equivalences. This process repeats until finally all equivalence classes are demonstrated correct. When this fixedpoint is reached, the netlist may be simplified by *merging* gates which are proved equivalent. In particular, within each

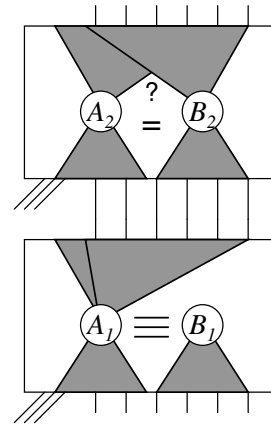


Fig. 1. Speculative reduction simplifies the unrolled netlist.

equivalence class, a *representative gate* is chosen, and each other gate in that equivalence class will be replaced with its representative in the netlist.

There are two fundamental techniques to enable the scalability of sequential redundancy removal. The first is the use of induction to establish the correctness of the *conjunction* of the postulated equivalences, which individually would often be non-inductive and require substantially more expensive proof techniques [3], [1]. Even if heavier-weight proof techniques are ultimately needed for maximal redundancy removal, conjunctive induction is often able to efficiently solve most of the proof obligations. The second is the use of *speculative reduction*, which reduces the size of the miter-annotated netlist by reconnecting the fanout of a given candidate equivalence gate (refer to gate B_1 in Figure 1) to its representative (gate A_1). This reduces the complexity of the logic in the fanout of the speculatively-merged gate and often trivializes downstream miters. Speculative reduction is capable of yielding orders of magnitude speedups in both inductive- and TBV-based approaches for redundancy removal [2], [5]. However, as a result of these two techniques, Algorithm 1 cannot generally be used to identify redundancies before a fixedpoint is reached, as one incorrect candidate may invalidate the soundness of the proof of the other candidates.

III. THE PROOF GRAPH

To deduce sound redundancies prior to achieving a fixedpoint, we record *dependencies* between candidate equivalences. Two sources of dependencies may arise in a sequential redundancy removal framework. **(1)** Speculative reduction may simplify the netlist under the assumption that $A \equiv B$, e.g. by merging the fanout B onto A as in Figure 1. If the merged gate B is in the *cone-of-influence (COI)* of some other postulated equivalence $C \equiv D$, then $C \equiv D$ depends on $A \equiv B$. **(2)** If using induction, the inductive hypothesis constrains the SAT solver to only explore state sequences where $A \equiv B$ and $C \equiv D$ on the first k time steps. If the solver utilizes these inductive hypotheses, then $A \equiv B$ depends on $C \equiv D$ and vice-versa. If $C \equiv D$ depends on $A \equiv B$ and we can

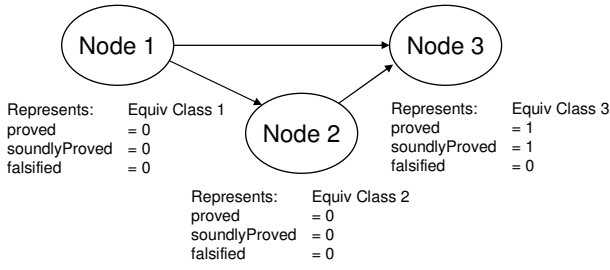


Fig. 2. An example Proof Graph

Alg. 2. Proof callback function

```

1: function informProved(proofGraph, class)
2:   ensure that the proofGraph is condensed
3:   node = the Proof Graph node containing class
4:   node.proved[class] = 1
5:   if (( $\forall$  classes C  $\in$  node, node.proved[C] == 1) and
        ( $\forall$  children D of node, D.soundlyProved == 1)) then
6:     node.soundlyProved = 1
7:     inform the calling application that node's classes are soundly proved
8:     for all parents P of node, P.proved[*] == 1 do informProved(P)
9:   end if
10: end function

```

demonstrate that $A \not\equiv B$ then a proof of $C \equiv D$ does not soundly indicate that C and D are equivalent.

Dependencies are recorded in a directed graph called the *Proof Graph*. Each node in the Proof Graph represents a set of one or more equivalences. An edge $node_1 \rightarrow node_2$ represents the dependency $node_1$ “depends on” $node_2$. An example Proof Graph is shown in Figure 2.

We initially construct the Proof Graph to represent a single equivalence class per node. The resulting Proof Graph is cyclic in general, though we may render it acyclic without jeopardizing the optimality of identified redundancies in two ways. First, all *strongly connected components* (SCCs) [11] are identified, and the graph is *condensed* by collapsing the nodes in each SCC into a single Proof Graph node. Second, self-edges are suppressed. In this way, each Proof Graph node thus represents a set of equivalence classes.

We use the Proof Graph within a redundancy identification framework to identify when a proof represents a soundly-proved redundancy. In our algorithms we use three types of flags within the Proof Graph: (1) *proved* means that a given equivalence class has been proved relative to the other (possibly incorrect) redundancy candidates; (2) *soundlyProved* means that the proof of the corresponding equivalence class(es) is sound; and (3) *falsified* means that either this node contains a falsified equivalence, or it has a falsified dependency. A Proof Graph node has a single *soundlyProved* and *falsified* flag, and a *proved* flag for each equivalence class within that node. Because the topology of the Proof Graph depends upon the nature of the equivalence classes, the Proof Graph and its flags generally must be recomputed at each iteration of the fixedpoint Algorithm 1.

Algorithm 2 is called when all miters corresponding to a postulated equivalence class are proved. We set the *proved* flag on this class and conclude that this proof is sound iff all

Alg. 3. Falsification callback function

```

1: function informFalsified(proofGraph, class)
2:   node = the Proof Graph node containing class
3:   if (node.falsified == 1) then return
4:   node.falsified = 1
5:   for all parents P of node do informFalsified(proofGraph, P)
6: end function

```

classes in the same SCC are proved and all dependencies are soundly proved. Whenever Algorithm 2 deduces that a proof is sound, it recurses to the parents in the Proof Graph as the proofs of these parent classes may now be sound as well. As an example of Algorithm 2, if we call `informProved` on Class 2 of Figure 2 then we deduce that this proof is sound because all classes in Class 2’s SCC are proved and the only dependency, Node 3, is soundly proved.

Algorithm 3 is called whenever an equivalence class is falsified. This sets the falsified flag on the corresponding Proof Graph node and propagates this flag to all ancestors in the Proof Graph. This flag is used to inform the higher-level algorithms that an equivalence class cannot be soundly proved and therefore need not be checked. As an example, calling `informFalsified` on Class 2 of Figure 2 will result in the falsified flag being set on Proof Graph Nodes 2 and 1. Node 1 can thereafter never be soundly proved, and the higher-level algorithms can use this information to forgo any attempts to prove the equivalences from Node 1.

Using the Proof Graph within a redundancy removal framework will not alter the set redundancies that are proved, as will be established in Theorem 2. Instead, the Proof Graph is used to improve the performance of the associated redundancy removal framework.

The Proof Graph is a general way to track dependencies. In this work, we apply this datastructure in the context of induction (Section IV) and TBV (Section V).

IV. INDUCTION AND THE PROOF GRAPH

In this section we enhance inductive redundancy identification frameworks using the Proof Graph. In induction, there are two types of dependencies that must be recorded in the Proof Graph: combinational structural dependencies, and proof dependencies.

An inductive proof unrolls the transition relation, performing speculative reduction to simplify the unrolled logic [2]. For example, in Figure 3A, the lower time frame will be simplified by assuming $A_1 \equiv B_1$ and $C_1 \equiv D_1$. If these assumptions are invalid, the behavior of the downstream logic may be altered, implying that downstream miters are dependent on these speculatively-reduced equivalences.

Algorithm 4 describes the process to infer such dependencies, called *combinational structural dependencies*. Given an equivalence class, the unfolding depth k used for induction, and a set of gates whose fanout was merged due to speculative reduction, we first mark the COI of all miters within the class. Then within this COI, we find gates that have been merged by speculative reduction. The given equivalence class will be

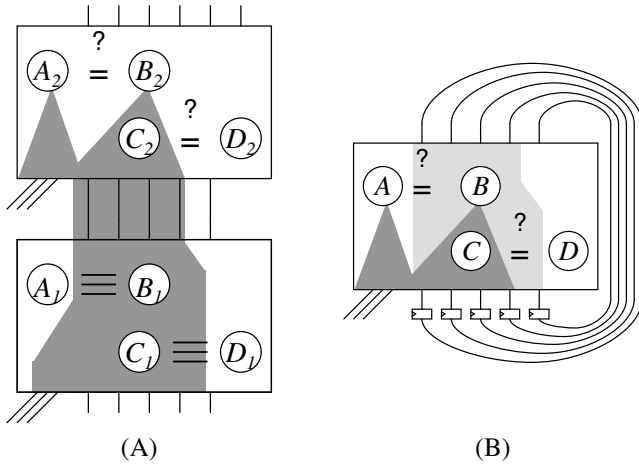


Fig. 3. (A) A combinational structural dependency, affecting induction. (B) A sequential structural dependency, affecting TBV.

Alg. 4. Discovery of combinational structure dependencies

```

1: function getCombStructureDeps(class, k, specReduction)
2:   coi =  $\emptyset$ 
3:   for all gate in class do
4:     u = unrolled instance gate in frame k
5:     coi = coi  $\cup$  combinational cone of influence of u
6:   end for
7:   for all simplifiedGate in specReduction  $\cap$  coi do
8:     C = equivalence class that spec-reduces simplifiedGate
9:     record the dependency "class  $\rightarrow$  C"
10:  end for
11: end function

```

marked as dependent upon all classes responsible for these simplifications.

A second type of dependency arises from the inductive hypothesis. In k -induction we hypothesize that all equivalences hold at times $0, \dots, k-1$. These hypotheses are typically implemented by passing additional constraints to the SAT solver, causing it to only explore paths for which the equivalences hold in the first k steps. If the proof of a miter depends on the inductive hypothesis, then the miter and its associated equivalence class have an additional dependency.

Algorithm 5 shows how to prove the miters from an equivalence class and extract the resultant dependencies, termed *proof dependencies*. A SAT solver is used to test the conjunction of all miters along with all inductive hypotheses. If the result is unsatisfiable, meaning the miters are proved, we extract an unsatisfiable core from the solver and inspect it to determine which hypotheses were utilized in the proof.

Alg. 5. Discovery of proof dependencies

```

1: function proveAndGetDeps(class, hypotheses)
2:   result = SAT_solve( $\bigwedge_{\text{miter} \in \text{class}} \text{miter} \wedge \text{hypotheses}$ )
3:   if (result is "unsatisfiable") then
4:     core = extract an unsatisfiable core from the SAT solver
5:     for all hyp in hypotheses  $\cap$  core do
6:       C = equivalence class responsible for hyp
7:       record the dependency "class  $\rightarrow$  C"
8:     end for
9:   end if
10:  return result
11: end function

```

Alg. 6. Determining the order in which to prove equivalences

```

1: function getProofObligations(proofGraph)
2:   ensure that the proofGraph is condensed
3:   classesToProve =  $\emptyset$ 
4:   for all condensed node  $\in$  proofGraph, node.falsified == 0 do
5:     if  $\forall$  children C of node, C.soundlyProved == 1 then
6:       classesToProve = classesToProve  $\cup$  {node's classes}
7:     end if
8:   end for
9:   return classesToProve
10: end function

```

Each hypothesis has an associated equivalence class C , and we record the dependence on each such C . Techniques to minimize the unsatisfiable core [12] may be employed to minimize these dependencies if desired.

Note that proof dependencies render the Proof Graph a dynamic datastructure when it is used for induction. Edges may be added after any single SAT call, and the topology of the Proof Graph can thus change. This is why Algorithm 2 may need to re-condense the Proof Graph.

With combinational structural dependencies and proof dependencies identified, we may use the Proof Graph techniques from Section III to reason about the equivalences that have been soundly proved in a single iteration of induction. Soundly-identified redundancies can be obtained despite inaccurate or unproved candidate equivalences, before the inductive fixedpoint is reached. This gives us partial results in the case that computational resources are exhausted before induction converges. In addition, if an equivalence is soundly proved during one induction iteration, the equivalence doesn't need to be re-tested during later induction iterations. As our experiments demonstrate, this dramatically reduces the number of SAT calls without jeopardizing the optimality of the final derived set of redundancies.

The Proof Graph can also be used to detect equivalence classes that cannot be soundly proved because they have a falsified dependency. We can skip these proof attempts during induction, further reducing the overall number of SAT calls without sacrificing the optimality of soundly proved equivalences as per Theorem 2.

Algorithm 6 may be used to derive an optimal ordering of equivalence classes to be proved by an induction framework. The induction framework repeatedly calls this function until no more equivalence classes need to be tested in the current induction iteration. The algorithm traverses the Proof Graph to look for nodes that are not falsified and have no unproved children. These represent the equivalence classes that if proved are most likely to yield sound equivalences, hence induction is directed to test these classes first. Because the Proof Graph is maintained to be acyclic, this algorithm is guaranteed to return a nonempty set of equivalence classes if any unsolved classes may yield a soundly-proved equivalence. Note that if Algorithm 3 sets the *falsified* flag, then induction will entirely skip any proof attempts for the corresponding candidate equivalences.

 Alg. 7. Discovery of sequential structure dependencies

```

1: function getSeqStructureDeps(class, specReduction)
2:   coi = sequential cone of influence of all gate in class
3:   for all simplifiedGate in specReduction  $\cap$  coi do
4:     C = equivalence class that spec-reduces simplifiedGate
5:     record the dependency "class  $\rightarrow$  C"
6:   end for
7: end function
  
```

V. TBV AND THE PROOF GRAPH

Like induction, TBV can be used to prove that equivalences hold on every reachable state. Here the set of algorithms used to carry out a proof may be arbitrary. The netlist is transformed by adding miters for the suspected equivalences, speculatively reducing the (sequential) netlist, and passing this sub-problem to another user-specified algorithm or sequence of algorithms.

When the Proof Graph is used in a TBV context, there is only one type of dependency: those arising from speculative reduction. An example of this speculative reduction is shown in Figure 3B, where (1) the netlist is simplified assuming $A \equiv B$ and $C \equiv D$ by moving fanouts of A to B and fanouts of D to C , and (2) miters are added to test $A \equiv B$ and $C \equiv D$.

Algorithm 7 is used to extract speculative reduction dependencies, termed *sequential structure dependencies*, for TBV. This function is called once on each equivalence class, and it is passed the class and the set of gates merged using speculative reduction. The sequential COI of all miters in the class is marked, and simplifications within this COI are explored. For each simplification, a dependence on the associated class C is recorded.

As with Algorithm 6, it is advantageous to prove miters associated with leaves of the Proof Graph before attempting to prove other miters. In our implementation, we influence the proof order by associating assigning a priority to each miter. Additionally, we instruct downstream algorithms to skip proofs of miters associated with Proof Graph nodes that have the *falsified* flag set.

VI. SOUNDNESS AND OPTIMALITY

Our first theorem establishes the validity of any redundancy identified using our techniques.

Theorem 1 (Soundness): Any redundancy identified as “soundly proved” using the Proof Graph is valid.

Proof: If no speculative reduction or conjunctive induction is used within the underlying proof framework, the Proof Graph is unconnected hence this theorem trivially holds.

Speculative reduction may jeopardize the validity of a proof, since the corresponding fanout merge may alter netlist behavior if the corresponding postulated equivalence is incorrect. Note however that a speculative merge only may alter the behavior of gates in the fanout of the *merged* gate: not the *representative* onto which it was merged. Any miter in the fanout of this merged gate will have an associated edge in the Proof Graph, hence such fanout miters will be marked as *falsified* if the speculatively-merged gate is demonstrated inaccurate. Furthermore, no proved miter in the fanout of this speculatively-merged gate will be identified as “soundly proved” until the

corresponding candidate equivalence is soundly proved and it is thereby guaranteed that the speculative merge does not alter netlist behavior. This theorem thus follows for speculative reduction given the results of [2], particularly that speculative reduction preserves the ability to identify invalid equivalences.

If using conjunctive induction, recall that a Proof Graph edge is added to any postulated equivalence upon which another equivalence proof is determined to rely. This will ensure that no proved equivalence will be identified as sound until the corresponding source of the necessary inductive hypothesis has been proved as accurate, thereby validating the soundness of using that hypothesis. ■

The following theorem establishes the optimality of the identified redundancies when using *fine-grained equivalence classes*, wherein each equivalence class contains a pair of gates: one to be merged onto the other representative. Coarser-grained equivalence classes are possible, though may trade reduction optimality for performance.

Theorem 2 (Optimality): Given fine-grained equivalence classes, the set of redundancies derived when using the Proof Graph is *optimal*. In particular, any proof discarded via use of the Proof Graph could not correlate to a soundly-identified redundancy under the chosen proof framework.

Proof: First consider the use of speculative reduction. Every miter in the fanout of a speculatively-merged gate will have an associated dependency identified in the Proof Graph, and thus will not be demonstrated as soundly proved until the speculatively-merged gate itself is demonstrated accurate. We note that this set of dependencies is minimal in that dependencies are limited to precisely those gates whose behavior would be altered if the corresponding postulated equivalence is invalid. Note that collapsing SCCs within the Proof Graph does not affect the minimality of this transitive dependency.

Next consider the use of conjunctive induction as the chosen proof framework, where the Proof Graph might additionally contain proof dependencies. If a candidate equivalence $e1$ cannot be proved, and another candidate proof $e2$ is proved using the inductive hypothesis of $e1$, the proof of $e2$ will be discarded along with all other candidates which transitively depend on $e1$. Such invalidation is necessary for soundness, since otherwise a potentially-invalid hypothesis would be used as the basis of a proof. Use of an unsatisfiable core furthermore ensures a *minimal* set of such dependencies and hence invalidations, whereas a traditional framework would require invalidating *all* proofs due to risk of such unsoundness. In general, transitive dependencies may include edges of both types. Optimality of identified redundancy follows noting that both types of dependencies are minimally identified. ■

VII. EXPERIMENTAL RESULTS

All techniques described in this paper have been implemented in the IBM internal verification tool *SixthSense* [9]. We utilize two disjoint benchmark suites:

- 1300 industrial property checking and sequential equivalence checking benchmarks. These designs are derived primarily from IBM high-performance microprocessors

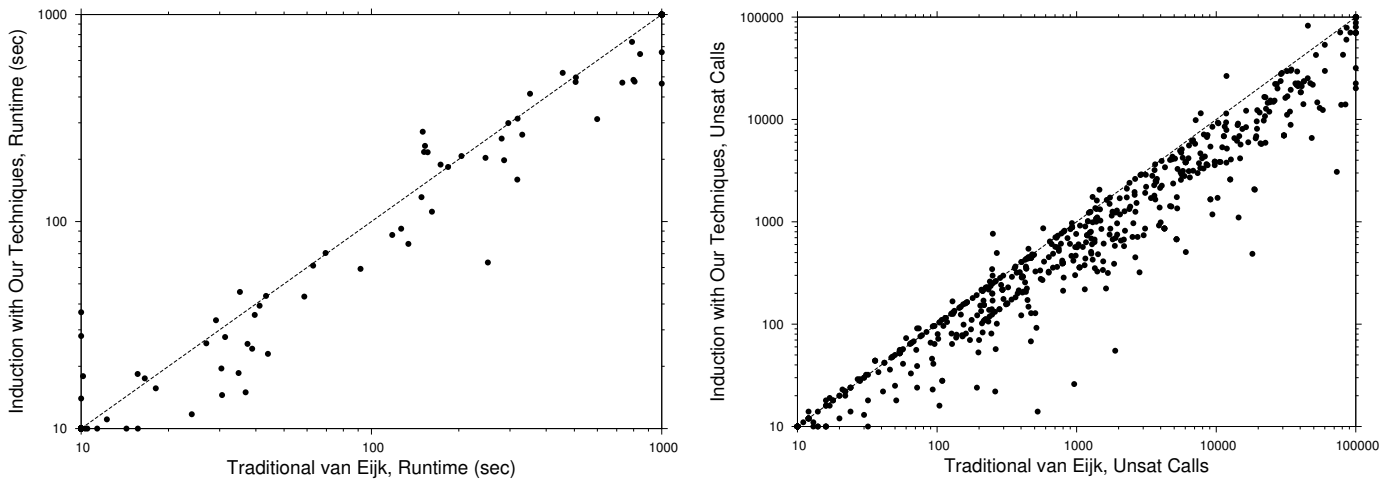


Fig. 4. Finding redundancies with $k = 1$ induction on 1300 IBM designs. Left: runtime, Right: number of unsatisfiable miters

and range in size up to 5.3M AIG AND gates and 330k registers.

- The publicly available HWMCC’10 benchmarks [10].

All experiments were run on a cluster of 16 GB, 2 GHz POWER 5 workstations.

A. Induction Results

We first examine the impact of the Proof Graph on induction. We preprocessed each netlist with combinational simplifications [13], light-weight sequential simplifications, phase abstraction [14], transient elimination [7], and input reparameterization [15]. Next, we use 640 passes of 32-cycle random simulation to derive candidate gate equivalences, and we use $k = 1$ induction to prove these equivalences. This flow was repeated twice: with the techniques presented in this paper, and without our techniques in a more traditional flow referred to as “van Eijk” below.

Figure 4 shows the difference in induction runtime of the van Eijk flow vs. our proposed algorithms on the IBM designs. Our techniques improve the runtime on almost all designs, and the maximum reduction in runtime is 75%. The occasional slowdowns are cases where the order of SAT calls imposed by the Proof Graph (Algorithm 6) is disadvantageous². With our proposed methods, the order in which the miters are tested is influenced by the structure of the Proof Graph, while in the van Eijk flow we test the miters in topological order. In our implementation, we utilize incremental SAT which makes the ordering significant.

The runtime improvement is primarily due to the reduction in the number of SAT calls made by the induction package. Methods exist to reduce the number of SAT calls that are satisfiable – re-simulation of inductive counterexamples to quickly detect satisfiable miters [5]. Using the techniques described in this paper we are able to furthermore reduce the number of unsatisfiable calls. We do this by (1) avoiding

²Note that there is overhead associated with maintaining the Proof Graph. In our implementation, this overhead is minimal, and the change in the ordering of SAT calls is responsible for any slowdowns in the cases have studied.

re-testing soundly proved equivalences in the later induction iterations, and (2) skipping SAT calls for equivalences that cannot be soundly proved. Figure 4 shows a comparison of the number of unsatisfiable SAT calls on the IBM designs. Our techniques reduce the number of unsatisfiable calls by 25% on average and 97% in cases³.

Figure 5 analyzes the performance of our induction implementation on a subset of the most challenging HWMCC’10 benchmarks. In most cases, our techniques improves runtime significantly, by 11% on average and 70% in cases. As with the IBM benchmarks, the runtime improvement is primarily due to a reduction in the number of unsatisfiable SAT calls, 37% on average and 92% in cases.

When we enable our Proof Graph algorithms the number of iterations increases slightly, 17% on average. The reason is that because SAT calls are skipped, inductive counterexamples may not be seen in the earlier iterations. This causes the equivalence classes to not be refined as aggressively as in a traditional flow. However, the net decrease in the number of SAT calls makes up for the slight increase in induction iterations.

Figure 5 also shows the number of merges. When our induction package deduces that an equivalence is soundly proved, it merges the equivalence and simplifies the design. *Early merges* are merges that are performed before the fixedpoint is reached. We can perform a significant percentage of the merges early, 37% on average. In one case, `bjrb07amba10andenv`, we hit an induction timeout of 1200 seconds and thus all merges were early merges – there was no fixedpoint.

B. TBV Results

Next we examine the impact of our algorithms on TBV. As in Section VII-A we aggressively pre-process the design and use random simulation to postulate register equivalences. We prove these suspected equivalences with 1-induction and apply TBV on those equivalences which are suspected to hold but

³Because other aspects of the redundancy removal framework consume significant runtime, e.g. formulation of the SAT problem and resimulation of counterexamples, the reduction in the number of unsatisfiable SAT calls is not directly proportional to the reduction in the total runtime.

Benchmark	Preprocessed Size		van Eijk					Our Techniques				
	Ands	Reg.	Time	Iter.	Total Sat Calls	Unsat. Sat Calls	Total Merges	Time	Iter.	Total Sat Calls	Unsat. Sat Calls	Early Merges
bj08amba5g62	12411	39	91.6	4	27639	27490	4532	59.0	5	20237	20095	0
bjrb07amba10andenv	63127	58	1204.5	2	38230	38066	138	1200.3	3	29634	29454	138
bjrb07amba3andenv	5473	30	8.1	3	7727	7697	1912	8.4	3	4691	4662	12
bjrb07amba4andenv	13478	33	30.5	3	7955	7926	2473	14.5	4	4207	4175	2463
bjrb07amba5andenv	15063	38	134.0	4	22827	22738	4234	78.0	4	12786	12718	23
bjrb07amba6andenv	23622	41	295.9	3	26527	26437	5759	299.1	4	22323	22228	32
bjrb07amba7andenv	22198	45	173.0	3	16493	16383	4113	188.3	4	12285	12174	40
bjrb07amba9andenv	45539	52	1200.5	5	81087	80956	11065	657.7	5	43015	42894	106
bob1u05cu	12201	2146	6.7	34	25027	24305	746	5.2	87	15177	14753	63
bobmitersynbm	31015	5984	43.3	31	79981	78684	3225	43.7	58	15152	13905	3223
bobsmcodic	18447	1850	15.6	5	6212	6056	954	6.0	8	596	507	954
bobsmmem	55105	3584	18.0	8	13021	12740	1873	15.6	10	4350	4063	1852
bobsmrisc	9422	1323	2.7	5	8666	8587	7329	8.4	5	6813	6732	0
bobsynthetic2	2387	24	161.5	45	85792	85706	1345	111.6	44	60293	60208	0
bobuns2p10d20l	2229	20	351.7	2	226	223	283	414.5	2	231	228	0
mentorbm1and	17628	3138	9.4	44	41483	40990	3536	8.8	44	21372	21105	3536
mentorbm1p02	12255	2111	10.3	42	38285	37746	1231	6.2	43	22532	21954	252
mentorbm1p03	12254	2111	7.1	43	39087	38578	1229	8.9	42	22750	22186	252
mentorbm1p04	12282	2117	8.6	43	39012	38495	1260	5.6	42	22742	22177	252
mentorbm1p05	12290	2119	7.3	43	39415	38891	1270	5.8	43	22823	22258	252
mentorbm1p07	17465	3109	11.3	40	37433	36960	3413	9.0	41	23033	22490	280
mentorbm1p08	12273	2115	7.4	44	39863	39342	1251	6.2	42	22711	22147	252
mentorbm1p09	12253	2111	7.8	43	39104	38576	1230	5.9	43	23114	22533	252
mentorbm1p10	12253	2111	6.8	42	38176	37646	1225	4.9	45	22079	21790	1225
mentorbm1p12	12288	2114	7.5	43	39069	38575	1231	5.8	43	21421	21149	1231
neclafp1001	35903	5360	204.2	7	78296	77352	24234	207.1	8	71842	70920	96
neclafp1002	35734	5360	280.8	8	86909	85985	24167	251.7	8	79691	78766	287
neclafp2001	21240	3478	12.2	4	29614	29596	23381	11.0	4	28528	28510	0
neclafp2002	21891	3478	4.0	4	29112	29095	23682	4.7	4	27802	27785	0
pdtpmvip	15066	574	10.1	2	10364	10277	5960	17.9	3	9396	9304	0
pj2002	16769	686	4.6	3	9766	9756	3250	1.4	3	3882	3872	2935
pj2003	16769	686	4.4	3	9766	9756	3250	1.7	3	3882	3872	2935
pj2006	16855	702	4.5	3	9773	9756	3248	1.8	3	3589	3572	2935
			1.00	1.00	1.00	1.00	1.00	0.89	1.17	0.63	0.62	0.37

Fig. 5. Finding redundancies with $k = 1$ induction on a subset of the HWMCC'10 designs

were unproved with induction. This TBV flow speculatively reduces the sequential netlist, annotates it with miters, and passes it downstream to first a combinational simplification engine and then an interpolation engine. We repeat this flow twice: once using our new techniques, and again with the Proof Graph disabled.

Figure 6 shows the TBV runtime on 93 of the most difficult IBM designs⁴. Nearly all runtimes are greatly improved by our Proof Graph techniques. We improve the runtime by 90% in cases and 18% cumulatively. The primary causes for these improvements are: (1) early merging prevents later TBV iterations from needing to re-prove what was soundly proved in earlier iterations, and (2) we use the proof graph to guide the downstream algorithms, only attempting proofs where a proved equivalence can lead directly to a merge, similar to Algorithm 6.

Figure 6 also shows the number of times interpolation was used to solve a miter. Our techniques are able to reduce the number of properties that interpolation attempts to prove by 80% in cases, 13% on average. Because each interpolation call has a 30-second time limit, by reducing the number of interpolation calls we improve the runtime substantially.

⁴Our aggressive pre-processing proves all properties in many of our benchmark designs.

VIII. RELATED WORK

There has been much work in the field of sequential redundancy identification. Due to space limitations, we limit our focus to more recent work which transitively subsumes prior foundational work.

[16] proposes an incremental version of a redundant latch fixedpoint similar to Algorithm 1, using 1-step induction to correlate latches for combinational equivalence checking (CEC) frameworks. The induction itself is performed using an off-the-shelf CEC tool. The authors propose that the effort of the CEC tool in finding internal equivalence points at each iteration may be simplified by avoiding re-verification of internal equivalence points driven solely by latches which did not change in correlation since they were last proved. This result relates to our ability to infer soundly-proved equivalences before all proofs are completed. However, there are several differences from our work: (1) We may soundly identify and leverage redundancy even before a fixedpoint is reached, whereas their technique requires a fixedpoint in being leveraged solely for CEC. (2) Our approach is designed to handle general k -induction as well as arbitrary TBV flows to identify redundancies over arbitrary gates in the netlist, while their approach is focused upon 1-induction to identify latch equivalence using a CEC tool. (3) Our approach is robust enough to handle inductive hypothesis constraints while their

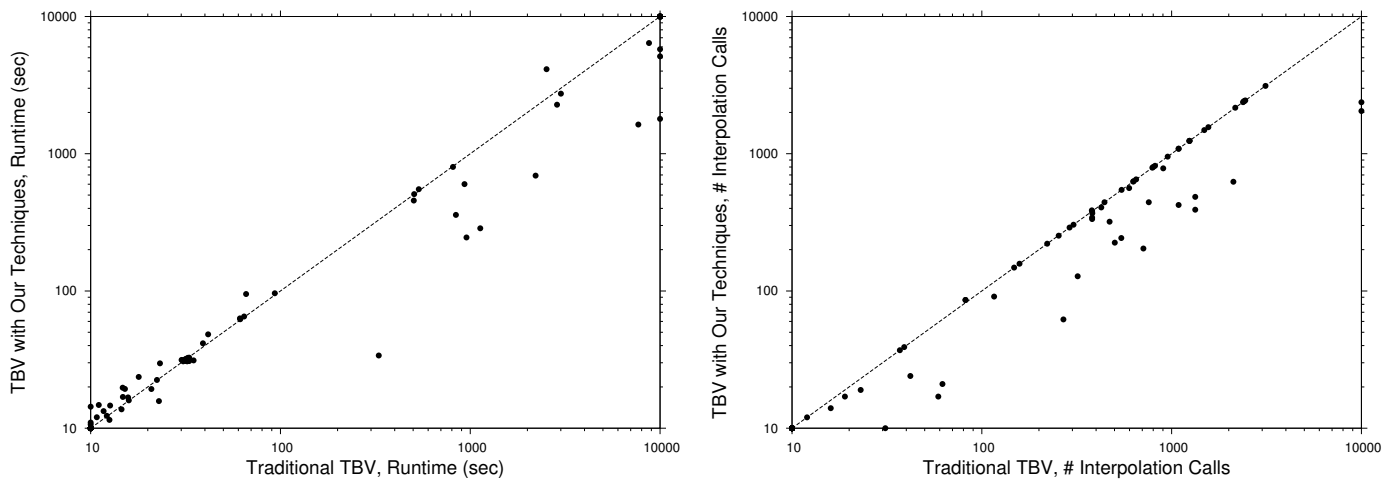


Fig. 6. Finding non-inductive redundancies with TBV on the 93 most difficult IBM designs. Left: runtime, Right: number of calls to interpolation [8]

approach need not consider them, as such hypotheses in their more limited settings are effectively “latch mappings.”

[2] discusses how one may leverage postulated equivalences through speculative reduction, enabling greater simplification of the resulting netlist for enhanced bounded or unbounded proof analysis. However, this work does not provide a method to soundly simplify the netlist until all equivalence proof obligations are proved – hence does not offer early merging capability. In addition [2] is typically implemented by using a SAT solver test each suspected equivalence at every iteration of the fixedpoint procedure, a complexity we strive to avoid.

[5] describes a method to minimize the number of satisfiable SAT calls through re-simulation of induction counterexamples, which combined with speculative reduction yields up to 5 orders of magnitude speedup on a cumulative benchmark suite. However, aside from eliminating “implied” proofs via speculative reduction, this work does not address how to minimize the number of unsatisfiable calls, which is a primary contribution of this paper. This work is nonetheless complementary to ours, as we have also found it useful to aggressively resimulate induction counterexamples to rule out satisfiable induction queries.

IX. CONCLUSION

We have presented a method to improve the efficiency of redundancy identification frameworks by tracking dependencies between redundancy candidates. The dependencies are tracked using a datastructure called the Proof Graph, which is applied to enhance both inductive and transformation-based redundancy identification frameworks. Our techniques provide numerous benefits to redundancy identification frameworks.

- Many redundancies may be determined to be soundly proved before reaching a fixedpoint, allowing for useful reduction in the design size in the event that computational resources are exhausted, or an *incomplete* proof method is used.
- The total proof burden is reduced because soundly proved redundancies need not be re-proved in later fixedpoint iterations.

- The proof burden is additionally reduced because the Proof Graph allows us to identify redundancies which can never be soundly proved under a given set of candidates. The proofs of such redundancies can be skipped.

Experiments confirm that our techniques reduce the number of attempted proofs by up to 97%, and improve runtime by up to 75%, for redundancy identification frameworks on industrial as well as public benchmark sets.

REFERENCES

- [1] P. Bjesse and K. Claessen, “SAT-based verification without state space traversal,” in *FMCAD*, Nov. 2000.
- [2] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, “Exploiting suspected redundancy without proving it,” in *DAC*, June 2005.
- [3] C. A. J. van Eijk, “Sequential equivalence checking without state space traversal,” in *DATE*, Feb. 1998.
- [4] D. Stoffel and W. Kunz, “Record & play: A structural fixed point iteration for sequential circuit verification,” in *Int’l Conference on Computer-Aided Design*, Nov. 1997.
- [5] H. Mony, J. Baumgartner, A. Mishchenko, and R. K. Brayton, “Speculative reduction-based scalable redundancy identification,” in *DATE*, pp. 1674–1679, IEEE, 2009.
- [6] A. Kuehlmann and J. Baumgartner, “Transformation-based verification using generalized retiming,” in *CAV*, July 2001.
- [7] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, “Enhanced verification through temporal decomposition,” in *FMCAD*, Nov. 2009.
- [8] K. McMillan, “Interpolation and SAT-based model checking,” in *CAV*, July 2003.
- [9] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, “Scalable automated verification via expert-system guided transformations,” in *FMCAD*, Nov. 2004.
- [10] A. Biere and K. L. Claessen, “Hardware Model Checking Competition (HWMCC) 2010 benchmarks,” <http://fmv.jku.at/hwmc10>, 2010.
- [11] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.
- [12] L. Zhang and S. Malik, “Extracting small unsatisfiable cores from unsatisfiable boolean formula,” in *SAT*, 2003.
- [13] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG rewriting: A fresh look at combinational logic synthesis,” in *DAC*, 2006.
- [14] P. Bjesse and J. Kukula, “Automatic generalized phase abstraction for formal verification,” in *ICCAD*, Nov. 2005.
- [15] J. Baumgartner and H. Mony, “Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies,” in *CHARME*, Oct. 2005.
- [16] K. Ng, M. R. Prasad, R. Mukherjee, and J. Jain, “Solving the latch mapping problem in an industrial setting,” in *Design Automation Conference*, June 2003.